OPITZ CONSULTING

Menschen. Innovationen. Lösungen.

Vom Umgang mit fremden Datenmodellen

Ein Leidensbericht

Uwe Küchler, Senior Consultant

OPITZ CONSULTING Deutschland GmbH

(NL Bad Homburg v.d.H.)

E-Mail: uwe.kuechler@opitz-consulting.com

Web: www.opitz-consulting.com

DOAG Regio Rhein-Neckar

DOAG-Regio Rhein-Neckar, 11.2.2014

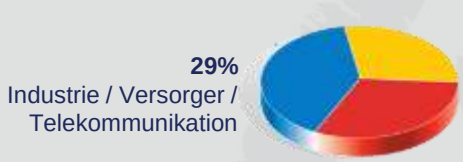
Mission

Wir entwickeln gemeinsam mit allen Branchen Lösungen, die dazu führen, dass sich diese Organisationen besser entwickeln als ihr Wettbewerb.

Unsere Dienstleistung erfolgt partnerschaftlich und ist auf eine langjährige Zusammenarbeit angelegt.

Märkte

- Branchenübergreifend
- Über 600 Kunden



Leistungsangebot

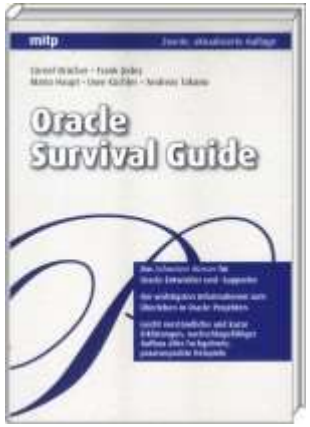
- Business IT Alignment
- Business Information Management
- Business Process Management
- Anwendungsentwicklung
- SOA und System-Integration
- IT-Infrastruktur-Management

Eckdaten

- Gründung 1990
- 400 Mitarbeiter
- 9 Standorte

Zur Person

- Generation C=64
- Seit über 25 Jahren in der IT tätig
- 1997-2000 bei Oracle Deutschland
- Seither durchgehend Oracle-Berater, im DBA- und Entwicklungs-Umfeld, Tutor
- Seit 09/2013 bei OPITZ CONSULTING
- Buch- und Blogautor (oraculix.de)
- Performance als „Steckenpferd“



Agenda

1. **Das perfekte, universelle Datenmodell!**
2. **Die perfekte Code-/Decode-Lösung**
3. **Praxisbeispiele**
4. **Fragen + Antworten**

