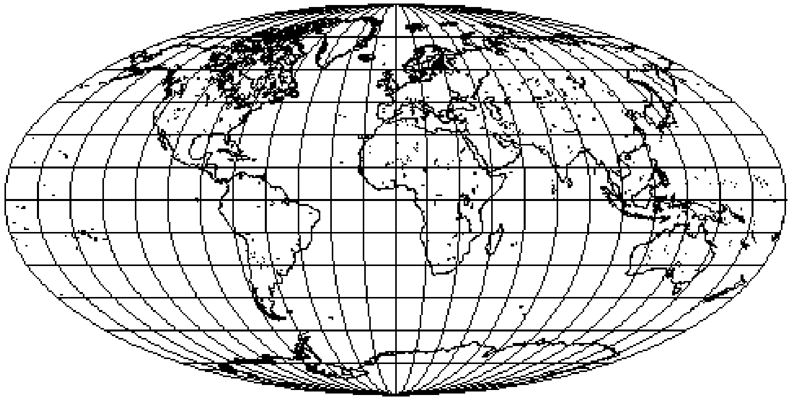
Database Programming in SQL/ORACLE



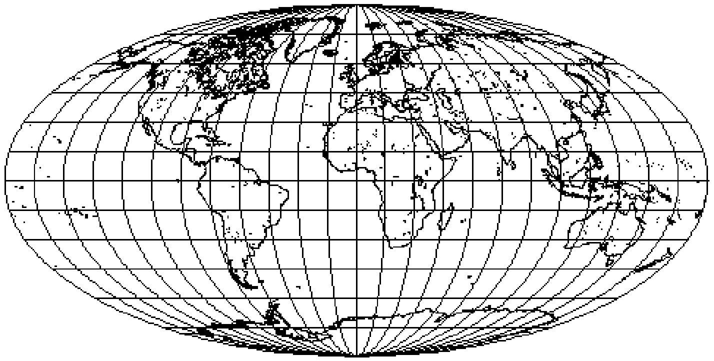
Wolfgang May

Termin: Wednesday, 14 Uhr ct. – $\leq$ 17.00 h,
Bldg. 51, Room 00-006.

SQL-3 Standard/ORACLE 8:

- ER-Modeling
- Schema Generation
- Queries
- Views
- Complex attributes, nested tables
- Database Optimization
- Access Control/Authorization
- Transactions
- Updates, Schema Modifications
- Referential Integrity
- PL/SQL: Triggers, Procedures, Functions
- Object-relational Features
- Embedded SQL
- JDBC (Embedding into Java)

The Database: MONDIAL



- Continents
- Countries
- Administrative Divisions
- Cities
- Organizations
- Mountains
- Rivers
- Lakes
- Seas
- Deserts
- Economy
- Population
- Languages
- Religions
- Ethnic Groups

- CIA World Factbook
- “Global Statistics”: Countries, Adm. Divisions, Cities
- TERRA-Database of the *Institut für Programmstrukturen und Datenorganisation der Universit”at Karlsruhe*
- . . . some more Web-Pages
- Data Integration has been done with *FLORID*

Organization of the SQL training

- Scripts for generating the database:

  /home/dbis/MondialDBP/
  Read: all users, Write: supervisors.

- Working Groups:

  `/home/dbis/db-prakt/gr1/`
  `.../home/dbis/db-prakt/gr`n`/`
  own SQL scripts.
  Read and Write for all group members.

- ORACLE-Database:

  Each participant has an own ORACLE account with a complete database.

  ⇒ Updates without problems

  Possible to grant read and write to group members.
  own database: `SELECT FROM CITY;`
  other database: `SELECT FROM <name>.CITY;`

Literature

- Textbooks on Databases (in german):

  A. Kemper, A. Eickler: Datenbanksysteme - Eine Einf”uhrung, Oldenbourg, 1996

  G. Vossen: Datenmodelle, Datenbanksprachen und Datenbankmanagement-Systeme. Addison-Wesley, 1994.

- Textbook on SQL (in german):
  G. Matthiessen and M. Unterstein: Relationale Datenbanken und SQL: Konzepte der Entwicklung und Anwendung. Addison-Wesley, 1997.

- The book on the practical DB training at Uni Karlsruhe with TERRA:
  M. Dürr and K. Radermacher: Einsatz von Datenbanksystemen. Springer Verlag, 1990.

- Explanation of the SQL-2 Standard:
  C. Date and H. Darwen: A guide to the SQL standard: a user's guide to the standard relational language SQL. Addison-Wesley, 1994.

- Textbooks on relational databases and SQL:

  H. F. Korth and A. Silberschatz: Database System Concepts. McGraw-Hill, 1991.

  J. Ullman and J. Widom: A First Course in Database Systems. Prentice Hall, 1997.

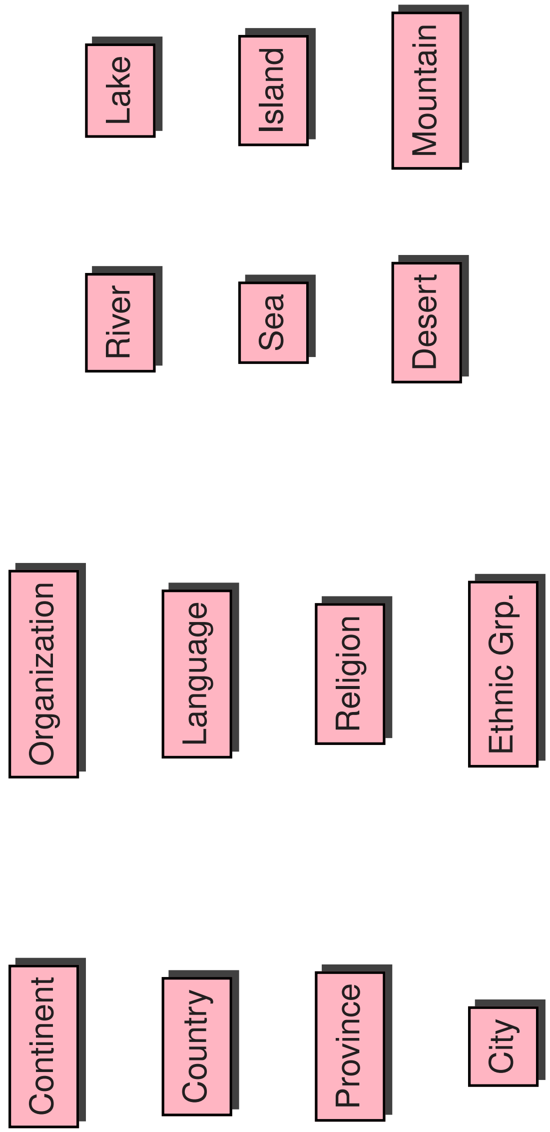
  and some more ...

Database Programming in SQL/ORACLE

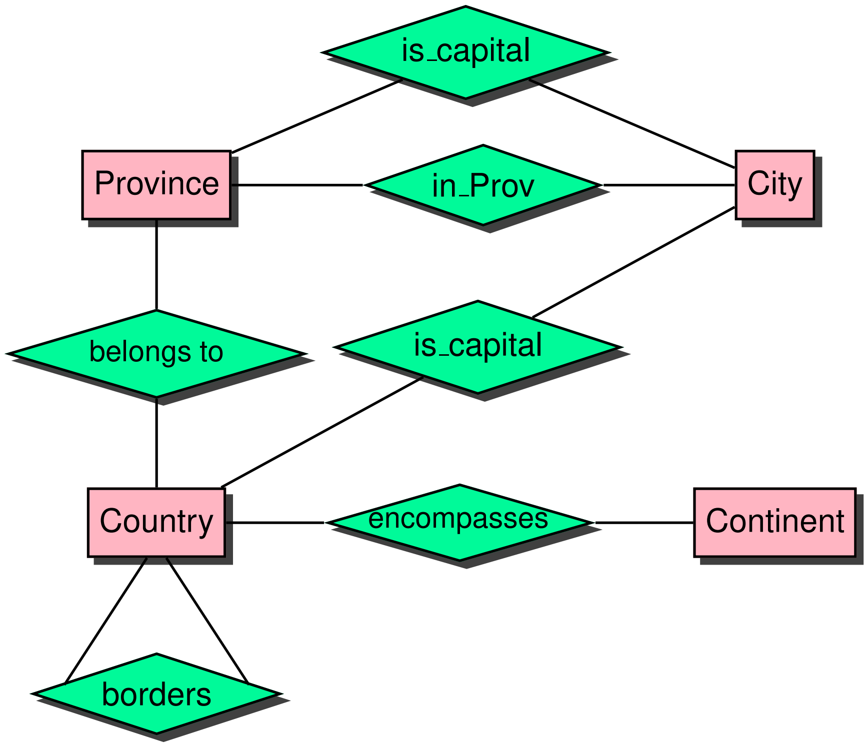
Semantic Modeling:
Entity Relationship Model (ERM; Chen, 1976)

Structuring concepts for describing a database schema in the ERM:

- Entity types ($\equiv$ Object types) and
- Relationship types



Entities and Relationships



Entities

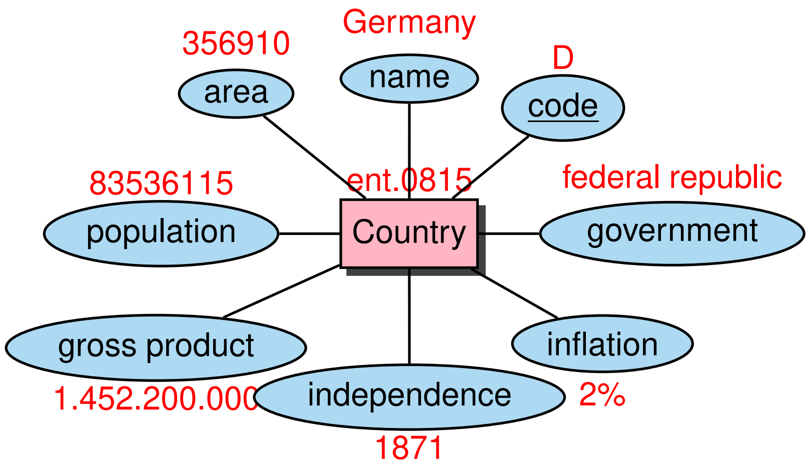
Entity type: An entity type represents a concept in the real world. It is given as a pair $(E, \{A_1, \ldots, A_n\})$, where E is the name and $\{A_1, \ldots, A_n\}$, $n \geq 0$ are the attributes (value properties) of a type.

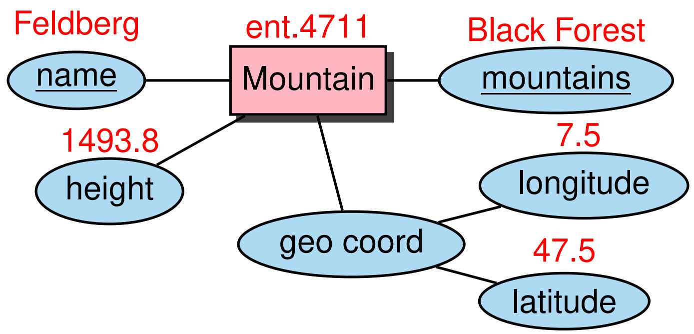
Attribute: a relevant property of entities of a given type. Each attribute can have *values* from a given *domain*.

Entity: each entity describes a real-world object. Thus, it must be of one of the defined entity types E. It assigns a value to each attribute that is declared for the entity type E.

Key attributes: a *key* is a set of attributes of an entity type, whose values together allow for a unique identification of all amongst all entities of a given type (cf. *candidate keys, primary keys*).

Entities:





Database Programming in SQL/ORACLE

Relationships

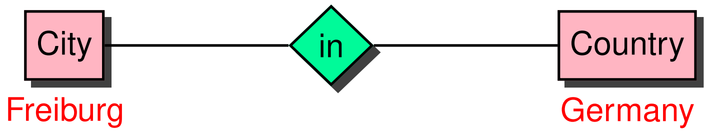
Relationship type: describes a concept of relationships between entities. It is given as a triple $(B, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_n\})$, where B is the name, $\{RO_1, \ldots, RO_k\},\ k \geq 2$, is a list of *roles*, $\{E_1, \ldots, E_k\}$ is a list of entity types associated to the roles, and $\{A_1, \ldots, A_n\}, n \geq 0$ is the set of attributes of the relationship type.

Roles are pairwise different – the associated entity types are not necessarily pairwise distinct. In case that $E_i = E_j$ for $i \neq j$, there is a **recursive** relationship.

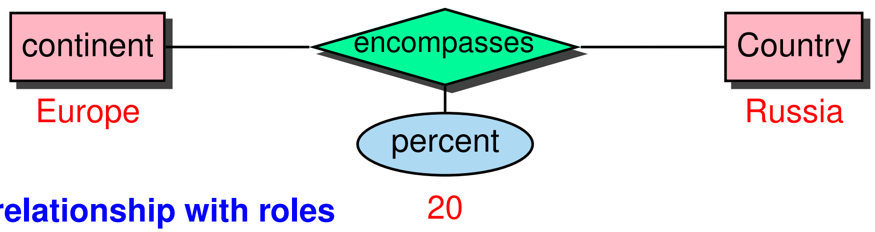
Attribute: relevant properties of relationships of a given type.

Relationship: A relationship of a relationship type B is defined by the entities that are involved in the relationship, according to their associated roles. For each role, there is exactly one entity involved in the relationship, and every attribute is assigned a value.

