Freie Universität Berlin

Datenbanksysteme 2014
Projektdokumentation

Carsten Keller, Jonas Grunert, Henrique Hepp

Am Anfang war das Wort – (1/6)

Aus der ursprünglichen Anwendungsbeschreibung haben wir, im Bestreben eine möglichst minimale Datenbank zu erhalten, zunächst alle relevanten Entitäten, Attribute und Beziehungen bestimmt und ein Entity-Relationship-Modell und ein relationales Schema erstellt.

Die größten Probleme erwarteten wir in der Umsetzung und der späteren Auswertung des Spielergebnisses.

Ursprüngliches relationales Schema:

Spieler (ID, Name, Trikot_Nr, Land, Tore, Position, Vereinsname)

Verein (Name, Siege, Niederlagen, Liganame)

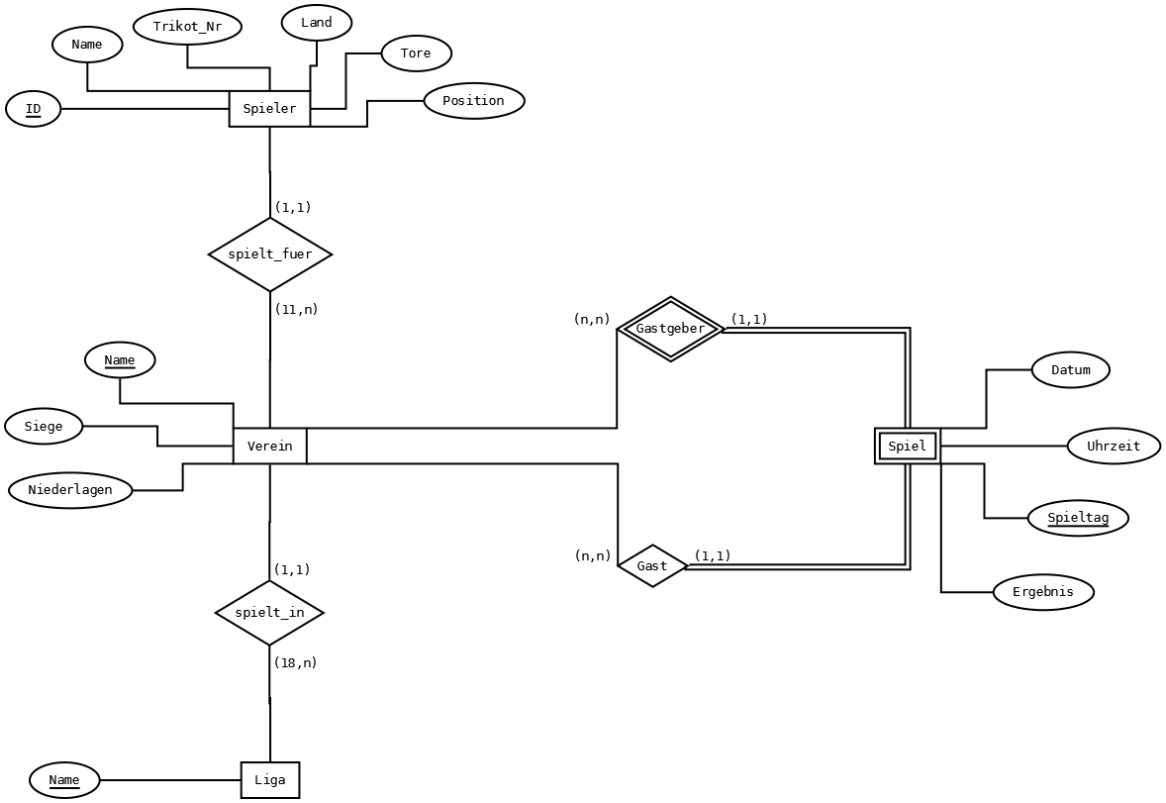
Spiele (Spieltag, Datum, Uhrzeit, Ergebnis, Gastgeber, Gast)

Liga (Name)

doppelte Unterstreichung → Einfach + gestrichelt

Am Anfang war das Wort – (2/6)