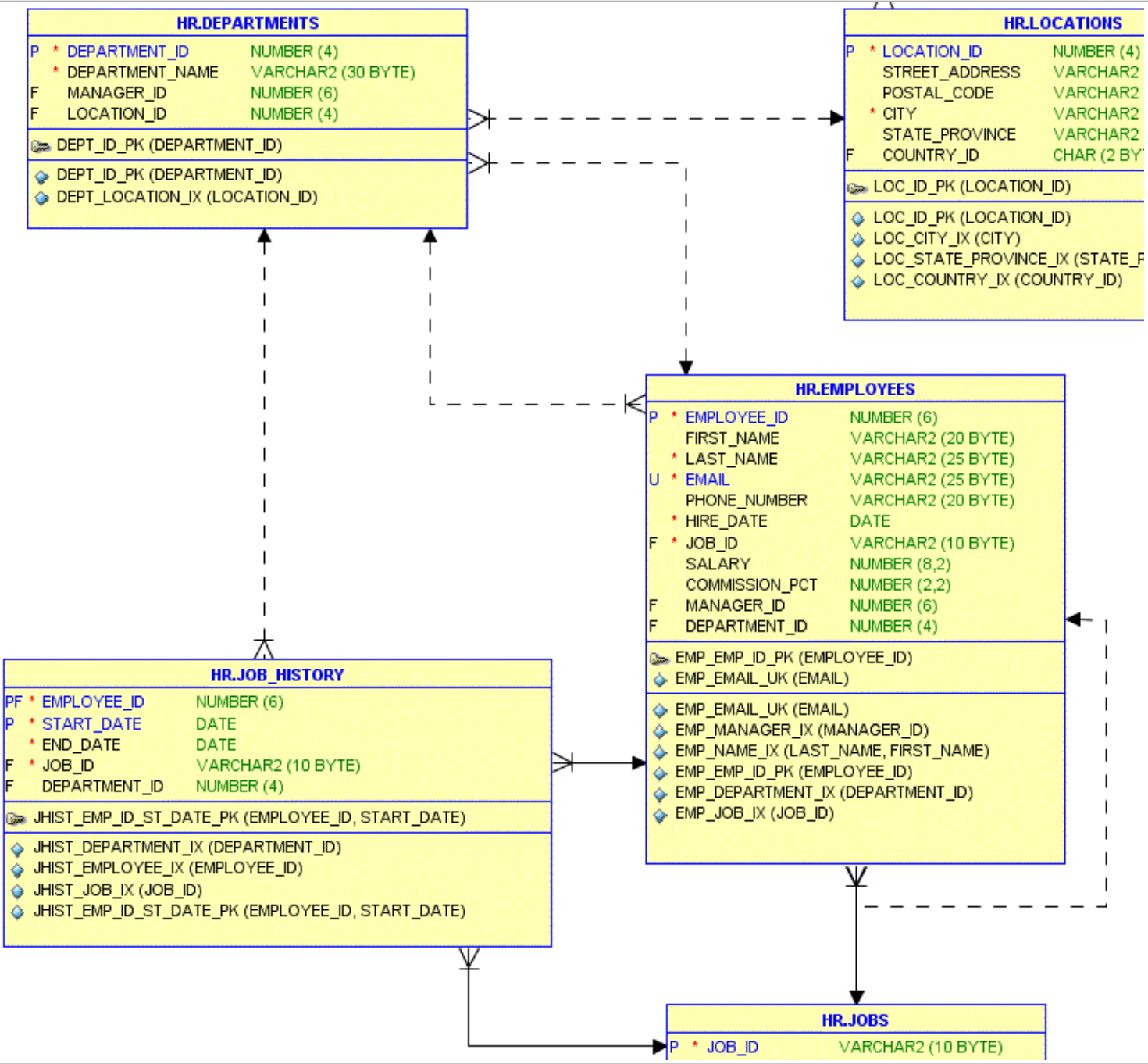
Motivation

- **Datenmodelle werden nur noch selten von Grund auf entwickelt.**
- **Meist trifft man – ob als Berater oder in einer neuen Position – auf existierende Systeme/Datenmodelle, mit denen man nun zunächst leben muss.**
- **Von der Praxiserfahrung einiger wiederkehrender Designfehler handelt dieser Vortrag.**
- **Ziel: Problemfelder erkennen, behandeln und idealerweise im Vorfeld vermeiden.**

1 Das perfekte, universelle Datenmodell!

Das perfekte, universelle Datenmodell

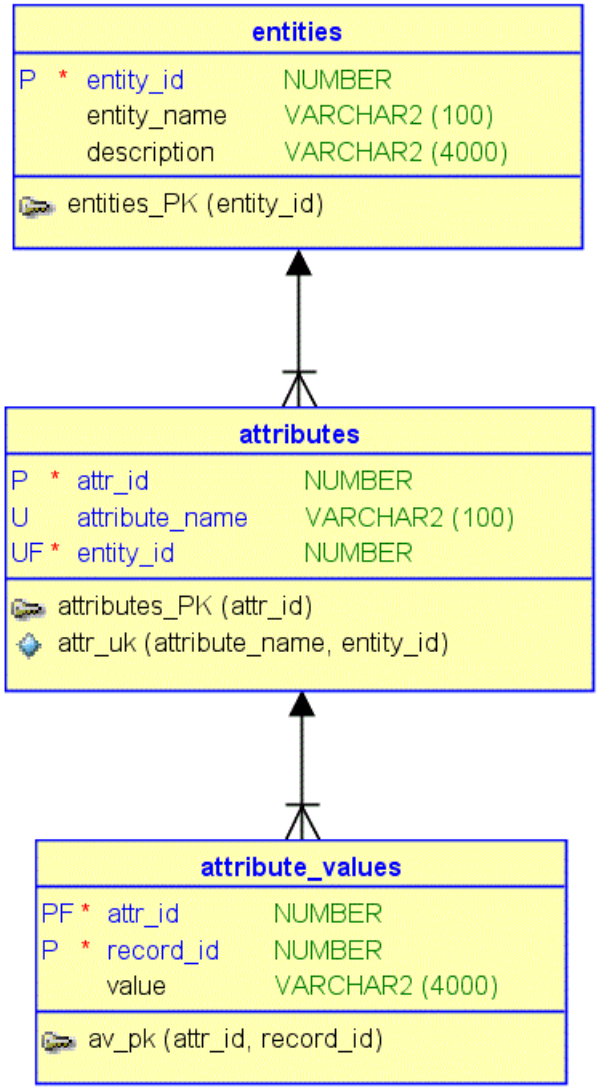
- **Vergleich eines schnöden, relationalen Datenmodells mit einer viel universelleren Lösung**
- **Ausgangsdesign: Oracles HR-Schema**