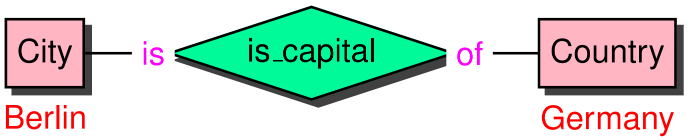
Relationships



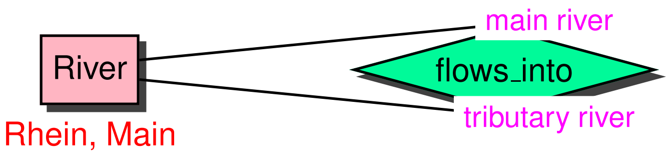
relationship with attributes



relationship with roles



recursive relationship



Database Programming in SQL/ORACLE

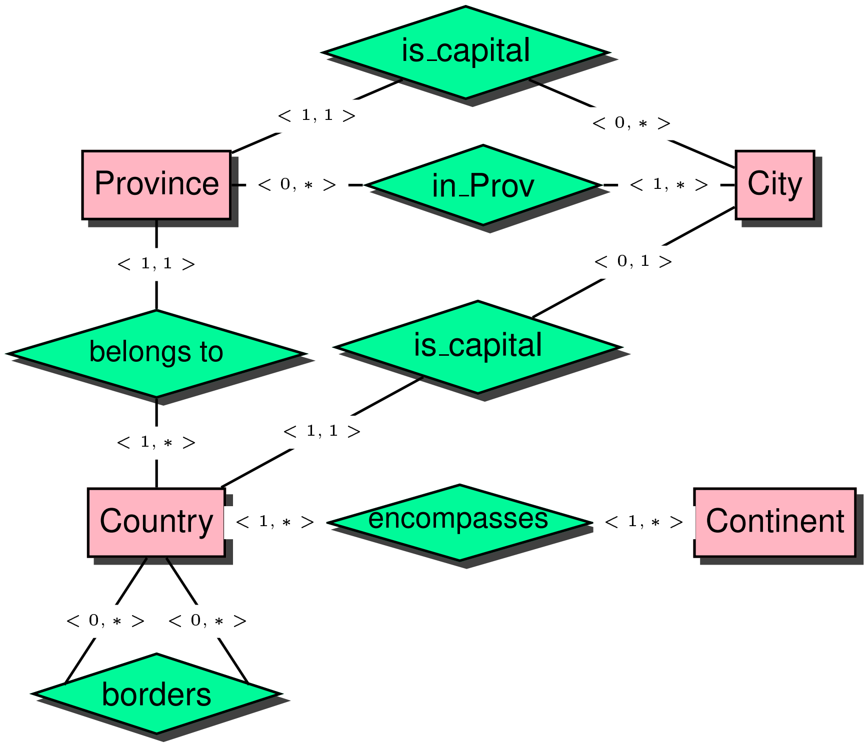
Complexities of relationships

Every relationship type is assigned a complexity that specifies the minimal and maximal number of relationships in which an entity of a given type may be involved.

The **complexity degree** of a relationship type B wrt. one of its roles RO is an expression of the form (min, max).

A set b of relationships satisfies the complexity degree (min, max) of a role RO if for all entities e of the corresponding entity type, the following holds: there exist at least min and at most max relationships b in which e is involved in the role RO.

Relationships



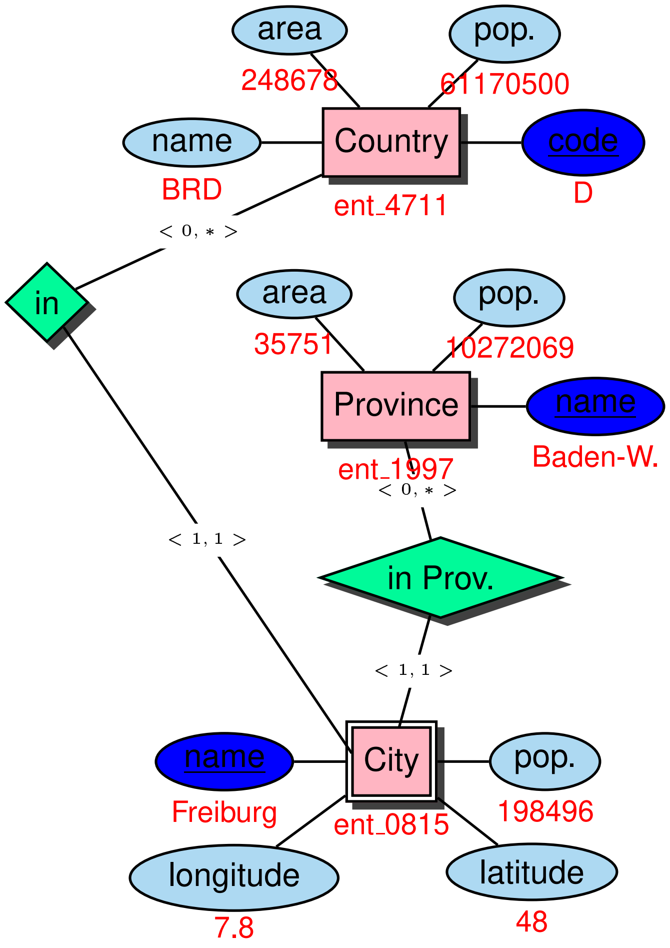
Weak Entity types

A weak entity type is an entity type without a key.

Thus their entities must be identified by the help of another entity.

- Weak entity types must be involved in at least one $n:1$-relationship with a strong entity type (where the strong entity type stands on the 1-side).
- They must have a **local** key, i.e., a set of attributes that can be extended by the primary keys of the corresponding strong entity type to provide a key for the weak entity type.

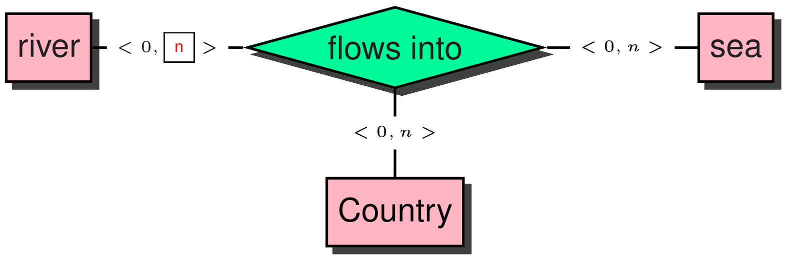
Weak Entity Types



There is also a Freiburg/CH
and Freiburg/Elbe, LowerSaxonia (Niedersachsen)

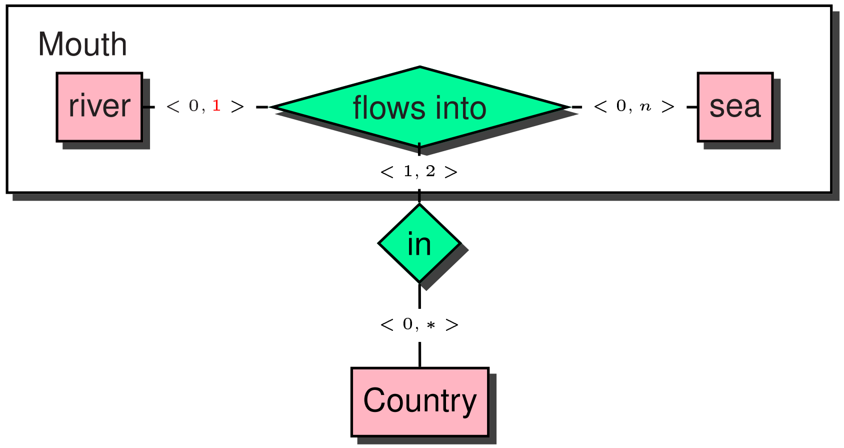
n-ary Relationships:

A river flows into a sea/lake/river; more detailed, this point can be described by giving one or two countries.



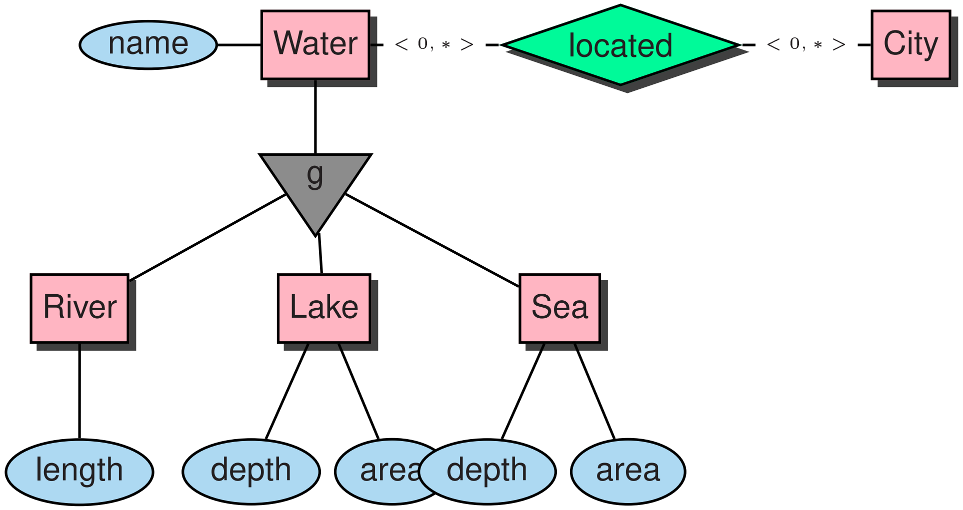
Aggregation:

Useful to introduce an *Aggregate type mouth*:



Generalization/Specialization

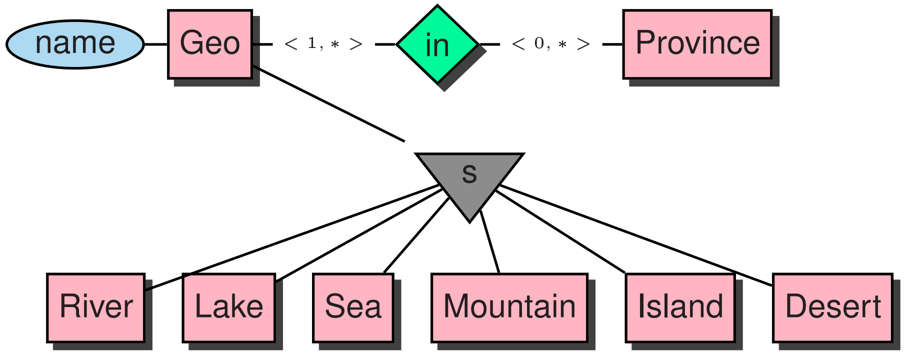
- Generalization: rivers, lakes, and seas are *waters*. These can e.g. be involved in *located-at* relationships with cities:



Database Programming in SQL/ORACLE

Generalization/Specialization

- Specialization: MONDIAL does not describe all geographical things, but only rivers, lakes, seas, mountains, deserts, and islands (no lowlands, highlands, savannas, fens, etc). All such geographical things have in common that they are involved in *in*-relationships with administrative divisions:



The Relational Model

- only a single structural concept *Relation* for entity types *and* relationship types,
- *Relational Model* by Codd (1970): mathematical foundation: set theory
- a relation schema consists of a name and a set of attributes,
  Continent: Name, Area
- each attribute is associated with a *Domain* which specifies the possible values of the attribute. Often, attributes also can have a *null value*.
  Continent: Name: VARCHAR(25), Area: NUMBER
- elements of a relation are called *tuples*.
  (Asia,4.5E7)

A **(relational) database schema** $\mathbf{R}$ is given by a (finite) set of (relation) schemata.
Continent: ...; Country: ...; City: ...

A **(database) state** associates each relation schema to a **relation**.

Mapping ERM to RM

Let E_{ER} an entity type and B_{ER} a relationship type in the ERM.

1. Entity types: $(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$,
2. Relationship types:
   $(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
   $B(E_1\_K_{11}, \ldots, E_1\_K_{1p_1}, \ldots,$
   $E_k\_K_{k1}, \ldots, E_k\_K_{kp_k},\ A_1, \ldots, A_m)$,
   where $\{K_{i1}, \ldots, K_{ip_i}\}$ are the primary keys of $E_i, 1 \leq i \leq k$.

   In case that for a relationship type B_{ER}, the keys of involved entity types have coinciding names, the role specifications may be used to guarantee the uniqueness of key attributes in the relationship type.

   In case that $k = 2$ and a (1,1) relationship complexity, the relation schema of the relationship type and that of the entity type may be merged.
3. For a weak entity type, the key attributes of the identifying entity type must be added.
4. Aggregate types can be ignored if the underlying relationship type is mapped.