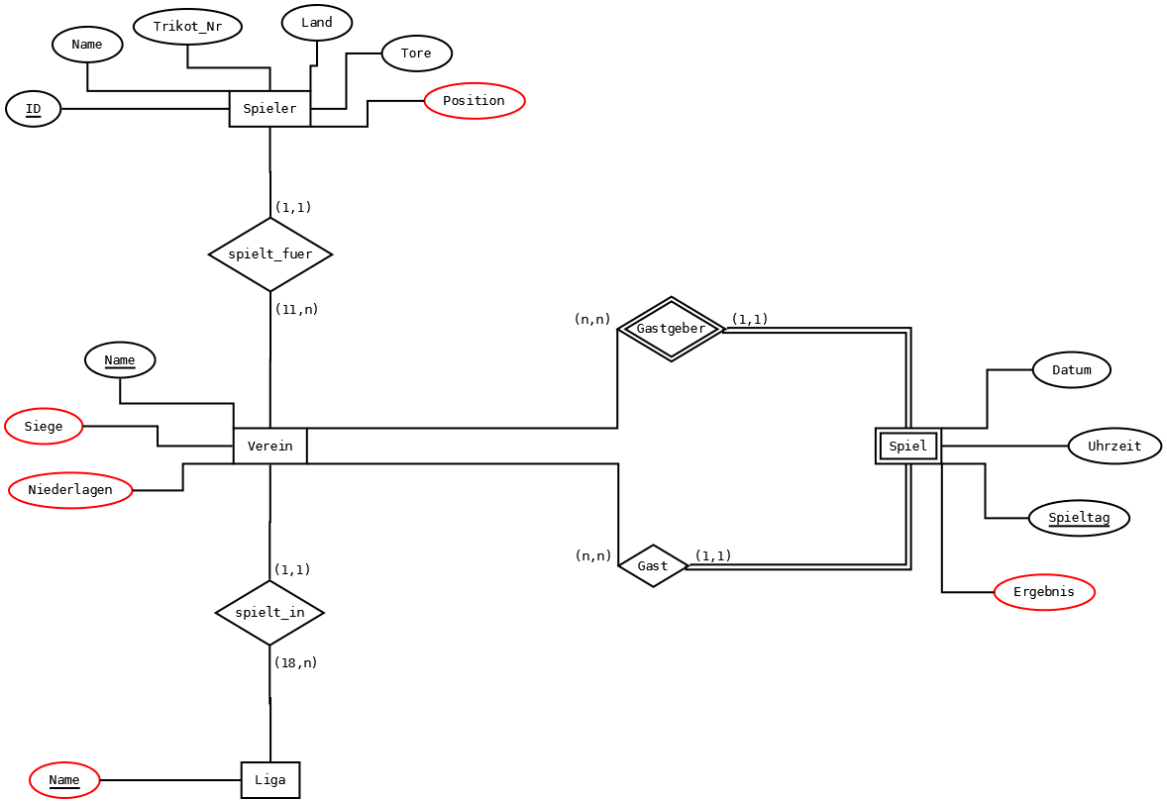
Ursprüngliches Entity-Relationship-Modell:



Nachdem die zu importierenden Daten bekannt wurden, haben wir nicht vorhandene und redundant gewordene Attribute gestrichen ...

Am Anfang war das Wort – (3/6)

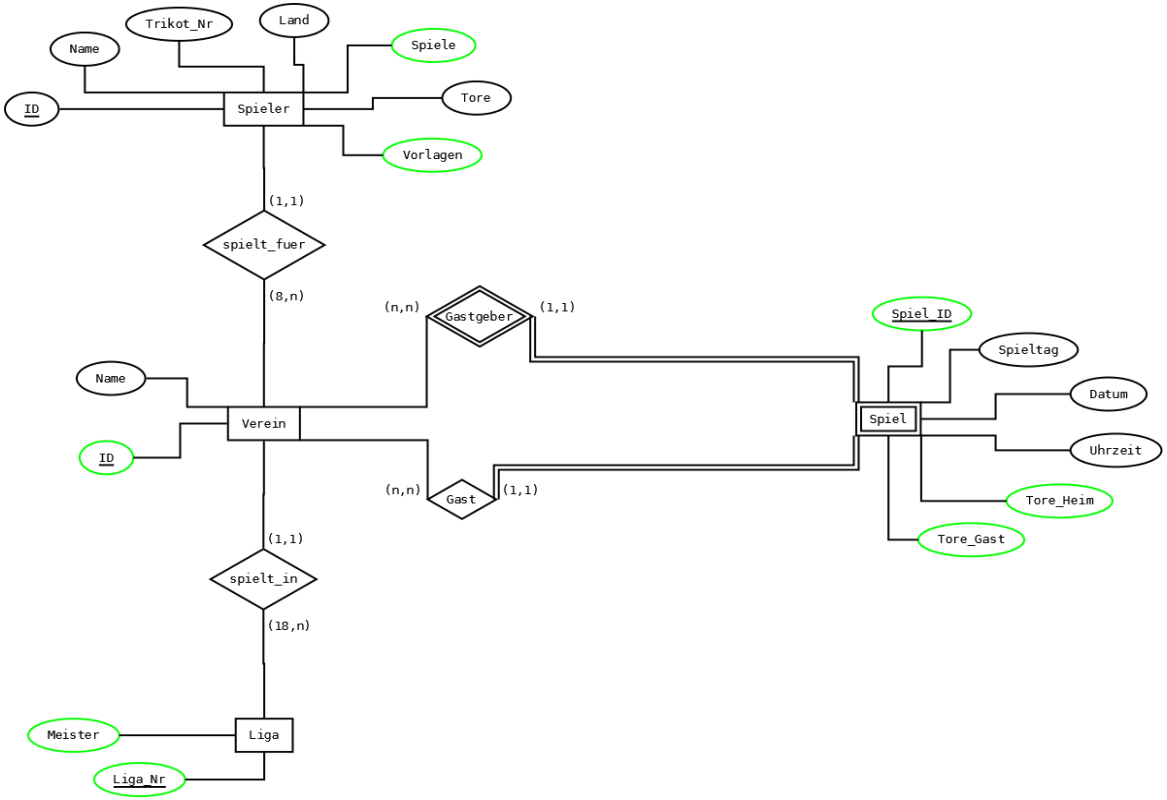
Streichungen im ursprünglichen Entity-Relationship-Modell:



und Attribute für die als sinnvoll erachteten zusätzlichen Informationen eingefügt.

Am Anfang war das Wort – (4/6)

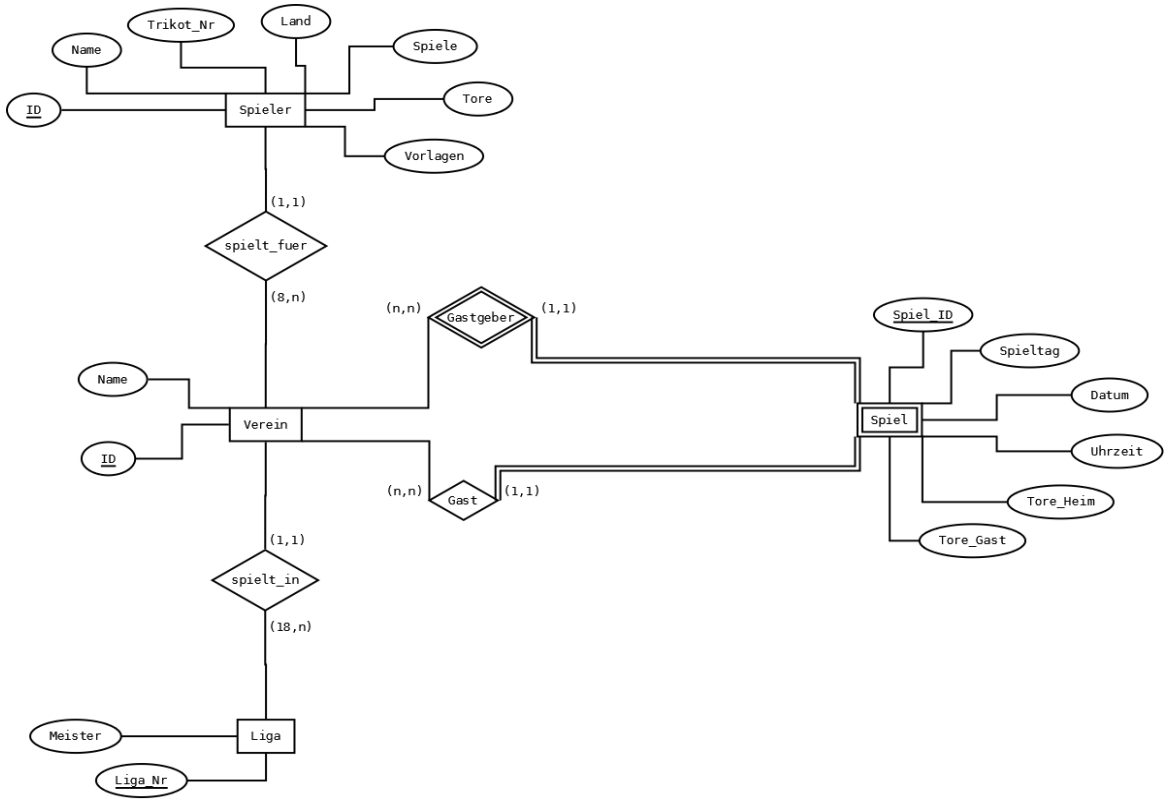
Hinzufügungen zum ursprünglichen Entity-Relationship-Modell:



Glücklicherweise konnten dabei alle Entitäten und Relationen der ursprünglichen Idee beibehalten werden. Nicht übernommen haben wir ...

Am Anfang war das Wort – (5/6)

Endgültiges Entity-Relationship-Modell:



die Informationen, die sich lediglich auf vergangene Saisons beziehen, da wir diese als nicht notwendig erachteten.