HR.DEPARTMENTS
P * DEPARTMENT_ID NUMBER (4)
* DEPARTMENT_NAME VARCHAR2 (30 BYTE)
F MANAGER_ID NUMBER (6)
F LOCATION_ID NUMBER (4)
DEPT_ID_PK (DEPARTMENT_ID)
DEPT_ID_PK (DEPARTMENT_ID)
DEPT_LOCATION_IX (LOCATION_ID)
HR.LOCATIONS
P * LOCATION_ID NUMBER (4)
STREET_ADDRESS VARCHAR2
POSTAL_CODE VARCHAR2
* CITY VARCHAR2
STATE_PROVINCE VARCHAR2
F COUNTRY_ID CHAR (2 BY
LOC_ID_PK (LOCATION_ID)
LOC_ID_PK (LOCATION_ID)
LOC_CITY_IX (CITY)
LOC_STATE_PROVINCE_IX (STATE_P
LOC_COUNTRY_IX (COUNTRY_ID)
HR.EMPLOYEES
P * EMPLOYEE_ID NUMBER (6)
FIRST_NAME VARCHAR2 (20 BYTE)
* LAST_NAME VARCHAR2 (25 BYTE)
U * EMAIL VARCHAR2 (25 BYTE)
PHONE_NUMBER VARCHAR2 (20 BYTE)
* HIRE_DATE DATE
F * JOB_ID VARCHAR2 (10 BYTE)
SALARY NUMBER (8,2)
COMMISSION_PCT NUMBER (2,2)
F MANAGER_ID NUMBER (6)
F DEPARTMENT_ID NUMBER (4)
EMP_EMP_ID_PK (EMPLOYEE_ID)
EMP_EMAIL_UK (EMAIL)
EMP_EMAIL_UK (EMAIL)
EMP_MANAGER_IX (MANAGER_ID)
EMP_NAME_IX (LAST_NAME, FIRST_NAME)
EMP_EMP_ID_PK (EMPLOYEE_ID)
EMP_DEPARTMENT_IX (DEPARTMENT_ID)
EMP_JOB_IX (JOB_ID)
HR.JOB_HISTORY
PF * EMPLOYEE_ID NUMBER (6)
P * START_DATE DATE
* END_DATE DATE
F * JOB_ID VARCHAR2 (10 BYTE)
F DEPARTMENT_ID NUMBER (4)
JHIST_EMP_ID_ST_DATE_PK (EMPLOYEE_ID, START_DATE)
JHIST_DEPARTMENT_IX (DEPARTMENT_ID)
JHIST_EMPLOYEE_IX (EMPLOYEE_ID)
JHIST_JOB_IX (JOB_ID)
JHIST_EMP_ID_ST_DATE_PK (EMPLOYEE_ID, START_DATE)
HR.JOBS
P * JOB_ID VARCHAR2 (10 BYTE)

Das perfekte, universelle Datenmodell: Nachteile des relationalen Modells

- **Man muss sich von vornherein darauf festlegen, welche Entitäten es gibt und wie sie ausmodelliert werden**
  - Wahl der richtigen Datentypen und Feldlängen
  - Beziehungen
  - Normalisierung / Denormalisierung
- **Spätere Erweiterungen und Veränderungen müssen per DDL gemacht werden**
  - DDL-Scripts sind oft einem aufwendigen Release-/Genehmigungsprozess unterworfen
  - „Das behindert agile Softwareentwicklung!“
- **Spätere Erweiterungen und Veränderungen erzwingen ein Umschreiben von Anwendungscode**
- **Das muss doch einfacher gehen!**

Ta Daaa!
Das Entity-Attribute-Value-Modell (EAV)!



Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Eine neue Abteilung im relationalen Modell anlegen:
INSERT INTO departments(department_id, department_name)
VALUES(100, 'DOAG Mannem')
/
-- Neue Angestellte einer Abteilung anlegen:
INSERT INTO employees( employee_id, first_name, last_name,
department_id)
VALUES( 1, 'Frank', 'Stöcker', 100)
/
INSERT INTO employees( employee_id, first_name, last_name,
department_id)
VALUES( 2, 'Scott', 'Tiger', 100)
/
COMMIT;
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Entitäten im EAV-Modell anlegen:
-- Zuerst:
INSERT INTO entities( entity_id, entity_name, description )
VALUES( 1000, 'DEPARTMENTS', 'Abteilungen')
/
INSERT INTO entities( entity_id, entity_name, description )
VALUES( 1001, 'EMPLOYEES', 'Angestellte')
/
-- Attribute für die Entitäten anlegen:
INSERT INTO attributes( attr_id, attribute_name, entity_id )
VALUES( 1, 'DEPARTMENT_ID', 1000 )
/
INSERT INTO attributes( attr_id, attribute_name, entity_id )
VALUES( 2, 'DEPARTMENT_NAME', 1000 )
/
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Attribute für die Entitäten anlegen (Fortsetzung):
INSERT INTO attributes( attr_id, attribute_name, entity_id )
VALUES( 3, 'FIRST_NAME', 1001 )
/
INSERT INTO attributes( attr_id, attribute_name, entity_id )
VALUES( 4, 'LAST_NAME', 1001 )
/
INSERT INTO attributes( attr_id, attribute_name, entity_id )
VALUES( 5, 'DEPARTMENT_ID', 1001 )
/
COMMIT;
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Eine neue Abteilung im EAV-Modell anlegen:
-- Zuerst die ID:
INSERT INTO attribute_values( attr_id, record_id, value )
VALUES( 1, 1, '100' )
/
-- Dann den Namen:
INSERT INTO attribute_values(attr_id, record_id, value)
VALUES( 2, 1, 'DOAG Mannem' )
/
-- Neue Angestellte einer Abteilung anlegen:
INSERT INTO attribute_values( attr_id, record_id, value)
VALUES( 3, 1, 'Frank' )
/
-- usw. usf. etc. pp.
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