Entity types

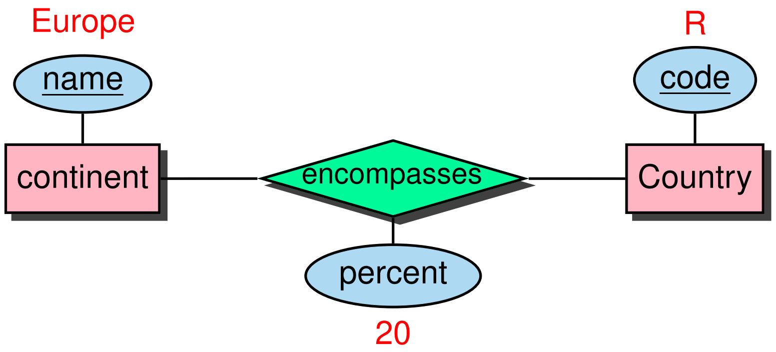
$$(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$$

Continent	
<u>**Name**</u>	**Area**
VARCHAR(20)	NUMBER
Europe	9562489.6
Africa	3.02547e+07
Asia	4.50953e+07
America	3.9872e+07
Australia	8503474.56

Relationship Types

$$(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$$
$$B(E_1\_K_{11}, \ldots, E_1\_K_{1p_1}, \ldots,$$
$$E_k\_K_{k1}, \ldots, E_k\_K_{kp_k},\ A_1, \ldots, A_m),$$

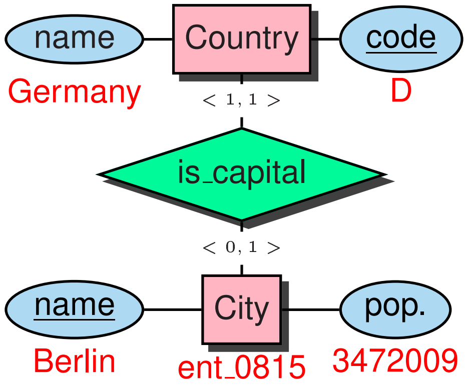
where $\{K_{i1}, \ldots, K_{ip_i}\}$ are the primary keys of $E_i, 1 \leq i \leq k$. (it is allowed to rename, e.g., to use *Country* for *Country.Code*)



encompasses		
<u>**Country**</u>	<u>**Continent**</u>	**Percent**
VARCHAR(4)	VARCHAR(20)	NUMBER
R	Europe	20
R	Asia	80
D	Europe	100
...	...	...

Relationship Types

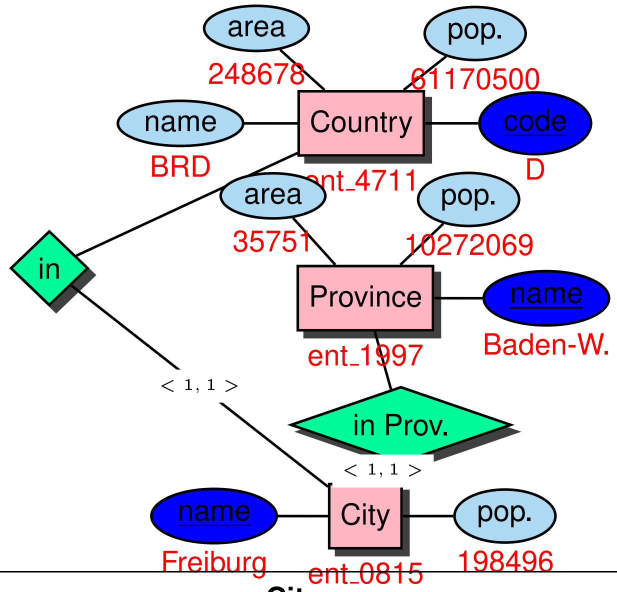
In case that $k = 2$ and a (1,1) relationship complexity, the relation schema of the relationship type and that of the entity type may be merged.



Country					
Name	**code**	**Population**	**Capital**	**Province**	**...**
Germany	D	83536115	Berlin	Berlin	
Sweden	S	8900954	Stockholm	Stockholm	
Canada	CDN	28820671	Ottawa	Quebec	
Poland	PL	38642565	Warsaw	Warszwaskie	
Bolivia	BOL	7165257	La Paz	Bolivia	
..	..	..	..	..	

Weak Entity Types

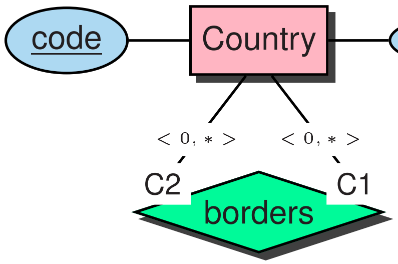
For a weak entity type, the key attributes of the identifying entity type must be added.



City				
Name	**Country**	**Province**	**Population**	**...**
Freiburg	D	Baden-W.	198496	..
Berlin	D	Berlin	3472009	..
..	..	..	..	..

Relationship Types

In case that for a relationship type B, the keys of involved entity types have coinciding names, the role specifications may be used to guarantee the uniqueness of key attributes in the relationship type.



borders	
Country1	**Country2**
D	F
D	CH
CH	F
..	..

Database Programming in SQL/ORACLE

SQL = Structured **Q**uery **L**anguage

- common query language
- standardization: SQL-89, SQL-2 (1992), SQL-3 (1996)
- SQL-2 in 3 stages: entry, intermediate, and full level
- SQL-3: object-orientation
- descriptive querying language
- results are always *sets of tuples (relations)*
- implementation: ORACLE (and many others)

- SQL is case-insensitive, i.e., CITY=city=City=cItY.
- inside quotes, SQL is not case-insensitive, i.e., City='Berlin' $\neq$ City='berlin'.
- every command has to be ended with a semicolon ";"
- comment lines are embraced in $/* \ldots */$, or introduced by `--` or `rem`.

Data Dictionary: Contains *meta data* about the database

Database Language:

DDL: **D**ata **D**efinition **L**anguage for defining schema

- tables
- views
- indexes
- integrity constraints

DML: **D**ata **M**anipulation **L**anguage for manipulating database states

- Search/Read
- Insert
- Modify
- Delete

Database Programming in SQL/ORACLE

Data Dictionary

Consists of tables and views that contain *meta data* about the database.

With `SELECT * FROM DICTIONARY` (abbrev. `SELECT * FROM DICT`),the Data Dictionary explains itself.

TABLE_NAME
COMMENTS
ALL_ARGUMENTS Arguments in objects accessible to the user
ALL_CATALOG All tables, views, synonyms, sequences accessible to the user
ALL_CLUSTERS Description of clusters accessible to the user
ALL_CLUSTER_HASH_EXPRESSIONS Hash functions for all accessible clusters
⋮

Data Dictionary

`ALL_OBJECTS`: contains all objects that are accessible for a user.

`ALL_CATALOG`: contains all tables, views, and synonyms that are accessible for a user.

`ALL_TABLES`: contains all tables that are accessible for a user.

Analogously for several other things. (`select * from ALL_CATALOG where TABLE_NAME LIKE 'ALL%';`).

`USER_OBJECTS`: contains all objects that where the user is the owner.

Analogously for other database object types, in most case there is also an abbreviation for `USER_...`, e.g. `OBJ` for `USER_OBJECTS`.

`ALL_USERS`: contains informations about all users of the database.


```
SELECT table_name FROM tabs;
```

Table_name	Table_name
BORDERS	ISLAND
CITY	LAKE
CONTINENT	LANGUAGE
COUNTRY	LOCATED
DESERT	IS_MEMBER
ECONOMY	MERGES_WITH
ENCOMPASSES	MOUNTAIN
ETHNIC_GROUP	ORGANIZATION
GEO_DESERT	POLITICS
GEO_ISLAND	POPULATION
GEO_LAKE	PROVINCE
GEO_MOUNTAIN	RELIGION
GEO_RIVER	RIVER
GEO_SEA	SEA

28 rows selected.

The schema of individual tables and views can be displayed by using DESCRIBE <table> or abbreviated DESC <table>:

```
DESC City;
```

Name	NULL?	Typ
NAME	NOT NULL	VARCHAR2(25)
COUNTRY	NOT NULL	VARCHAR2(4)
PROVINCE	NOT NULL	VARCHAR2(35)
POPULATION		NUMBER
LONGITUDE		NUMBER
LATITUDE		NUMBER

Queries: SELECT-FROM-WHERE

Queries against the database are in SQL formulated by the SELECT command. Its basic structure is simple:

SELECT	Attributes
FROM	Relation(s)
WHERE	Condition

Simplest form: all columns and rows of a relation

```
SELECT * FROM City;
```

Name	C.	Province	Pop.	Long.	Lat.
⋮	⋮	⋮	⋮	⋮	⋮
Vienna	A	Vienna	1583000	16,3667	48,25
Innsbruck	A	Tyrol	118000	11,22	47,17
Stuttgart	D	Baden-W.	588482	9.1	48.7
Freiburg	D	Germany	198496	NULL	NULL
⋮	⋮	⋮	⋮	⋮	⋮

3114 rows selected.

Projection: Choose some columns

```
SELECT <attr-list>
FROM <table>;
```

For all cities, give its name and the country to which it belongs:

```
SELECT Name, Country
FROM City;
```

Name	COUNTRY
Tokyo	J
Stockholm	S
Warsaw	PL
Cochabamba	BOL
Hamburg	D
Berlin	D
..	..

DISTINCT

```
SELECT * FROM Island;
```

Name	Islands	Area	...
⋮	⋮	⋮	⋮
Jersey	Channel Islands	NULL	...
Mull	Inner Hebrides	910	...
Montserrat	Antilles	106	...
Grenada	Antilles	NULL	...
⋮	⋮	⋮	⋮

```
SELECT Islands
FROM Island;
```

Islands
⋮
Channel Islands
Inner Hebrides
Antilles
Antilles
⋮

```
SELECT DISTINCT Islands
FROM Island;
```

Islands
⋮
Channel Islands
Inner Hebrides
Antilles
⋮

Duplicate Elimination

- Duplicates are not automatically eliminated:
  - duplicate elimination is expensive (sorting and deleting)
  - user may be interested in duplicates
  - later: aggregate functions on relations with duplicates
- Duplicate elimination: `DISTINCT`-clause
- later: Duplicates are automatically eliminated when set operations `UNION`, `INTERSECT`, ... are used

Database Programming in SQL/ORACLE

Selections: Choose some rows

```
SELECT <attr-list>
FROM <table>
WHERE <predicate>;
```

<predicate> may be of the following forms:

- <attribute> <op> <value> with $op \in \{=, <, >, <=, >=\}$,
- <attribute> `[NOT] LIKE` <string>, where each underscore in the string stands for an arbitrary character, and "%" stands for arbitrary many characters,
- <attribute> `IN` <value-list>, where <value-list> is either of the form ('val1',...,'valn'), or may be given as the result of a subquery,
- `[NOT] EXISTS <subquery>`
- `NOT` (<predicate>),
- <predicate> `AND` <predicate>,
- <predicate> `OR` <predicate>.

Example:

```
SELECT Name, Country, Population
FROM City
WHERE Country = 'J';
```

Name	Country	Population
Tokyo	J	7843000
Kyoto	J	1415000
Hiroshima	J	1099000
Yokohama	J	3256000
Sapporo	J	1748000
⋮	⋮	⋮

Example:

```
SELECT Name, Country, Population
FROM City
WHERE Country = 'J' AND Population > 2000000
```

Name	Country	Population
Tokyo	J	7843000
Yokohama	J	3256000

Example:

```
SELECT Name, Country, Population
FROM City
WHERE Country LIKE '%J_%';
```

Name	Country	Population
Kingston	JA	101000
Amman	JOR	777500
Suva	FJI	69481
⋮	⋮	⋮

The requirement that the “J” is followed by at least one character excludes japanese cities (“J”) from the result.

ORDER BY

```
SELECT Name, Country, Population
FROM City
WHERE Population > 5000000
ORDER BY Population DESC;     (descending)
```

Name	Country	Population
Seoul	ROK	10.229262
Mumbai	IND	9.925891
Karachi	PK	9.863000
Mexico	MEX	9.815795
Sao Paulo	BR	9.811776
Moscow	R	8.717000
⋮	⋮	⋮

ORDER BY, Alias

```
SELECT Name, Population/Area AS Density
FROM Country
ORDER BY 2 ; (Default: ascending)
```

Name	Density
Western Sahara	,836958647
Mongolia	1,59528243
French Guiana	1,6613956
Namibia	2,03199228
Mauritania	2,26646745
Australia	2,37559768

Aggregate functions

- `COUNT (*| [DISTINCT] <attribute>)`
- `MAX (<attribute>)`
- `MIN (<attribute>)`
- `SUM ([DISTINCT] <attribute>)`
- `AVG ([DISTINCT] <attribute>)`

Example: How many cities are stored in the database?

```
SELECT Count (*)
FROM City;
```

Count(*)
3114

Example: How many countries are stored in the database for which at least one city with more than 1,000,000 inhabitants is stored?

```
SELECT Count (DISTINCT Country)
FROM City
WHERE Population > 1000000;
```

Count(DISTINCT(Country))
68

Aggregate functions

Example: Compute the sum of the population of all Austrian cities, and the number of inhabitants of Austria's largest city.

```
SELECT SUM(Population), MAX(Population)
FROM City
WHERE Country = 'A';
```

SUM(Population)	MAX(Population)
2434525	1583000

And what, if these values are needed for *each* of the countries??

Grouping

`GROUP BY` conputes one row for every group. This group contains data that is obtained by using aggregate functions over all rows of the group.

```
SELECT <expr-list>
FROM <table>
WHERE <predicate>
GROUP BY <attr-list>;
```

returns for every value of `<attr-list>` *a single* row. Thus, in `<expr-list>` only the following expressions are allowed:

- constants,
- attribute from `<attr-list>`,
- attribute, which have the same value for all rows in such a group (e.g. `Code`, if `<attr-list>` contains *Country*),
- *Aggregate functions*, which are then applied to all tuples of the corresponding group.

The `WHERE` clause `<predicate>` contains only attributes of the relations mentioned in `<table>` (i.e., *no* aggregate functions).