Am Anfang war das Wort – (6/6)

Durch die Änderungen war die Umsetzung des Spielergebnisses für uns geklärt worden und uns erwartete nur noch die Auswertung nach Sieg/Unentschieden/Niederlage für die Anfragen 5 und 6.

Spieler (Spieler_ID, S_Name, Trikot_Nr, Land, Spiele, Tore, Vorlagen, Vereins_ID)

Verein (V_ID, Name, Liga)

Spiele (Spiel_ID, Spieltag, Datum, Uhrzeit, Ergebnis, Heim, Gast, Tore_Heim, Tore_Gast)

Liga (Liga_Nr, Meister)

doppelte Unterstreichung → Einfach + gestrichelt

Von der Idee zum Code – (1/3)

Zunächst mussten die vorgegebenen Daten von der MySQL-Syntax zur PostgreSQL-Syntax portiert werden.

Dazu wurden Kommentare per Hand und abgrenzende Akzente (`) mittels Suchen/Ersetzen gelöscht.

MySQL:

```
--
-- Table structure for table `Liga`
--

DROP TABLE IF EXISTS `Liga`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `Liga` (
  `Liga_Nr` int(1) NOT NULL,
  `Verband` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
```

Von der Idee zum Code – (2/3)

PostgreSQL:

```
DROP TABLE IF EXISTS Liga;

CREATE TABLE Liga (
  Liga_Nr int(1) NOT NULL,
  Verband varchar(255) COLLATE utf8_unicode_ci NOT NULL,
```

Als nächstes stellten wir fest, dass keine Fremdschlüssel zu Tabellen eingefügt werden konnten, die noch nicht existierten. Darum haben wir diese Operationen an das Ende der Befehlskette gesetzt und wegen Konsistenz auch die Festlegung der Primärschlüssel.

Außerdem beschlossen wir auch das Erstellen der Tabellen sowie das spätere Einfügen der Werte jeweils aufeinander folgend zu bearbeiten.

Um diese Trennung dann auch im Code später wieder zu erkennen wurden die Befehle in drei Dateien aufgeteilt; je für Erstellen, Einfügen und Ändern der Datenbank.

Von der Idee zum Code – (3/3)

Die Umsetzung in JAVA bereitete keine besonderen Probleme, da sich Anleitungen für die Anbindung von Datenbanken in JAVA im Internet finden ließen.

Die größten Schwierigkeiten waren noch das Zusammenstellen der SELECT-Statements für die geforderten Anfragen und das Erstellen der Features für die spätere Data-Mining-Anwendung.

Getting Information from Information – (1/3)

Nach der Einarbeitung in Weka überlegten wir uns 3 verschiedene Klassifizierer zu verwenden, deren Eigenschaften bekannt und leicht zu verstehen sind.

Folgende benutzen wir: NaiveBayesSimple, J48, ZeroR

Ursprünglich erzeugte unser JAVA-Programm einzelne Dateien für die verschiedenen Features.

Resultierende Probleme: Mit den einzelnen Features lohnte es sich nicht einen Baum aufzustellen, deswegen wurde alles zusammen gelegt um den J48 Klassifikator sinnvoll zu machen.

Getting Information from Information – (2/3)

```
writer.write("@relation spiel\n\n");
writer.write("@attribute Verein numeric\n");
writer.write("@attribute Name string\n");
writer.write("@attribute Spieltag numeric\n");
writer.write("@attribute letzte3Tore numeric\n");
writer.write("@attribute letzte3GTore numeric\n");
writer.write("@attribute niederlagen5 numeric\n");
writer.write("@attribute siege5 numeric\n");
writer.write("@attribute draws5 numeric\n");
writer.write("@attribute real_outcome {Sieg, Unentschieden, Niederlage}\n\n");
```

Allerdings funktionieren sowohl der J48 und der NaiveBayesSimple Klassifikator nur wenn man ihn darauf trainiert einen Nominal-Wert zu generieren.

Also werden wir kein numerisches Ergebnis vorhersagen, sondern ob die Mannschaft gewinnt, verliert oder unentschieden spielt.

Getting Information from Information (3/3)

Zur besseren Lesbarkeit wäre es angenehm gewesen den Namen der Mannschaft einzuspeichern, leider ist dies aber eine String, mit dem man keinen Klassifikator benutzen kann.

Ausgabe der gesammelten Features:

```
56,'Wacker Burghausen',7,3,6,4,0,1,Unentschieden
5,'VfL Wolfsburg',18,8,3,0,2,3,Unentschieden
45,'Hallescher FC',15,2,6,2,3,0,Unentschieden
```

Ausgabe nach Data-Mining-Operationen:

```
56,7,3,6,4,0,1,Unentschieden,Unentschieden
5,18,8,3,0,2,3,Sieg,Unentschieden
45,15,2,6,2,3,0,Unentschieden,Unentschieden
```