Zwischenfazit:

- **Daten ins klassische Modell einzutragen**
  - Benötigt weniger Codezeilen
  - Ist schneller
  - Ist besser verständlich für Menschen
- **Dafür kann ich im EAV**
  - Ohne DDL jederzeit neue Entitäten anlegen
    - Das kann also jede Anwendung, ohne jeglichen Release-Aufwand!
  - Beinahe beliebige Daten einfügen, da ich keinen weiteren Constraints außer den 3 PKs und 2 FKs unterworfen bin. → hohe Reusability
  - Das DB-Design ist mit 3 „CREATE TABLE“ beendet! ☺

Wie bekomme ich die Daten nun wieder heraus?

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Einfache Abfrage: Alle Angestellten
-- (Also alle Werte zum Schlüssel “EMPLOYEES”)
SELECT av.record_id, attr.attribute_name, av.value
  FROM attributes attr, attribute_values av, entities en
 WHERE attr.entity_id = en.entity_id
   AND attr.attr_id = av.attr_id
   AND attr.entity_name = 'EMPLOYEES'
 ORDER BY av.record_id
/
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Eine immer noch einfache Abfrage:
-- Alle Angestellten mit Name der Abteilung
-- Im klassischen Modell:
SELECT dept.department_name, emp.first_name, emp.last_name
  FROM employees emp
     , departments dept
 WHERE emp.department_id = dept.department_id
 ORDER BY emp.last_name
/
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- Alle Angestellten mit Name der Abteilung, EAV-Style:
WITH dept AS
(
  SELECT did.value as department_id, dname.value as department_name
    FROM attribute_values did, attribute_values dname, attributes atr_did,
         attributes atr_name
   WHERE did.attr_id = atr_did.attr_id
     AND dname.attr_id = atr_name.attr_id
     AND did.record_id = dname.record_id
     AND atr_did.entity_id = atr_name.entity_id
     AND atr_did.entity_id = 1000 --'DEPARTMENTS'
     AND atr_did.attribute_name = 'DEPARTMENT_ID'
     AND atr_name.attribute_name = 'DEPARTMENT_NAME'
),
emp AS
(
  SELECT did.value as department_id,
         fname.value as first_name,
         lname.value as last_name
```

Das perfekte, universelle Datenmodell: Vergleich EAV ←→ „Classic“