Grouping

Example: For every country, return the number of inhabitants that live in cities.

```
SELECT Country, Sum(Population)
FROM City
GROUP BY Country;
```

Country	SUM(Population)
A	2434525
AFG	892000
AG	36000
AL	475000
AND	15600
⋮	⋮

Conditions over Groups

The HAVING clause allows to state additional conditions on the groups:

```
SELECT <expr-list>
FROM <table>
WHERE <predicate1>
GROUP BY <attr-list>
HAVING <predicate2>;
```

- WHERE clause: conditions on individual tuples *before* grouping,
- HAVING clause: conditions to select groups for the result. In the HAVING clause, in addition to aggregate function expressions over attributes, only those attributes are allowed that are mentioned *explicitly* in the GROUP BY clause.

Database Programming in SQL/ORACLE

Conditions on Groups

Example: Compute for each country the total number of inhabitants that live in cities with more than 100,000 inhabitants. Output only those countries where this number is more than 10 millions.

```
SELECT Country, SUM(Population)
FROM City
WHERE Population > 10000
GROUP BY Country
HAVING SUM(Population) > 10000000;
```

Country	SUM(Population)
AUS	12153500
BR	77092190
CDN	10791230
CO	18153631
⋮	⋮

Set Operations

SQL queries can be joined by set operations:

```
<select-clause> <set-op> <select-clause>;
```

- `UNION [ALL]`
- `MINUS [ALL]`
- `INTERSECT [ALL]`
- automatical elemination of duplicates (can be prevented by ALL)

Example: Give all names of cities that also occur as names of countries:

```
(SELECT Name
 FROM City)
INTERSECT
(SELECT Name
 FROM Country);
```

Name
Armenia
Djibouti
Guatemala
⋮

Join Queries

Join queries provide a possibility to combine several relations into a query.

```
SELECT <attr-list>
FROM <table-list>
WHERE <predicate>;
```

Basically, a join is based on the cartesian product of the contributing relations (Theory: see “Introduction to Databases”).

- resulting attributes: union of all attributes of contributing relations
- attributes that occur in several relations must be qualified by `<table>.<attr>`.
- join of a relation with itself – aliases.

Example: All countries that have less inhabitants than Tokyo.

```
SELECT Country.Name, Country.Population
FROM City, Country
WHERE City.Name = 'Tokyo'
AND Country.Population < City.Population;
```

Name	Einwohner
Albania	3249136
Andorra	72766
Liechtenstein	31122
Slovakia	5374362
Slovenia	1951443
⋮	⋮

Database Programming in SQL/ORACLE

Equijoin

Example: For all organizations, give the continents where their headquarter is located.

encompasses: Country, Continent, Percentage.

Organization: Abbreviation, Name, City, Country, Province.

```
SELECT Continent, Abbreviation
FROM encompasses, Organization
WHERE encompasses.Country = Organization.Country;
```

Name	Organization
America	UN
Europe	UNESCO
Europe	CCC
Europe	EU
America	CACM
Australia/Oceania	ANZUS
⋮	⋮

Join of a relation with itself

Example: Compute all pairs of cities in different countries which have the same name.

```
SELECT A.Name, A.Country, B.Country
FROM City A, City B
WHERE A.Name = B.Name
AND A.Country < B.Country;
```

A.Name	A.Country	B.Country
Alexandria	ET	RO
Alexandria	ET	USA
Alexandria	RO	USA
Barcelona	E	YV
Valencia	E	YV
Salamanca	E	MEX
⋮	⋮	⋮

Subqueries

The WHERE clause can contain results of subqueries:

```
SELECT <attr-list>
FROM <table>
WHERE <attribute> (<op> [ANY|ALL] | IN) <subquery>;
```

- <subquery> is a SELECT query (*Subquery*),
- for <op> $\in \{=, <, >, <=, >=\}$, <subquery> must result in a relation with a single column,
- for IN <subquery>, also multi-column results are allowed (since ORACLE 8),
- for <op> without ANY or ALL, the result of <subquery> must contain only a single row.

Uncorrelated Subquery

- independent from the values of the tuple which is currently processed in the surrounding query,
- evaluated *once* before the surrounding query,
- the result is then used for evaluating the `WHERE clause` of the surrounding query,
- strictly sequential evaluation, thus, the qualification of multiply occurring attributes is not necessary.

Example: Give all countries where there exists a city with name “Victoria”:

```
SELECT Name
FROM Country
WHERE Code IN
  (SELECT Country
   FROM City
   WHERE Name = 'Victoria');
```

Country.Name
Canada
Seychelles

Uncorrelated Subquery with `IN`

Example: Give all cities that are known to be situated at a river, lake, or a sea:

```
SELECT *
FROM CITY
WHERE (Name,Country,Province)
  IN (SELECT City,Country,Province
      FROM located);
```

Name	Country	Province	Population	...
Ajaccio	F	Corse	53500	...
Karlstad	S	Värmland	74669	...
San Diego	USA	California	1171121	...
⋮	⋮	⋮		

Subquery with ALL

Example: `ALL` can e.g. be used for computing all countries that are smaller than all countries that have more than 10 million inhabitants:

```
SELECT Name,Area,Population
FROM Country
WHERE Area < ALL
  (SELECT Area
   FROM Country
   WHERE Population > 10000000);
```

Name	Area	Population
Albania	28750	3249136
Macedonia	25333	2104035
Andorra	450	72766
⋮	⋮	⋮

Database Programming in SQL/ORACLE

Correlated Subquery

- Subquery depends on attribute values of the tuple which is currently processed in the outer query,
- evaluated *once for every tuple of the surrounding query*,
- *imported* attributes must be qualified.

Example: Compute all cities where more than 1/4 of the population of the corresponding country is living.

```
SELECT Name, Country
FROM City
WHERE Population * 4 >
  (SELECT Population
   FROM Country
   WHERE Code = City.Country);
```

Name	Country
Copenhagen	DK
Tallinn	EW
Vatican City	V
Reykjavik	IS
Auckland	NZ
⋮	⋮

The EXISTS Operator

`EXISTS` and `NOT EXISTS` simulate the existential quantifier.

```
SELECT <attr-list>
FROM <table>
WHERE [NOT] EXISTS
(<select-clause>);
```

Example: Compute all countries for which cities with more than 1,000,000 inhabitants are stored.

```
SELECT Name
FROM Country
WHERE EXISTS
     ( SELECT *
       FROM City
       WHERE Population > 1000000
       AND City.Country = Country.Code) ;
```

Name
Serbia and Montenegro
France
Spain
Austria
⋮

Database Programming in SQL/ORACLE

Transformation `EXISTS`, Subquery, Join

Equivalent to the previous one are the following queries:

```
SELECT Name
FROM Country
WHERE Code IN
     ( SELECT Country
       FROM City
       WHERE City.Population > 1000000);
```

```
SELECT DISTINCT Country.Name
FROM Country, City
WHERE City.Country = Country.Code
AND City.Population > 1000000;
```

Example

A country is strongly urbanized if more than 10 percent of its population live in cities with more than 500,000 inhabitants. Which member countries of the EU are strongly urbanized?

```
SELECT Country.Name
FROM Country, City, is_member
WHERE Organization = 'EU'
AND is_member.Country = Country.Code
AND is_member.Type = 'member'
AND City.Population > 500000
AND City.Country = Country.Code
GROUP BY Country.Name, Country.Population
HAVING (SUM(City.Population)/Country.Population) > 0.1;
```

Name
Austria
Denmark
Germany
Ireland
Italy
Netherlands
Spain
United Kingdom

Subqueries in the `FROM` Clause

```
SELECT <attr-list>
FROM <table/subquery-list>
WHERE <condition>;
```

Values which are obtained in different ways from different tables can be related.

Example: Compute the total number of people who do not live in the stored cities.

```
SELECT Population-Urban_Residents
FROM
  (SELECT SUM(Population) AS Population
   FROM Country),
  (SELECT SUM(Population) AS Urban_Residents
   FROM City);
```

Population-Urban_Residents
4620065771

Subqueries in the FROM Clause

... especially suitable for nested computations with aggregate functions

Example: Compute the total number of people who live in the largest city of their countries.

```
SELECT sum(pop_biggest)
FROM (SELECT country, max(population) as pop_biggest
   FROM City
   GROUP BY country);
```

sum(pop_biggest)
273837106

Database Programming in SQL/ORACLE

Schema Definition

- the database schema contains all information about the structure of the database,
- tables, views, constraints, indexes, clusters, triggers ...
- ORACLE 8: datatypes, methods
- is defined and modified using the DDL (Data Definition Language),
- CREATE, ALTER, and DROP of schema objects,
- access rights: GRANT.

Generation of Tables

```
CREATE TABLE <table>
  (<col> <datatype>,
     ⋮
   <col> <datatype>)
```

`CHAR(`n`)`: string with fixed length n.

`VARCHAR2(`n`)`: string with variable length $\leq n$.
$||$: string concatenation.

`NUMBER`: numbers. for `NUMBER`, the usual operators $+$, $-$, $*$, and $/$, and the comparisons $=$, $>$, $>=$, $<=$, and $<$ are allowed. Additionally there is `BETWEEN x AND y`. Inequality: $!=$, $\wedge=$, $\neg=$, or $<>$.

`DATE`: Dates and times: Century – Year – Month – Day – Hour – Minute – Second. There is also arithmetics and some more functions for these datatypes.

additional Datatypes are described in the manual.

Database Programming in SQL/ORACLE

Table Definition

The below SQL statement generates the *City* relation (still without integrity constraints):

```
CREATE TABLE City
      ( Name          VARCHAR2(35),
        Country       VARCHAR2(4),
        Province      VARCHAR2(32),
        Population    NUMBER,
        Longitude     NUMBER,
        Latitude      NUMBER );
```

Definition of Tables: Constraints

With the definition of tables, properties and constraints on the attribute values can be specified.

- Constraints on a single or on several attributes:
- Constraints on the domain,
- Specification of default values,
- NULL values allowed or not,
- Specification of key constraints,
- Predicates over each individual tuple.

Syntax:

```
CREATE TABLE <table>
  (<col> <datatype> [DEFAULT <value>]
        [<colConstraint> ... <colConstraint>],
  ⋮
  <col> <datatype> [DEFAULT <value>]
        [<colConstraint> ... <colConstraint>],
  [<tableConstraint>,]
  ⋮
  [<tableConstraint>])
```

- `<colConstraint>` concerns only a *single* column,
- `<tableConstraint>` can concern several columns.

Database Programming in SQL/ORACLE

Definition of Tables: Default Values

```
DEFAULT <value>
```

A member country of an organization is assumed to be a full member if nothing else is specified:

```
CREATE TABLE is_member
      ( Country       VARCHAR2(4),
        Organization  VARCHAR2(12),
        Type          VARCHAR2(30)
                      DEFAULT 'member')

INSERT INTO is_member VALUES
    ('CZ', 'EU', 'membership applicant');
INSERT INTO is_member (Land, Organization)
    VALUES ('D', 'EU');
```

Country	Organization	Type
CZ	EU	membership applicant
D	EU	member
⋮	⋮	⋮

Definition of Tables: Constraints

Two types of constraints:

- A column condition `<colConstraint>` is a condition that is concerned only with a *single* column (to which it is associated)
- A table condition `<tableConstraint>` may concern several columns.

Each `<colConstraint>` or `<tableConstraint>` is of the form

```
[CONSTRAINT <name>] <condition>
```


Definition of Tables: Conditions (Overview)

Syntax:

```
[CONSTRAINT <name>] <condition>
```

Keywords in `<condition>`:

1. `CHECK (<condition>)`: no line is allowed to violate `<condition>`. `NULL` values result in an *unknown* that does not violate any check condition.
2. `[NOT] NULL`: indicates whether a column is allowed to contain null values (only as `<colConstraint>`).
3. `UNIQUE (<column-list>)`: requires every value in a column to be unique (wrt. all tuples in this table).
4. `PRIMARY KEY (<column-list>)`: Declares the given columns as primary keys of this table.
5. `FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE]`: declares a set of attributes to be a foreign key.

Definition of Tables: Syntax

```
[CONSTRAINT <name>] <condition>
```

where `CONSTRAINT <name>` is optional (otherwise, an internal name is assigned).

- `<name>` is needed for `NULL`-, `UNIQUE`-, `CHECK`-, and `REFERENCES`-constraints, if the constraint should be changed or deleted eventually,
- `PRIMARY KEY` can be changed or deleted without having an explicit name.

Since for a `<colConstraint>`, the column is implicitly known, the `(<column-list>)` part is omitted.

Definition of Tables: CHECK Constraints

- as column constraints: domain constraint

```
CREATE TABLE City
 ( Name VARCHAR2(35),
   Population NUMBER CONSTRAINT CityPop
     CHECK (Population >= 0),
   ...);
```

- as table constraints: arbitrary integrity constraints on the values of each individual tuple.

Definition of Tables: PRIMARY KEY, UNIQUE, and NULL

- PRIMARY KEY (<column-list>): declares these columns to be the primary key of a table.
- PRIMARY KEY is equivalent to combining UNIQUE and NOT NULL.
- UNIQUE is *not* necessarily violated by NULL values, whereas PRIMARY KEY *forbids* NULL values.

One	Two
a	b
a	NULL
NULL	b
NULL	NULL

satisfies UNIQUE (One,Two).

- Since for each table, only one PRIMARY KEY may be defined, candidate keys must be specified by NOT NULL and UNIQUE.