```
-- ...
    FROM attribute_values did, attribute_values fname, attribute_values lname,
         attributes atr_did, attributes atr_fname, attributes atr_lname
   WHERE did.attr_id = atr_did.attr_id
     AND fname.attr_id = atr_fname.attr_id
     AND lname.attr_id = atr_lname.attr_id
     AND did.record_id = fname.record_id
     AND did.record_id = lname.record_id
     AND atr_did.entity_id = atr_fname.entity_id
     AND atr_did.entity_id = atr_lname.entity_id
     AND atr_did.entity_id = 1001 --'EMPLOYEES'
     AND atr_did.attribute_name = 'DEPARTMENT_ID'
     AND atr_fname.attribute_name = 'FIRST_NAME'
     AND atr_lname.attribute_name = 'LAST_NAME'
)
SELECT dept.department_name, emp.first_name, emp.last_name
  FROM dept, emp
 WHERE dept.department_id = emp.department_id
   AND dept.department_id = '100'
/
```

Das perfekte, universelle Datenmodell: Nachteile des EAV-Modells

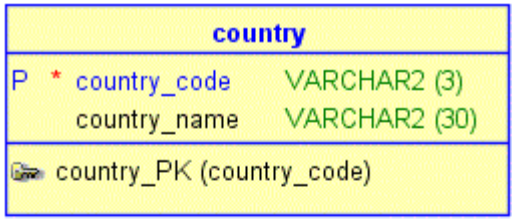
- **Prädikate erfordern pro Attribut einen zusätzlichen Join**
  - Wenn dazu noch LIKE oder NULL ins Spiel kommen → Full Scans, evt. Cartesian Joins → Adieu, Performance!
- **Wie code ich eine Anwendung gegen das Modell?**
  - IDs können sich ändern → Code ändern? Dynamisch generieren? Änderungen „einfach nicht vorgesehen“?
  - Anwendung muss Attributnamen kennen → kein Vorteil ggü. „Classic“
  - Aufgrund Attributnamen müssen Ids ermittelt werden – oder sie werden in die Anwendung hart codiert.
- **Wie formuliere ich Beziehungen?**
  - Und wie gewährleiste ich deren Konsistenz?
- **Wie stelle ich Beziehungen zu anderen Entitäten außerhalb meines EAV-Dreigestirns her?**
  - Siehe hierzu Praxisbeispiel weiter unten

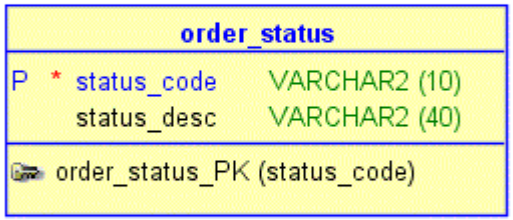
2

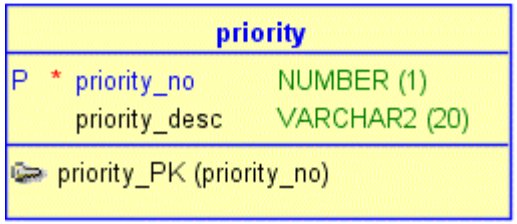
Die perfekte Code-/Decode-Lösung

Die perfekte Code-/Decode-Lösung

- **Warum viele einzelne Lookup-Tabellen…**

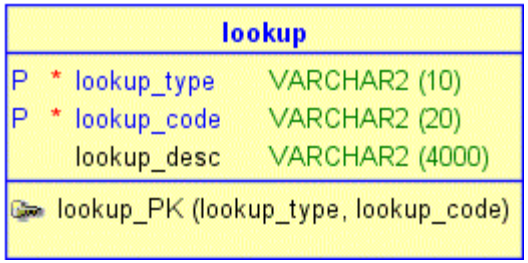






Die perfekte Code-/Decode-Lösung

- **…wenn es auch mit einer einzigen geht?**



Die perfekte Code-/Decode-Lösung
Vor- und Nachteile

- **Zuerst stellt sich die Frage, was die Tabelle(n) bezwecken sollen:**
  - Steuern von Eingabemasken (LOVs)
  - Prüfen der Datenintegrität (Constraints)
  - Lesbarkeit des Datenmodells
- **Danach kann man die Vor- und Nachteile des OTLT-Ansatzes beurteilen.**
- **Betrachtung hier aus DB-Sicht, nicht Anwendung**

Die perfekte Code-/Decode-Lösung
Vor- und Nachteile

- **Vorteile:**
  - Eine Tabelle statt vieler vereinfacht Anwendungscode (?) und Wartung
  - Eine Tabelle statt vieler ist übersichtlicher (?)
- **Nachteile:**
  - Nur noch ein Datentyp für Werte, keine Unterscheidung mehr
  - Auch Feldlängen sind nicht mehr spezifisch
  - Keine einfachen, Kategorie-spezifischen Constraints möglich
  - Wegen zusätzlicher Kategorie im PK
    - Entweder kein FK zur Tabelle mehr möglich
    - Oder FK muss aus Kategorie und Code bestehen
  - Erste Normalform verletzt
  - Performance sinkt mit wachsender Tabellengröße, besonders wenn Werte einer Kategorie nicht unmittelbar nacheinander plaziert sind.

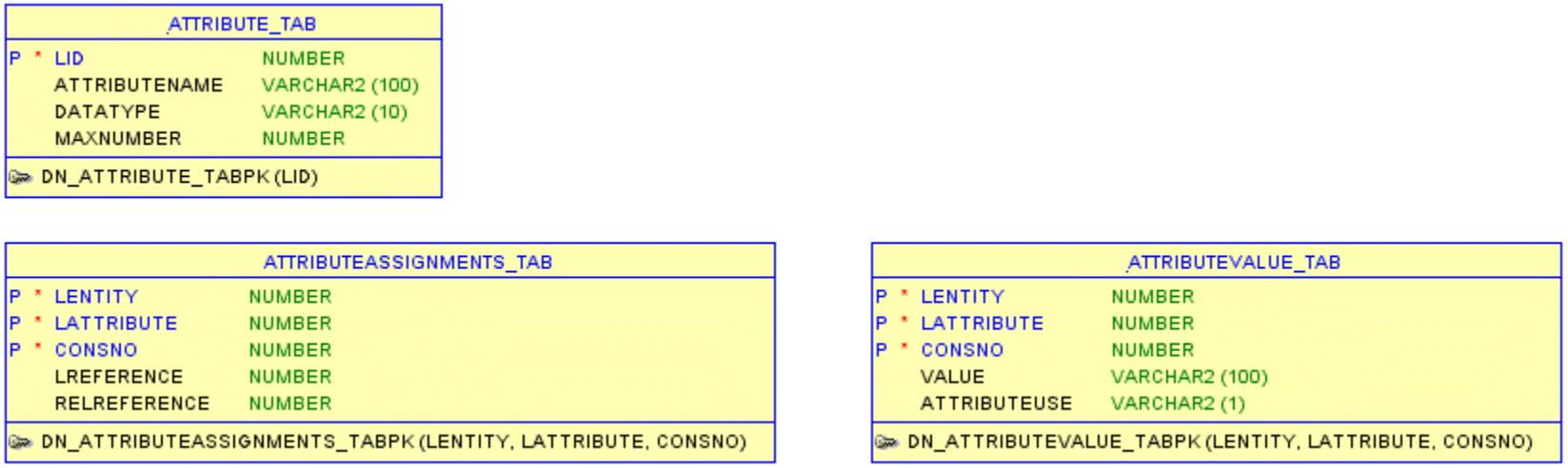
Alles schön und gut, aber das macht doch keiner in ernsthaften Anwendungen mit großen Datenmengen!

3 Praxisbeispiele