Relation *Country*: Code is the PRIMARY KEY, Name is a candidate key:

```
CREATE TABLE Country
  ( Name          VARCHAR2(32) NOT NULL UNIQUE,
    Code          VARCHAR2(4) PRIMARY KEY);
```

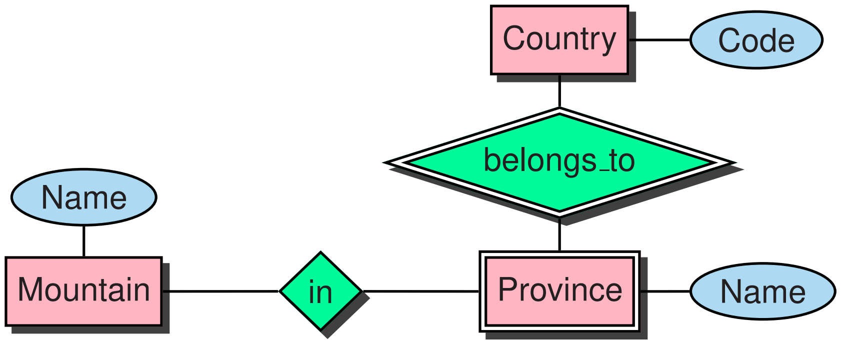

Definition of Tables: FOREIGN KEY ...REFERENCES

- FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE]: declares the attribute tuple <column-list> of the table to be a foreign key that references the attribute tuple <column-list2> of the table <table>.
- The referenced attribute tuple <table>(<column-list2>) must be declared as PRIMARY KEY of <table>.
- A REFERENCES condition is not violated by NULL values.
- ON DELETE CASCADE: referential action (later).

```
CREATE TABLE is_member
  (Country        VARCHAR2(4)
        REFERENCES Country(Code),
   Organization   VARCHAR2(12)
        REFERENCES Organization(Abbreviation),
   Type           VARCHAR2(30) DEFAULT 'member');
```

Definition of Tables: Foreign Keys

A mountain is located in a province of come country:




```
CREATE TABLE geo_Mountain
  ( Mountain VARCHAR2(20)
                  REFERENCES Mountain(Name),
    Country VARCHAR2(4) ,
    Province VARCHAR2(32) ,
    CONSTRAINT GMountRefsProv
                  FOREIGN KEY (Country,Province)
                  REFERENCES Province (Country,Name));
```


Definition of Tables

Complete definition of the table *City*, including conditions and keys:

```
CREATE TABLE City
 ( Name VARCHAR2(35),
   Country VARCHAR2(4)
     REFERENCES Country(Code),
   Province VARCHAR2(32)     -- + <tableConstraint>
   Population NUMBER CONSTRAINT CityPop
     CHECK (Population >= 0),
   Longitude NUMBER CONSTRAINT CityLong
     CHECK ((Longitude >= -180) AND (Longitude <= 180)),
   Latitude NUMBER CONSTRAINT CityLat
     CHECK ((Latitude >= -90) AND (Latitude <= 90)),
   CONSTRAINT CityKey
     PRIMARY KEY (Name, Country, Province),
   FOREIGN KEY (Country,Province)
     REFERENCES Province (Country,Name));
```

- if a table is generated with a `REFERENCES <table>(<column-list>)` clause, `<table>` must already be defined, and `<column-list>` must be declared as `PRIMARY KEY`.

Views

- Virtual tables
- are not computed at the time of their definition, but are
- computed each time when they are accessed.
- mirror the current state of the database.
- modifications (of the data) are restricted.

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
<select-clause>;
```

Example: A user ofte needs the information in which country some city is located, but is not interested in country codes and population:

```
CREATE VIEW CityCountry (City, Country) AS
  SELECT City.Name, Country.Name
  FROM City, Country
  WHERE City.Country = Country.Code;
```

If a user now searches for all cities in Cameroon, he can state the following query:

```
SELECT *
FROM CityCountry
WHERE Country = 'Cameroon';
```


Deleting Tables and Views

- tables and views are deleted with `DROP TABLE` or `DROP VIEW`:

  ```
  DROP TABLE <table-name> [CASCADE CONSTRAINTS];
  DROP VIEW <view-name>;
  ```
- tables need not to be empty when they are deleted.
- it is not possible to delete a table that contains referenced tuples.
- a table which is still a target of a `REFERENCES` declaration cannot be deleted by a simple `DROP TABLE` command.
- with `DROP TABLE <table> CASCADE CONSTRAINTS` a table is deleted together with all referential integrity constraints that point to it.

Modification of Tables and Views

later.

Inserting Information

- INSERT statement.
- insert individual tuples manually,

```
INSERT INTO <table>[(<column-list>)]
VALUES (<value-list>);
```

  or

- insert the result of a query:

```
INSERT INTO <table>[(<column-list>)]
<subquery>;
```

- remaining columns are filled with null values.

E.g., insert the subsequent tuple:

```
INSERT INTO Country (Name, Code, Population)
VALUES ('Lummerland', 'LU', 4);
```

A table *Metropolis (Name, Country, Population)* can be populated by the following statement:

```
INSERT INTO Metropolis
SELECT Name, Country, Population
FROM City
WHERE Population > 1000000;
```


Deletion of Tuples

Tuples can be deleted with the DELETE command:

```
DELETE FROM <table>
WHERE <predicate>;
```

With an empty WHERE clause, all tuples of a table are deleted (the table itself remains, it can be removed with DROP TABLE):

```
DELETE FROM City;
```

The below command deletes all cities that have less than 50,000 inhabitants:

```
DELETE FROM City
WHERE Population < 50000;
```

Modifying Tuples

```
UPDATE <table>
SET <attribute> = <value> | (<subquery>),
                          ⋮
    <attribute> = <value> | (<subquery>),
    (<attribute-list>) = (<subquery>),
                          ⋮
    (<attribute-list>) = (<subquery>)
WHERE <predicate>;
```

Example:

```
UPDATE City
SET Name = 'Leningrad',
    Population = Population + 1000,
WHERE Name = 'Sankt-Peterburg';
```

Beispiel: Set the total population of each country to the sum of the population of its administrative divisions:

```
UPDATE Country
SET Population = (SELECT SUM(Population)
                  FROM Province
                  WHERE Province.Country=Country.Code);
```


Date and Time

The `DATE` datatype stores century, year, month, day, hour, minute, second.

- Set input format by `NLS_DATE_FORMAT`,
- Default: `'DD-MON-YY'` e.g., `'20-Oct-97'`.

```
CREATE TABLE Politics
  ( Country VARCHAR2(4),
    Independence DATE,
    Government VARCHAR2(120));

ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';

INSERT INTO politics VALUES
  ('B','04 10 1830','constitutional monarchy');
```

All countries that have been founded between 1200 and 1600:

```
SELECT Country, Independence
FROM Politics
WHERE Independence BETWEEN
'01 01 1200' AND '31 12 1599';
```

Land	Datum
MC	01 01 1419
NL	01 01 1579
E	01 01 1492
THA	01 01 1238

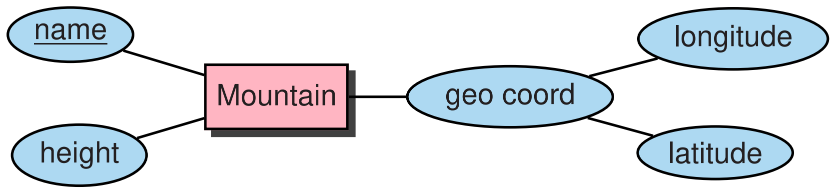
Date and Time

ORACLE provides some functions for working with DATE information:

- SYSDATE returns the current date/time.
- addition und subtraction of absolute values over DATE is allowed. Numbers are interpreted as days: SYSDATE $+ 1$ is tomorrow, SYSDATE $+ (10/1440)$ is “in ten minutes”.
- ADD_MONTHS(d, n) adds n months to a date d.
- LAST_DAY(d) yields the last day of a the month to which d belongs.
- MONTHS_BETWEEN(d_1, d_2) returns the number of months between two dates.

Object Orientation in ORACLE 8

- complex data types:



- nested tables:

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

- objects, methods, object tables, object references ... (later)

Generation of Data Types

New class of schema objects: `CREATE TYPE`

- ```
  CREATE [OR REPLACE] TYPE <name> AS OBJECT
    (<attr> <datatype>,
     ⋮
     <attr> <datatype>);
  ```
  For "full" objects, there is also a
  `CREATE TYPE BODY ...` where the methods are defined in PL/SQL ... later.

  Without body/methods, simply complex datatypes are generated (similar to *Records*).

- ```
  CREATE [OR REPLACE] TYPE <name>
     AS TABLE OF <datatype>
  ```
  ("Collection", tables as *data types*)

Complex Data Types

Geographical coordinates:

```
CREATE TYPE GeoCoord AS OBJECT
  ( Longitude NUMBER,
    Latitude  NUMBER);
/

CREATE TABLE Mountain
      ( Name          VARCHAR2(20),
        Height        NUMBER,
        Coordinates   GeoCoord);
```

`CREATE TYPE <type> AS OBJECT (...)` automatically defines a *Constructor method* `<type>`:

```
INSERT INTO Mountain
  VALUES ('Feldberg', 1493, GeoCoord(8,48));
```

```
SELECT * FROM Mountain;
```

Name	**Height**	**Coordinates(Longitude, Latitude)**
Feldberg	1493	GeoCoord(8,48)

Complex Data Types

Access to individual components of complex attributes uses the common *dot*-Notation (similar to records).
ORACLE 8.0: *only with qualification*:

```
SELECT Name, B.Coordinates.Longitude,
       B.Coordinates.Latitude
FROM Mountain B;
```

Name	Coordinates.Longitude	Coordinates.Latitude
Feldberg	8	48

Nested Tables

```
CREATE [OR REPLACE] TYPE <inner_type>
  AS OBJECT (...);
/
CREATE [OR REPLACE] TYPE <inner_table_type> AS
  TABLE OF <inner_type>;
/
CREATE TABLE <table_name>
  (...  ,
   <table-attr> <inner_table_type> ,
   ...  )
NESTED TABLE <table-attr> STORE AS <name >;
```

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Languages_nested;
```

Nested Tables

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Languages_nested;
```

Again: constructor methods

```
INSERT INTO NLanguage
VALUES( 'SK',
        Languages_list
          ( Language_T('Slovak',95),
            Language_T('Hungarian',5)));
```


Nested Tables

```
SELECT *
FROM NLanguage
WHERE Country='CH';
```

Country	Languages(Name, Percentage)
CH	Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

```
SELECT Languages
FROM NLanguage
WHERE Country='CH';
```

Languages(Name, Percentage)
Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

Querying Contents of Nested Tables

Contents of inner tables:

```
THE (SELECT <table-attr> FROM ...)
```

```
SELECT ...
FROM THE (<select-statement>)
WHERE ... ;
```

```
INSERT INTO THE (<select-statement>)
VALUES ... / SELECT ... ;
```

```
DELETE FROM THE (<select-statement>)
WHERE ... ;
```

```
SELECT Name, Percentage
FROM THE( SELECT Languages
          FROM NLanguage
          WHERE Country='CH');
```

Name	Percentage
German	65
French	18
Italian	12
Romansch	1

Copying Nested Tables

Nested tables can be inserted “as a whole” if the set of tuples is structured (casted) as a collection:

```
CAST(MULTISET(SELECT ...) AS <nested-table-type>)
```

```
INSERT INTO NLanguage -- allowed, but wrong !!!!
 (SELECT Country,
         CAST(MULTISET(SELECT Name, Percentage
                       FROM Language
                       WHERE Country = A.Country)
              AS Languages_List)
 FROM Language A);
```

each tuple (country, languageList) n-times
(n = number of languages in this country) !!

```
INSERT INTO NLanguage (Country)
 (SELECT Country
  FROM Language);
```

```
UPDATE NLanguage B
SET Languages =
  CAST(MULTISET(SELECT Name, Percentage
                FROM Language A
                WHERE B.Country = A.Country)
       AS Languages_List);
```

Nested Tables

If a query already results in a table, this can be inserted as a whole:

```
INSERT INTO <table>
VALUES (...,  THE ( SELECT <attr>
                    FROM <table'>
                    WHERE ...)    );
```

```
INSERT INTO NLanguage VALUES
 ('CHXX', THE (SELECT Languages from NLanguage
                 WHERE Country='CH'));
```


Working with Nested Tables

Not too simple ... (ORACLE 8.0)

- Subquery may only return a *single* nested table. ⇒ not possible to select an inner table, depending on the surrounding tuple:

All countries where german is spoken:

```
SELECT Country -- NOT ALLOWED !!!!
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country)
WHERE Name='German');
```

Working with Nested Tables

```
TABLE ([<table>.]<attr>)
```

can be used in *Subquery*:

```
SELECT Country
FROM NLanguage
WHERE EXISTS
  (SELECT *
   FROM TABLE (Languages) -- to the current tuple
   WHERE Name='German');
```

Country
A
B
CH
D
NAM

But: Attributes of the inner table cannot be selected in the outer SELECT statement.

⇒ not possible to return the percentage of the languages in the corresponding countries.

Working with Nested Tables

CURSOR-Operator:

Example:

```
SELECT Country,
       CURSOR (SELECT *
               FROM TABLE (Languages))
FROM NLanguage;
```

Country	**CURSOR(SELECT...)**
CH	CURSOR STATEMENT : 2
NAME	**PERCENTAGE**
French	18
German	65
Italian	12
Romansch	1

⇒ Cursors etc. in PL/SQL.

Working with Nested Tables

```
SELECT Country, Name -- NOT ALLOWED !!
FROM NLanguage A,
  THE ( SELECT Languages
        FROM NLanguage B
        WHERE B.Country=A.Country);

SELECT Country, Name
FROM NLanguage A,
  THE ( SELECT Languages
        FROM NLanguage B
        WHERE B.Country=A.Country)
WHERE A.Country = 'CH'; -- now allowed.
```

Using a table *All_Languages* that contains all languages:

```
SELECT Country, Name
FROM NLanguage, All_Languages
WHERE Name IN
  (SELECT Name
   FROM TABLE (Languages));
```

Conclusion: the domain of nested tables must be accessible in a *single* table.

Complex Data Types

`SELECT * FROM USER_TYPES`

Type_name	Type_oid	Typecode	Attributes	Methods	Pre	Inc
GeoCoord	_	Object	2	0	NO	NO
Language_T	_	Object	2	0	NO	NO
Languages_List	_	Collection	0	0	NO	NO

Delete: `DROP TYPE [FORCE]`

With `FORCE`, a datatype can be deleted whose definition is still needed by other types.

Same scenario:

`DROP TYPE Language_T`

“Typ mit abh”angigen Typen oder tables kann nicht gel”oscht oder ersetzt werden”

`DROP TYPE Language_T FORCE` deletes `Language_T`, but

```
SQL> desc Languages_List;
FEHLER:
ORA-24372:  Ung"ultiges Objekt f"ur Beschreibung
```

Transactions in ORACLE

Begin of a Transaction

```
SET TRANSACTION READ [ONLY | WRITE];
```

Safepoints

For a long transaction, savepoints can be set:

```
SAVEPOINT <savepoint>;
```

End of a Transaction

- `COMMIT` statement: all changes become persistent,
- `ROLLBACK [TO <savepoint>]` undoes all changes [since `<savepoint>`],
- DDL statement (e.g. `CREATE, DROP, RENAME, ALTER`),
- User exits from ORACLE,
- process is killed.

Referential Integrity – A First Look

- if a table that contains columns that are defined as foreign keys by `REFERENCES <table>(<column-list>)` is generated, `<table>` must be already defined, and `<column-list>` must already be declared as `PRIMARY KEY`.
- When tuples are inserted, the corresponding referenced tuples must already be present.
- When tuples are deleted, the referential integrity must be preserved.
- tables and views are deleted with `DROP TABLE` or `DROP VIEW`.
- it is not possible to delete a table that still contains referenced tuples.
- tables which are targets of a `REFERENCES` declaration can be deleted by `DROP TABLE <table> CASCADE CONSTRAINTS`.
- nested tables do not support referential integrity.

TEIL II: This and That

Part I: Basics

- ER model and relational data model
- generation of a (relational) schema: `CREATE TABLE`
- queries: `SELECT -- FROM -- WHERE`
- working on the database: `DELETE, UPDATE`

Part II: further topics on basic SQL

- modifications of the database schema
- referential integrity
- view updates
- access control
- optimization

Part III: prodecural concepts, OO, embedding

- PL/SQL: procedures, functions, triggers
- object-orientation
- Embedded SQL, JDBC

Database Programming in SQL/ORACLE

Modification of Schema Objects

- `CREATE` statement
- `ALTER` statement
- `DROP` statement

- `TABLE`
- `VIEW`
- `TYPE`
- `INDEX`
- `ROLE`
- `PROCEDURE`
- `TRIGGER`

⋮

Modification of Table Schemata

- ALTER TABLE
- add columns and conditions,
- change conditions,
- delete, deactivate, and reactivate conditions.

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ⋮
  DROP <drop-clause>
  DISABLE <disable-clause>
  ⋮
  DISABLE <disable-clause>
  ENABLE <enable-clause>
  ⋮
  ENABLE <enable-clause>;
```


Database Programming in SQL/ORACLE

Adding Columns to Tables

```
ALTER TABLE <table>
  ADD (<col> <datatype> [DEFAULT <value>]
            [<colConstraint> ... <colConstraint>],
       ⋮
       <col> <datatype> [DEFAULT <value>]
            [<colConstraint> ... <colConstraint>],
       <add table constraints>...)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ...  ;
```

New columns are filled with NULL values.

Beispiel: The relation *economy* is extended with a column *unemployment*:

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER CHECK (Unemployment > 0));
```

Adding Table Conditions

```
ALTER TABLE <table>
  ADD (<... add some columns ... >,
       <tableConstraint>,
       ⋮
       <tableConstraint>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ... ;
```

Add an assertion that the sum of the percentages of industry, service and agriculture of the GDP is at most 100%:

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER CHECK (Unemployment > 0),
       CHECK (Industry + Service + Agriculture <= 100));
```

- if a condition is added that does not hold in the current database state, an error message is returned.

```
ALTER TABLE City
  ADD (CONSTRAINT citypop CHECK (Population > 100000));
```


Modify Column Definitions of a Table

- column conditions can be added by `ALTER TABLE ... ADD`.

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<col> [<datatype>] [DEFAULT <value>]
          [<colConstraint> ... <colConstraint>],
          ⋮
          <col> [<datatype>] [DEFAULT <value>]
          [<colConstraint> ... <colConstraint>])
  DROP <drop-clause>
  ... ;
```

- for `<colConstraint>`, only `NULL` and `NOT NULL` are allowed here.

  All other conditions must be added by `ALTER TABLE ... ADD (<tableConstraint>)`.

```
ALTER TABLE Country MODIFY (Capital NOT NULL);
ALTER TABLE encompasses
  ADD (PRIMARY KEY (Country,Continent));
ALTER TABLE Desert
  ADD (CONSTRAINT DesertArea CHECK (Area > 10));
```

- Error message, if a condition is added that is not satisfied in the current database state.

ALTER TABLE ... DROP/DISABLE/ENABLE

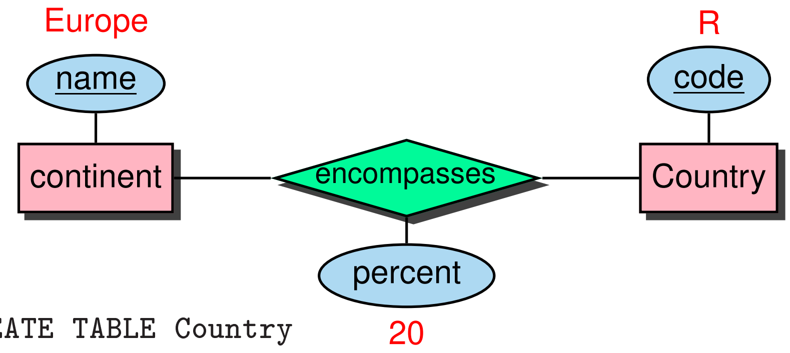
- (Integrity)constraints on a table
  - delete,
  - deactivate for some time,
  - reactivate.

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP    PRIMARY KEY [CASCADE] |
          UNIQUE (<column-list>) |
          CONSTRAINT <constraint>
  DISABLE PRIMARY KEY [CASCADE] |
          UNIQUE (<column-list>) |
          CONSTRAINT <constraint> | ALL TRIGGERS
  ENABLE  PRIMARY KEY |
          UNIQUE (<column-list>) |
          CONSTRAINT <constraint> | ALL TRIGGERS;
```

- `PRIMARY KEY` must not be deleted/disabled as long as there is a `REFERENCES` declaration to it.
- `DROP PRIMARY KEY CASCADE` deletes/disables corresponding `REFERENCES` declarations.
- `ENABLE`: if some constraints have been disabled cascadingly, they must be reactivated manually.

Referential Integrity

Referential integrity occur when in the transformation from the ER model to the relational model, key attributes of entities are incorporated into the relationship tables (correspondence between primary and foreign keys):




```
CREATE TABLE Country
 (Name     VARCHAR2(32),
  Code     VARCHAR2(4) PRIMARY KEY,
  ...);

CREATE TABLE Continent
 (Name     VARCHAR2(10) PRIMARY KEY,
  Area     NUMBER(2));

CREATE TABLE encompasses
 (Continent  VARCHAR2(10) REFERENCES Continent(Name),
  Country    VARCHAR2(4) REFERENCES Country(Code),
  Percentage NUMBER);
```

Referential Integrity

Country			
Name	Code	Capital	Province
Germany	D	Berlin	Berlin
United States	USA	Washington	Distr. Columbia
...	...	...	...

City		
Name	Country	Province
Berlin	D	Berlin
Washington	USA	Distr. Columbia
...	...	...

```
FOREIGN KEY (<attr-list>)
REFERENCES <table'> (<attr-list'>)
```

- (<attr-list'>) must be a candidate key of the referenced table.
- in ORACLE: must be declared as primary key.

Referential Integrity

- as column condition:

```
<attr> [CONSTRAINT <name>]
     REFERENCES <table'>(<attr'>)
```

```
CREATE TABLE City
(...
Country VARCHAR2(4)
      CONSTRAINT CityRefsCountry
      REFERENCES Country(Code) );
```

- as table condition:

```
[CONSTRAINT <name>]
  FOREIGN KEY (<attr-list>)
  REFERENCES <table'>(<attr-list'>)
```

```
CREATE TABLE Country
(...
CONSTRAINT CapitalRefsCity
  FOREIGN KEY (Capital,Code,Province)
  REFERENCES City(Name,Country,Province) );
```

Referential Actions

- if the contents of a table changes, actions are carried out for preserving referential integrity,
- if this is not possible, the changes are not executed, or even undone.

1. `INSERT` into a referenced table or `DELETE` from a referencing table does not cause any problems:

```
INSERT INTO Country
  VALUES ('Lummerland','LU',...);
DELETE FROM is_member ('D','EU');
```

2. `INSERT` or `UPDATE` in a referencing table must not generate foreign key values that do not exist in the referenced table:

```
INSERT INTO City
  VALUES ('Karl-Marx-Stadt','DDR',...);
```

   If the target key exists, there is no problem:

```
UPDATE City SET Country='A' WHERE Name='Munich';
```

3. `DELETE` und `UPDATE` of the referenced table: it is useful to adapt the referencing table by *referential actions* automatically:

   `UPDATE Country SET Code='UK' WHERE Code='GB';` or

   `DELETE FROM Country WHERE Code='I';`

Referential Actions in the SQL-2 Standard

`NO ACTION`:

The operation is executed; after execution, it is checked, whether "dangling references" occurred. If so, the operation is undone:

`DELETE FROM River;`

distinguish between the reference *River - River* and *located - River*!

`RESTRICT`:

The operation is executed only if no "dangling references" can occur:

`DELETE FROM Organization WHERE ...;`

error message if an organization would be deleted that still has some members.

`CASCADE`:

The operation is executed. Referencing tuples are also deleted or modified.

`UPDATE Country SET Code='UK' WHERE Code='GB';`

modifies also other tables:

Country: (United Kingdom,GB,...) $\leadsto$ (United Kingdom,UK,...)

Province: (Yorkshire,GB,...) $\leadsto$ (Yorkshire,UK,...)

City: (London,GB,Greater London,...) $\leadsto$ (London,UK,Greater London,...)

Referential actions in the SQL-2 Standard

`SET DEFAULT:`
: the operation is executed and for all referenced tuples, the foreign key value is set to the specified `DEFAULT` values (for which a corresponding tuple in the referenced relation must exist).

`SET NULL:`
: the operation is executed and for all referenced tuples, the foreign key value is set to the `NULL` value (for this, `NULL` values must be allowed).

`located`: city is located as a river/sea/lake
located(Bremerhaven,Nds.,D,Weser,NULL,North Sea)

```
DELETE * FROM River WHERE Name='Weser';
```

located(Bremerhaven,Nds.,D,NULL,NULL,North Sea)

Database Programming in SQL/ORACLE

Referential Actions in the SQL-2-Standard

Referential integrity constraints and referential actions are specified with the `CREATE TABLE` or `ALTER TABLE` command as

`<columnConstraint>` (for individual columns)

```
<col> <datatype>
  CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ ON DELETE {NO ACTION | RESTRICT | CASCADE |
               SET DEFAULT | SET NULL } ]
  [ ON UPDATE {NO ACTION | RESTRICT | CASCADE |
               SET DEFAULT | SET NULL } ]
```

or `<tableConstraint>` (for multiple columns)

```
CONSTRAINT <name>
  FOREIGN KEY (<attr-list>)
  REFERENCES <table'> (<attr-list'>)
  [ ON DELETE ...]
  [ ON UPDATE ...]
```

Referential Actions

Country			
Name	Code	Capital	Province
Germany	D	Berlin	Berlin
United States	USA	Washington	Distr. Columbia
...	...	...	...

CASCADE

NO ACTION

City		
Name	Country	Province
Berlin	D	Berlin
Washington	USA	Distr. Columbia
...	...	...

```
1.    DELETE FROM City WHERE Name='Berlin';
2.    DELETE FROM Country WHERE Name='Germany';
```


Referential Actions in ORACLE

- ORACLE 8: only `ON DELETE/UPDATE NO ACTION` and `ON DELETE CASCADE` are implemented.
- of no `ON ...` is specified, `NO ACTION` is used by default.
- `ON UPDATE CASCADE` is missing, which is cumbersome when applying updates.
- This has its reasons ...

Syntax as <columnConstraint>:

```
CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ON DELETE CASCADE]
```

Syntax as <tableConstraint>:

```
CONSTRAINT <name>
  FOREIGN KEY [ (<attr-list>)]
  REFERENCES <table'> (<attr-list'>)
  [ON DELETE CASCADE]
```

Referential Actions: UPDATE without CASCADE

Beispiel: Renaming of a country:

```
CREATE TABLE Country
  ( Name  VARCHAR2(32) NOT NULL UNIQUE,
    Code  VARCHAR2(4) PRIMARY KEY);
```

('United Kingdom','GB')

```
CREATE TABLE Province
  ( Name    VARCHAR2(32)
    Country VARCHAR2(4) CONSTRAINT ProvRefsCountry
                    REFERENCES Country(Code));
```

('Yorkshire','GB')

Now, the country code should be changed from 'GB' to 'UK'.