Praxisbeispiel: OLTP-System

- **Vorgefundene Implementierung bei folgenden Eckwerten:**
  - > 8000 Vermittler in
  - Mehreren 100, hierarchisch organisierten Filialen
  - > 10 Millionen Kunden
- **Probleme schon bei 20% Nutzlast**
  - Wichtigstes Problem: Logins dauern zu lange
- **Berichte über Monatsumsatz einer Filiale liefen über mehrere Stunden**

Praxisbeispiel: OLTP-System

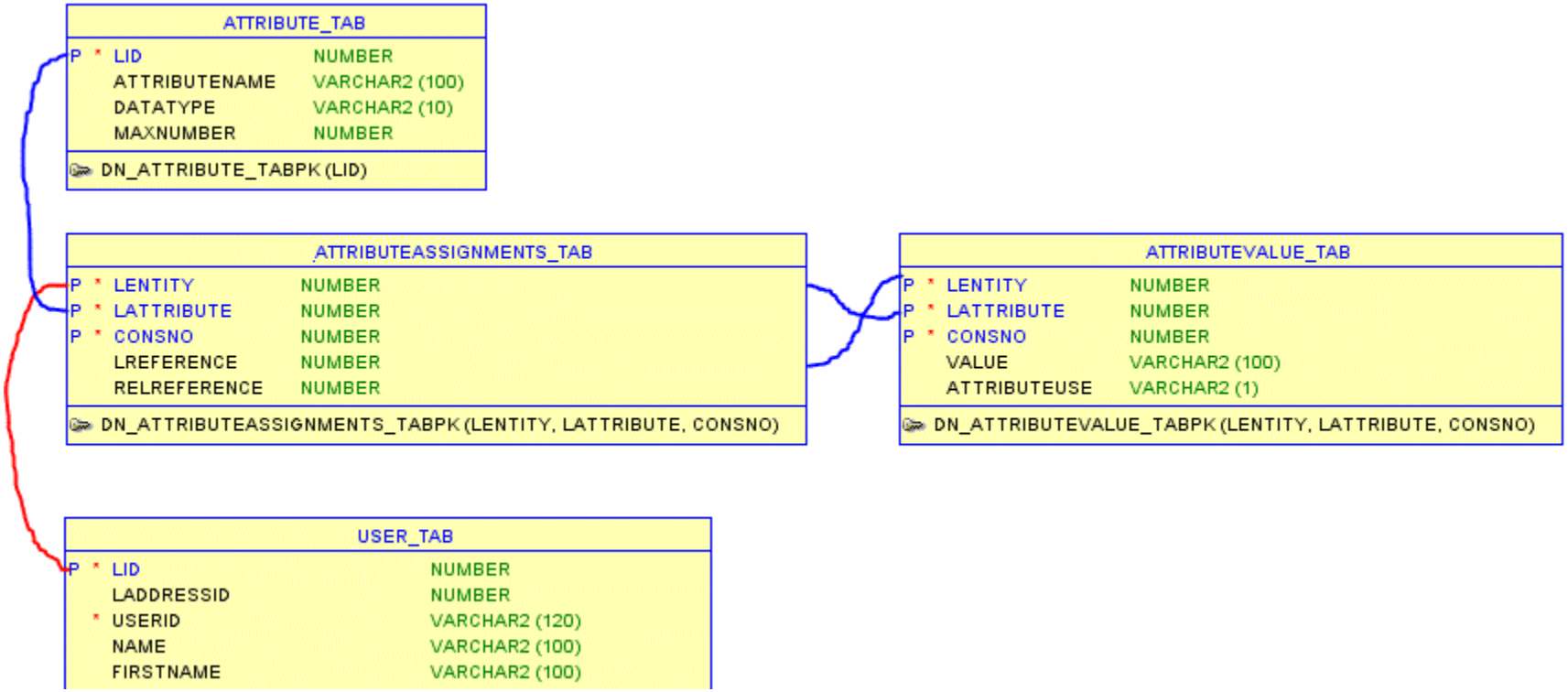


Praxisbeispiel: OLTP-System

Erste Auffälligkeiten:

- **Variante des EAV-Modells mit etwas mehr Metadaten**
  - Datentyp
  - Maximalzahl erlaubter Attribute
  - > 10 Millionen Kunden
- **Keine Foreign Keys definiert**
- **Inkonsistente Benamung der Spalten**
  - Zunächst unklar: lreference und relreference
- **Weitere Infos aus Views ersichtlich**
  - Es folgt die Ableitung der Fremdschlüssel aus dem View-Text:

Praxisbeispiel: OLTP-System

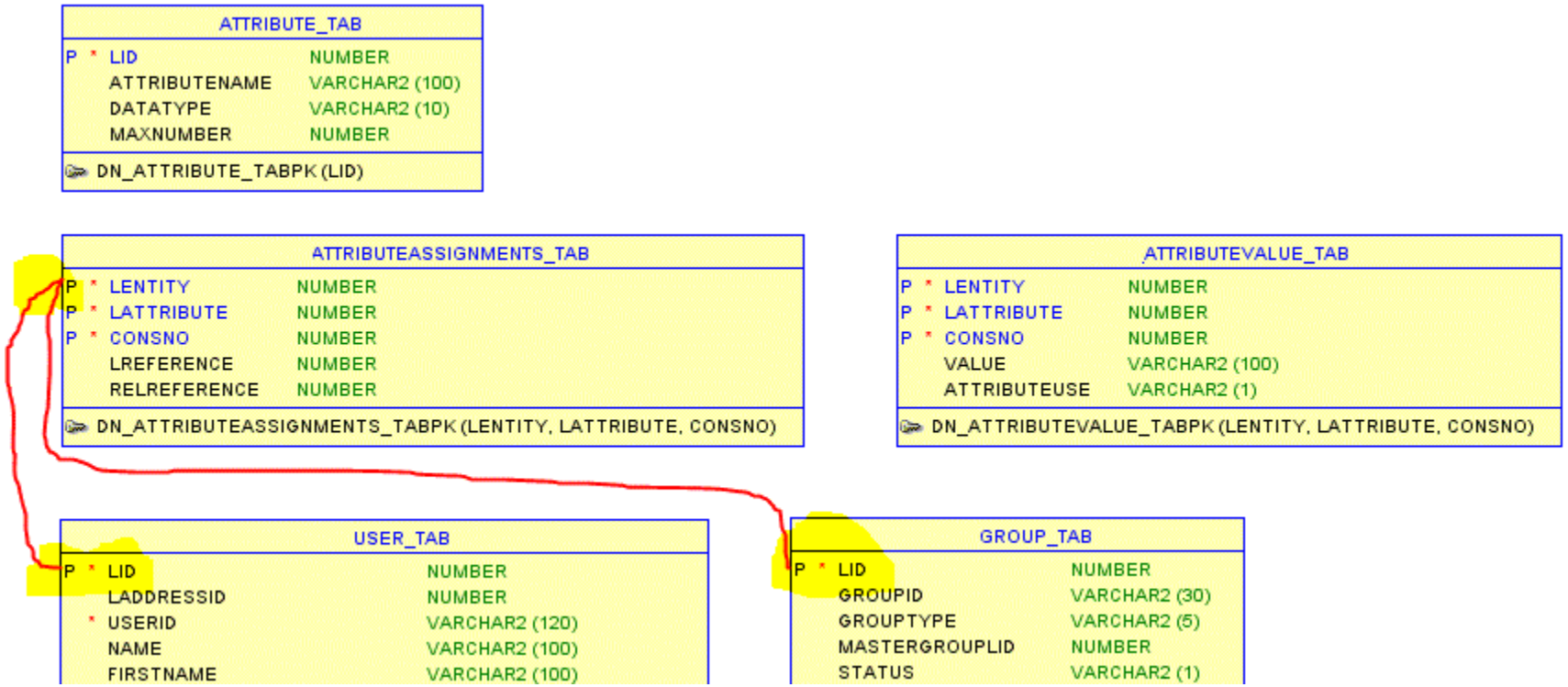


Praxisbeispiel: OLTP-System

Erste Auffälligkeiten:

- **Inkonsistente Benamung der Spalten**
  - lreference → lid
  - lid → lentity
- **Es kommt noch schlimmer:**

Praxisbeispiel: OLTP-System



Praxisbeispiel: OLTP-System

„It‘s in the mix“:

- **EAV-Modell wird mit klassischem Ansatz vermischt**
- **LID kann sowohl ein Benutzer als auch eine Gruppe sein!**
  - ➔ LID muss über zwei Tabellen hinweg eindeutig sein!
  - ➔ das kann nicht über ein Constraint geprüft werden!
    - Zwar über eine gemeinsame Sequence darstellbar, was aber, wenn SQL ohne Verwendung dieser Sequence ausgeführt wird?
  - Integrität auch hier nicht über FK abgesichert
- **Beispiel-View:**

Praxisbeispiel: OLTP-System

```
-- Benutzer-Attribute
-- Rollen, Rechte, Gruppen, Niederlassung, 2. Durchwahl, ...
CREATE OR REPLACE VIEW USERATTRIBUTES_VW AS
  SELECT a.lentity,
    a.lattribute,
    b.consno,
    [...]
FROM attributevalue_tab b,
     attributeassignments_tab a,
     attribute_tab c,
     user_tab d
  WHERE a.lreference = b.lentity
  AND a.lattribute   = b.lattribute
  AND a.lattribute   = c.lid
  AND d.lid          = a.lentity ;
/
```

Praxisbeispiel: OLTP-System

```
-- Benutzer-Attribute für Kunden
-- Joins für Zuordnung zu Benutzergruppen, zuständige Filiale und Berater
CREATE OR REPLACE VIEW appuser_view AS
  SELECT a.lid,
    [...]
  FROM user_tab a,
    group_tab b,
    attributevalue_tab c,
    attributevalue_tab d,
    attributeassignments_tab e,
    attributeassignments_tab f
  WHERE a.LMASTERGROUP = b.lid
  AND b.lid            = f.LENTITY
  AND f.LATTRIBUTE     = 123
  AND c.LENTITY        = f.LREFERENCE
  AND c.LATTRIBUTE     = 123
  AND b.lid            = e.LENTITY
  AND e.LATTRIBUTE     = 456
  AND d.LENTITY        = e.LREFERENCE
  AND d.LATTRIBUTE     = 456;
```

4 Fazit

Fazit

- **Projekterfahrung zeigt:**
  - EAV und OTLT kehren immer wieder
  - Dann oft schon seit Jahren implementiert
  - Schlimmerweise IDs in Anwendung hartcodiert
  - Referenzen darauf in Tausenden Codezeilen
- **EAV und OTLT vermeiden**
  - Ein RDBMS braucht klare Datentypen und Constraints!
  - EAV und OTLT skalieren schlecht
  - Sind sie erst einmal etabliert, wird man sie kaum mehr los
- **Migration zu „klassischem“ Modell**
  - Hart, dann möglicherweise hoher Fixing-Aufwand nach Release
  - Weich, z.B. über (Materialized) Views auf altes Modell und phasenweise Umstellung des Anwendungscodes auf das neue.

Literatur

- **EAV-Modell**
  - Kyte, Tom: Effective Oracle By Design (S. 34ff). Oracle Press 2003. ISBN 0-07-223065-7.
  - Auszug aus “Effective Oracle By Design” online: http://asktom.oracle.com/pls/asktom/f?p=100:11:0::::p11_question_id:10678084117056
  - Tony Andrews: http://tonyandrews.blogspot.de/2004/10/otlt-and-eav-two-big-design-mistakes.html
  - Anith Sen: https://www.simple-talk.com/sql/database-administration/five-simple--database-design-errors-you-should-avoid/
  - MUCK: http://rodgersnotes.wordpress.com/2010/09/21/muck-massively-unified-code-key-generic-three-table-data-model/
  - Mike Smithers: http://mikesmithers.wordpress.com/2013/12/22/the-anti-pattern-eavil-database-design/
  - C. J. Date: SQL and Relational Theory; O‘Reilly 2009, ISBN-13: 978-1449316402
- **One True Lookup Table (OTLT)**
  - Joe Celko: https://www.simple-talk.com/sql/t-sql-programming/look-up-tables-in-sql-/
  - Anith Sen: https://www.simple-talk.com/sql/database-administration/five-simple--database-design-errors-you-should-avoid/
  - Tony Andrews: http://tonyandrews.blogspot.de/2004/10/otlt-and-eav-two-big-design-mistakes.html
  - Stack Overflow: http://stackoverflow.com/questions/2691580/why-is-using-a-common-lookup-table-to-restrict-the-status-of-entity-wrong
  - Foreign Keys: http://docs.oracle.com/cd/E11882_01/appdev.112/e25518/adfns_constraints.htm#ADFNS273
- **Sonstige**
  - Serialisierte Objekte in der DB. http://asktom.oracle.com/pls/asktom/f?p=100:11:::::P11_QUESTION_ID:6692296628899