- `UPDATE Country SET Code='UK' WHERE Code='GB';`
  $\rightsquigarrow$ "dangling reference" of the old tuple ('Yorkshire','GB').
- `UPDATE Province SET Code='UK' WHERE Code='GB';`
  $\rightsquigarrow$ "dangling reference" of the new tuple ('Yorkshire','UK').

Referential Actions: UPDATE without CASCADE

- disable referential integrity constraint,
- apply updates,
- reactivate referential integrity constraint:

```
ALTER TABLE Province
  DISABLE CONSTRAINT ProvRefsCountry;
UPDATE Country
  SET Code='UK' WHERE Code='GB';
UPDATE Province
  SET Country='UK' WHERE Country='GB';
ALTER TABLE Province
  ENABLE CONSTRAINT ProvRefsCountry;
```

Referential Integrity Constraints

It is also possible to define a constraint with the table definition, and immediately disable it:

```
CREATE TABLE <table>
    ( <col> <datatype> [DEFAULT <value>]
            [<colConstraint> ... <colConstraint>],
      :
      <col> <datatype> [DEFAULT <value>]
            [<colConstraint> ... <colConstraint>],
      [<tableConstraint>],
      :
      [<tableConstraint>])
      DISABLE ...
      :
      DISABLE ...
      ENABLE ...
      :
      ENABLE ...;
```


Referential Actions: Cyclic References

Country

Name	Code	Capital	Province
Germany	D	Berlin	Berlin
United States	US	Washington	Distr.Col.
...	...	...	...

Province

Name	Country	Capital
Berlin	D	Berlin
Distr.Col.	US	Washington
...	...	...

City

Name	Country	Province
Berlin	D	B
Washington	USA	Distr.Col.
...	...	...

Referential Actions: Problems with `ON UPDATE`

Country			
Name	Code	Capital	Province
Germany	D	Berlin	Berlin
United States	US	Washington	Distr.Col.
...	...	...	...

CASCADE

SET NULL

Province		
Name	Country	Capital
Berlin	D	Berlin
Distr.Col.	US	Washington
...	...	...

CASCADE

City		
Name	Country	Province
Berlin	D	B
Washington	USA	Distr.Col.
...	...	...

```
DELETE FROM Country
WHERE Code='D'
```


Referential Actions

General case:

- already a single update may be ambiguous or even inconsistent when `ON DELETE/UPDATE SET NULL/SET DEFAULT` and `ON UPDATE CASCADE` are allowed.
- Due to SQL triggers an update often induces several other updates,
- non-trivial decision which updates should be triggered,
- in case of inconsistencies, their origin must be analyzed, and maximal admissible subsets must be investigated,
- stable models, exponential complexity.

Investigations on this topic in the dbis group:

- B. Ludäscher, W. May, and G. Lausen: Referential Actions as Logical Rules. In *Proc. 16th ACM Symposium on Principles of Database Systems*, Tucson, Arizona, 1997.
- B. Ludäscher, W. May, and G. Lausen: Triggers, Games, and Stable Models. Technical Report, IIF, 1997.
- B. Ludäscher, W. May: Referential Actions: From Logical Semantics to Implementation. In *Proc. 6th Intl. Conf. on Extending Database Technologies*, Valencia, Spain, 1998.

Views

- Combination with access permissions (later)
- presentation of the actual database in a different form for some users.

View Updates

- must be mapped onto updates of the base relation(s),
- not always possible.
- Table USER_UPDATABLE_COLUMNS in the Data Dictionary:

```
CREATE VIEW <name> AS ...
```

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = '<NAME>';
```


Database Programming in SQL/ORACLE

View Updates

- derived values cannot be changed:

Example:

```
CREATE OR REPLACE VIEW temp AS
SELECT Name, Code, Area, Population,
       Population/Area AS Density
FROM Country;
```

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = 'TEMP';
```

Table_Name	Column_Name	UPD	INS	DEL
temp	Name	yes	yes	yes
temp	Code	yes	yes	yes
temp	Area	yes	yes	yes
temp	Population	yes	yes	yes
temp	Density	no	no	no

```
INSERT INTO temp (Name, Code, Area, Population)
    VALUES ('Lummerland','LU',1,4)
SELECT * FROM temp where Code = 'LU';
```

- analogously for values that are computed by aggregate functions (COUNT, AVG, MAX, ...)

View Updates

Example:

```
CREATE VIEW CityCountry (City, Country) AS
   SELECT City.Name, Country.Name
   FROM City, Country
   WHERE City.Country = Country.Code;
```

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = 'CITYCOUNTRY';
```

Table_Name	Column_Name	UPD	INS	DEL
CityCountry	City	yes	yes	yes
CityCountry	Country	no	no	no

- city names can be changed:
  direct mapping to *City*:

```
UPDATE CityCountry
SET City = 'Wien'
WHERE City = 'Vienna';
```

```
SELECT * FROM City WHERE Country = 'A';
```

Name	Country	Province	...
Wien	A	Vienna	...
⋮	⋮	⋮	⋮

View Updates

Example:

- *Country* cannot be changed:

City	Country
Berlin	Germany
Freiburg	Germany

Mapping to base table would be ambiguous:

```
UPDATE CityCountry
SET Country = 'Poland'
WHERE Name = 'Berlin';
```

```
UPDATE CityCountry
SET Country = 'Deutschland'
WHERE Country = 'Germany';
```

```
DELETE FROM CityCountry
WHERE Name = 'Berlin';
```

```
DELETE FROM CityCountry
WHERE Country = 'Germany';
```

View Updates

- ORACLE: admissibility decided by heuristics,
- based only on schema information,
- not on the *current* database state!
- key properties are important.
- Key of a base table = key of the view:
  obvious mapping possible and unambiguous.
- key of a base table covers a key of the view: unambiguous translation, several tuples of the base table can be effected.
- key of a base table does not cover any key of the view: in general, no translation possible (see exercises).

View Updates

Example:

```
CREATE OR REPLACE VIEW temp AS
SELECT country, population
FROM Province A
WHERE population = (SELECT MAX(population)
                    FROM Province B
                    WHERE A.Country = B.Country);
```

```
SELECT * FROM temp WHERE Country = 'D';
```

Country	Name	Population
D	Nordrhein-Westfalen	17816079

```
UPDATE temp
SET population = 0 where Country = 'D';
SELECT * FROM Province WHERE Name = 'D';
```

Result: the population of the province with the highest population in Germany is set to 0. Thus, the view changes!

```
SELECT * FROM temp WHERE Country = 'D';
```

Country	Name	Population
D	Bayern	11921944

View Updates

- Tuples can drop out of the view definition,
- this can be prevented by the `WITH CHECK OPTION`:

Beispiel

```
CREATE OR REPLACE VIEW UScities AS
SELECT *
FROM City
WHERE Country = 'USA'
WITH CHECK OPTION;
```

```
UPDATE UScities
SET Country = 'D' WHERE Name = 'Miami';
```

```
FEHLER in Zeile 1:
ORA-01402: Verletzung der WHERE clause
           einer View WITH CHECK OPTION
```

- it is allowed to delete tuples from the view/base relation.

Materialized Views

- Views are computed from scratch for every query.

+ always represent the current database state.

- time-consuming, inefficient if the data changes only seldom.

⇒ *Materialized Views*

- are computed at definition time, and
- are updated whenever base relations change (e.g., by *triggers*).
- ⇒ problems of *view maintenance*.

User Authentification

- user name
- password
- `sqlplus /`: authorization via UNIX account

Access Permissions inside ORACLE

- access permissions associated to the ORACLE account
- initially defined by the DBA

Schema Concept

- each user is assigned an own *database schema* where his objects are located.
- *global* addressing of tables by `<username>.<table>` (e.g. `dbis.City`),
- in the own schema by `<table>`.

Database Programming in SQL/ORACLE

System Privileges

- entitle for schema operations

- `CREATE [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: user is allowed to generate schema objects of these types,
- `ALTER [ANY] TABLE/TYPE/TRIGGER/PROCEDURE`: user is allowed to change schema objects of these types,
- `DROP [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: user is allowed to delete schema objects of these types,
- `SELECT/INSERT/UPDATE/DELETE [ANY] TABLE`: user is allowed to read/create/change/delete tuples from tables.

- `ANY`: operation is allowed in *all* schemas,
- without `ANY`: operation is allowed only in the own schema.

In this course:

- `CREATE SESSION, ALTER SESSION, CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE CLUSTER.`
- permissions for accessing and changing the own tables are not mentioned explicitly (`SELECT TABLE`).

System Privileges

```
GRANT <privilege-list>
TO <user-list> | PUBLIC [ WITH ADMIN OPTION ];
```

- `PUBLIC`: every user gets a permission
- `ADMIN OPTION`: the grantee is allowed to grant this permission to other users.

Revoke permissions:

```
REVOKE <privilege-list> | ALL
FROM <user-list> | PUBLIC;
```

only if the user has granted this permission (cascading in the case of `ADMIN OPTION`).

Examples:

- `GRANT CREATE ANY INDEX, DROP ANY INDEX`
  `TO opti-person WITH ADMIN OPTION;`
  allows `opti-person` to create and delete indexes everywhere,
- `GRANT DROP ANY TABLE TO destroyer;`
  `GRANT SELECT ANY TABLE TO supervisor;`
- `REVOKE CREATE TABLE FROM clueless;`

Informations about access permissions in the data dictionary:

```
SELECT * FROM SESSION_PRIVS;
```


Object Privileges

allow for executing operations to existing schema objects.

- owner of a database object
- nobody else is allowed to use this object, except
- owner (or DBA) explicitly grants him some permissions:

  ```
  GRANT <privilege-list> | ALL [(<column-list>)]
  ON <object>
  TO <user-list> | PUBLIC
  [ WITH GRANT OPTION ];
  ```

- `<object>`: `TABLE`, `VIEW`, `PROCEDURE/FUNCTION`, `TYPE`,
- tables and views: detailed specification for `INSERT`, `REFERENCES`, and `UPDATE` by `<column-list>`,
- `<privilege-list>`: `DELETE`, `INSERT`, `SELECT`, `UPDATE` for tables and views,
  `INDEX`, `ALTER`, and `REFERENCES` for tables,
  `EXECUTE` for procedures, functions, and `TYPE`s.
- `ALL`: all privileges that one has for the corresponding object.
- `GRANT OPTION`: the grantee can grant the permission to other users.

Object Privileges

Revoke permissions:

```
REVOKE <privilege-list> | ALL
ON <object>
FROM <user-list> | PUBLIC
[CASCADE CONSTRAINTS];
```

- `CASCADE CONSTRAINTS` (bei `REFERENCES`): all referential integrity constraints, that are based on the revoked `REFERENCES` privilege are dropped.
- in case that a permission is obtained from several users, it is dropped with the last `REVOKE`.
- in case of `GRANT OPTION`, the revocation also cascades.

Granted and obtained permissions are stored in the Data Dictionary:

```
SELECT * FROM USER_TAB_PRIVS;
```

- permissions that one has granted for the own tables,
- permissions that one has obtained for other's tables

```
SELECT * FROM USER_COL_PRIVS;
SELECT * FROM USER_TAB/COL_PRIVS_MADE/RECD;
```

User roles are defined as prototypical patterns for maintaining permissions (e.g., student, dba, ...).

Database Programming in SQL/ORACLE

Synonyms

Schema objects can be accessed under another name as originally stored:

```
CREATE [PUBLIC] SYNONYM <synonym>
  FOR <schema>.<object>;
```

- Without `PUBLIC`: Synonym is defined only for its owner.
- `PUBLIC` creates system-wide synonyms. Only allowed if one has the `CREATE ANY SYNONYM` privilege.

Example: A user often needs the relation “City” from the “dbis” schema.

- `SELECT * FROM dbis.City;`
- ```
  CREATE SYNONYM City
    FOR dbis.City;
  ```

  ```
  SELECT * FROM City;
  ```

Delete synonyms: `DROP SYNONYM <synonym>;`
  ```

Access Restriction via Views

- `GRANT SELECT` cannot be restricted to columns.
- instead: use a view.

```
GRANT SELECT [<column-list>]   -- nicht erlaubt
ON <table>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

can be replaced by

```
CREATE VIEW <view> AS
  SELECT <column-list>
  FROM <table>;

GRANT SELECT
ON <view>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```


Access Restrictions via Views: Example

pol is owner of the relation *Country*, he wants to allow the user *geo* to read and write *Country* without the *Capital* column (and the column that gives the province where the capital is located)

View with appropriate access permissions for *geo*:

```
CREATE VIEW pubCountry AS
SELECT Name, Code, Population, Area
FROM Country;

GRANT SELECT, INSERT, DELETE, UPDATE
      ON pubCountry TO geo;
```

- References to views are not allowed.

```
<pol>:  GRANT REFERENCES (Code) ON Country TO geo;
<geo>:  ...  REFERENCES pol.Country(Code);
```

Optimization of the Database

- minimize number of secondary storage accesses
- keep as much data as possible in main memory

Storage:

- efficient access (search) to secondary memory
  $\longrightarrow$ access paths: indexes, hashing
- try to access data that semantically belongs together with a single access to secondary memory
  $\longrightarrow$ Clustering

Query optimization:

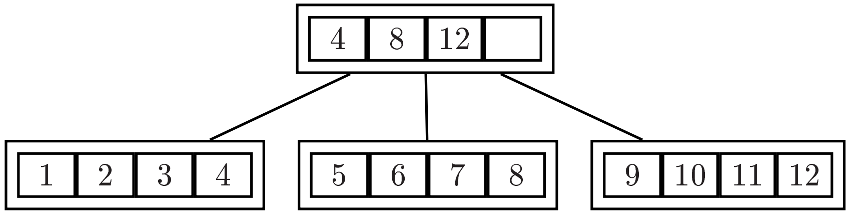
- keep amount of data small
- select early
- internal optimization strategies

Algorithmic optimization !

Access Paths: Indexes

Access by using indexes over columns is much more efficient.

- Trees; ORACLE: B*-tree,
- B*-tree: nodes contain *only* the information for searching for a value,
- high degree, height of the tree is small.



- searching by comparing keys: logarithmic effort.
- fast access (logarithmic) versus higher effort for reorganization ($\rightarrow$ algorithm theory),
- multiple indexes on a table possible (over different attribute sets),
- having many indexes on a table table may lead to poor performance for insertions, modifications, and deletions,
- logically and physically independent from the data of the corresponding table,
- no effect on the formulation of SQL statements,

Access Paths: Indexes

Access over indexed columns much more efficient:

- fetch index nodes from secondary memory,
- access the node that contains the tuple

```
CREATE TABLE zip
  (City       VARCHAR2(35)
   Country    VARCHAR2(4)
   Province   VARCHAR2(32)
   zip     NUMBER)

CREATE INDEX zipIndex ON zip (Country,zip);

SELECT *
  FROM zip
  WHERE zip = 79110 AND Country = 'D';
```

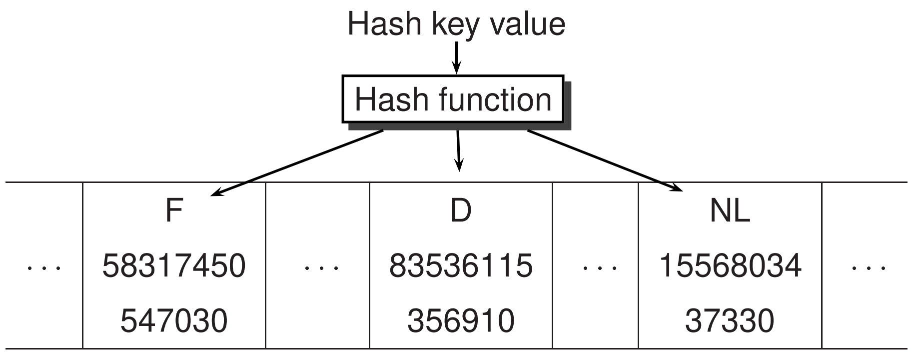

Hashing

Depending on the value(s) of one or more columns (*hash key*), a *hash function* is computed which indicates where the corresponding tuples are stored.

- access in *constant* time,
- no order of elements.

Example:

- access to the information about a specific country
  Hash key: Country.Code



In ORACLE, hashing is implemented only for *Clusters*.

Clusters

- collection of a group of tables which share one or more columns (cluster key), or
- special case: grouping of a table depending on one or more attributes.
- with a single secondary memory access, data that semantically belongs together is fetched into main memory.

Advantages of clustering:

- miminize the number of secondary memory access,
- saves memory space since cluster key is stored only once.

Disadvantages:

- inefficient if cluster keys are updated frequently since this requires a physical reorganization,
- loss of performance when inserting into clustered tables.

Clustering

Sea and geo_Sea with cluster key `Sea.Name`:

`Cl_Sea`		
Mediterranean Sea	`Depth`	
	5121	
	`Province`	`Country`
	Catalonia	E
	Valencia	E
	Murcia	E
	Andalusia	E
	Languedoc-R.	F
	Provence	F
	⋮	⋮
Baltic Sea	`Depth`	
	459	
	`Province`	`Country`
	Schleswig-H.	D
	Mecklenb.-Vorp.	D
	Szczecin	PL
	⋮	⋮

Clustering

City by (*Province,Country*):

Country	Province			
D	Nordrh.-Westf.	City	Population	...
		Düsseldorf	572638	...
		Solingen	165973	...
USA	Washington	City	Population	...
		Seattle	524704	...
		Tacoma	179114	...
⋮	⋮	⋮	⋮	⋮

Creating a Cluster in ORACLE

Create cluster and declare cluster key:

```
CREATE CLUSTER <name>(<col> <datatype>-list)
  [INDEX | HASHKEYS <integer> [HASH IS <funktion>]];
```

```
CREATE CLUSTER Cl_Sea (SeaName VARCHAR2(25));
```

Default: *indexed Cluster*, i.e., rows are indexed according to the cluster key.
Optional: `HASH`, with specifying a hash function for the cluster key values.

↪

Creating a Cluster in ORACLE

Assigning tables to a cluster by `CREATE TABLE`, with specification of the cluster key.

```
CREATE TABLE <table>
  (<col> <datatype>,
   ⋮
   <col> <datatype>)
  CLUSTER <cluster>(<column-list>);
```

```
CREATE TABLE CSea
  (Name   VARCHAR2(25) PRIMARY KEY,
   Depth  NUMBER)
  CLUSTER Cl_Sea (Name);
```

```
CREATE TABLE Cgeo_Sea
  (Province  VARCHAR2(32),
   Country   VARCHAR2(4),
   Sea       VARCHAR2(25))
  CLUSTER Cl_Sea (Sea);
```

Creating the cluster key index:
(must be done *before* the first DML command).

```
CREATE INDEX <name> ON CLUSTER <cluster>;
```

```
CREATE INDEX ClSeaInd ON CLUSTER Cl_Sea;
```


Prodecural Extensions: PL/SQL

- no procedural concepts in SQL (loops, if, variables)
- many tasks can only be performed awkwardly by using intermediate tables, or even impossible:
  - transitive closure
- programs represent application-specific *procedural* knowledge that is not contained in the database.

Extensions

- embedding of SQL into procedural host languages (*embedded SQL*); e.g., C, C++, or recently Java (JDBC),
- extending SQL with procedural elements *inside* the SQL environment, *PL/SQL* (*Procedural language extensions to SQL*).
- advantages of PL/SQL: better integration of procedural features into the database: procedures, functions, and triggers.
- required for object methods.

Block Structure of PL/SQL

```
Block Header
IS
    Declaration Section
BEGIN
    Execution Section
EXCEPTION
    Exception Section
END;
```

- block header: type of the object (function, procedure, or *anonymous* (inside another block)), and parameter declarations,
- declaration section: declarations of variables,
- execution section: command sequence of the block,
- exception section: reactions on errors.

Procedures

```
CREATE [OR REPLACE] PROCEDURE <proc_name>
  [(<parameter-list>)]
  IS <pl/sql-body>;
/
```

- OR REPLACE: if procedure definition already exists, it is overwritten.
- (<parameter-list>): declaration of formal parameters:

```
  (<variable> [IN|OUT|IN OUT] <datatype>,
    ⋮
   <variable> [IN|OUT|IN OUT] <datatype>)
```

- IN, OUT, IN OUT: specify how the procedure/function uses the parameter (read, write, both).
- default: IN.
- in case of OUT and IN OUT, the argument must always be an variable, in case of IN, also constants are allowed.
- <datatype>: all data types that are supported in PL/SQL; *without* length specification (VARCHAR2 instead of VARCHAR2(20)).
- <pl/sql-body> contains the definition of the procedure in PL/SQL.

Functions

Analogously, additionally the result type is specified:

```
CREATE [OR REPLACE] FUNCTION <funct_name>
  [(<parameter-list>)]
  RETURN <datatype>
  IS <pl/sql body>;
/
```

- PL/SQL functions are left by

  `RETURN <expression>;`

  Each function must contain at least one `RETURN` statement in its `<body>`.

- Functions must not have side effects.

Important: after the semicolon, a slash ("/"), must follow for executing the declaration!!!

In case of "... created with compilation errors":

```
SHOW ERRORS;
```

gives a more detailed error description.

Procedures and functions are deleted by
`DROP PROCEDURE/FUNCTION <name>`.

Database Programming in SQL/ORACLE

Procedures and Functions

- Invocation of procedures in a PL/SQL body:
  `<procedure> (arg1,...,argn);`
- Invocation of procedures in SQLPlus:
  `execute <procedure> (arg1,...,argn);`
- Usage of functions in PL/SQL:
  `... <function> (arg1,...,argn) ...`
  as in other programming languages.

The system-owned table `DUAL` is commonly used for displaying thr return value of functions:

```
SELECT <function> (arg1,...,argn)
FROM DUAL;
```

Example: Procedure

- Simple procedure: PL/SQL-Body contains only SQL statements

Information about countries is distributed over several relations.

```
CREATE OR REPLACE PROCEDURE InsertCountry
 (name VARCHAR2, code VARCHAR2, area NUMBER, pop NUMBER,
  gdp NUMBER, inflation NUMBER, pop_growth NUMBER)
IS
BEGIN
  INSERT INTO Country (Name,Code,Area,Population)
         VALUES (name,code,area,pop);
  INSERT INTO Economy (Country,GDP,Inflation)
         VALUES (code,gdp,inflation);
  INSERT INTO Population (Country,Population_Growth)
         VALUES (code,pop_growth);
END;
/
EXECUTE InsertCountry
   ('Lummerland', 'LU', 1, 4, 50, 0.5, 0.25);
```


Example: Function

- Simple function: population density of a country

```
CREATE OR REPLACE FUNCTION Density
 (arg VARCHAR2)
RETURN number
IS
  temp number;
BEGIN
  SELECT Population/Area
    INTO temp
    FROM Country
    WHERE code = arg;
  RETURN temp;
END;
/
SELECT Density('D')
FROM dual;
```

PL/SQL-Variables and Data Types.

Declaration of the PL/SQL Variables in the declaration section:

```
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
:
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
```

Simple data types:

`BOOLEAN`: `TRUE`, `FALSE`, `NULL`,

`BINARY_INTEGER`, `PLS_INTEGER`: Signed integers,

`NATURAL`, `INT`, `SMALLINT`, `REAL`, `...`: Numerical data types.

```
amount NUMBER DEFAULT 0;
name VARCHAR2(30);
```


anchored Type Declaration

By giving a PL/SQL variable or a table column (!) whose type should be used for a new variable:

```
<variable> <variable'>%TYPE
  [NOT NULL] [DEFAULT <value>];
or
<variable> <table>.<col>%TYPE
  [NOT NULL] [DEFAULT <value>];
```

- `cityname City.Name%TYPE`
  use the type of the Name column of the City table as the datatype of the newly defined variable.
- `%TYPE` is detected at compile time.

Variable Assignment

- "classical way" in the program:

```
a := b;
```

- assigning a (single-column and single-row!) result of a database query to a PL/SQL variable:

```
SELECT ...
INTO <PL/SQL-Variable>
FROM ...
```

Example:

```
the_name country.name%TYPE
        ⋮
SELECT name
INTO the_name
FROM country
WHERE name='Germany';
```


Database Programming in SQL/ORACLE

PL/SQL Data Types: Records

A RECORD consists of several fields, corresponding to a tuple of the database:

```
TYPE city_type IS RECORD
  (Name City.Name%TYPE,
   Country VARCHAR2(4),
   Province VARCHAR2(32),
   Population NUMBER,
   Longitude NUMBER,
   Latitude NUMBER);

the_city city_type;
```

anchored Type Declaration for Records

Records can be declared using a table definition: %ROWTYPE:

```
<variable> <table-name>%ROWTYPE;
```

equivalent to the above example:

```
the_city city%ROWTYPE;
```

Assignment to Records

- Aggregate assignment: two variables of the same record type:

```
<variable> := <variable'>;
```

- assignment of a single field:

```
<record.field> := <variable>|<value>;
```

- `SELECT INTO`: result of a query that yields a *single* tuple:

```
SELECT ...
INTO <record-variable>
FROM ... ;
```

```
the_country country%ROWTYPE
        ⋮
SELECT *
INTO the_country
FROM country
WHERE name='Germany';
```

Comparison of Records:
For comparing records, each field must be compared.

Database Programming in SQL/ORACLE

PL/SQL Data Types: PL/SQL Tables

Array-like structure, a *single column* with an arbitrary datatype (including `RECORD` types), usually indexed by `BINARY_INTEGER`.

```
TYPE <type> IS TABLE OF <datatype>
  [INDEX BY BINARY_INTEGER];
```

```
<var> <type>;
```

```
zip_table_type IS TABLE OF City.Name%TYPE
  INDEX BY BINARY_INTEGER;
```

```
zip_table zip_table_type;
```

- Addressing: `<var>(1)`

```
zip_table(79110):= Freiburg;
zip_table(33334):= Kassel;
```

- *sparse*: only those rows are stored that actually contain values.

Tables can also be assigned as a whole:

```
other_table := zip_table;
```

PL/SQL Data Types: PL/SQL Tables

PL/SQL tables provide *built-in* functions and procedures:

```
<variable> := <pl/sql-table-name>.<built-in-function>;
```

or

```
<pl/sql-table-name>.<built-in-procedure>;
```

- `COUNT` (fct): number of non-empty entries.
  zip_table.count = 2
- `EXISTS` (fct): `TRUE` is table non-empty.
- `DELETE` (proc): deletes all entries of a table.
- `FIRST/LAST` (fct): lowest/highest used index.
  zip_table.first = 33334
- `NEXT/PRIOR(n)` (fct): yields the next higher/lower used index value, starting from n .
  zip_table.next(33334) = 79110

Database Programming in SQL/ORACLE

SQL-Statements in PL/SQL

- DML-commands `INSERT`, `UPDATE`, `DELETE`, and `SELECT INTO` statements.
- these SQL statements may also contain PL/SQL variables.
- commands that *effect only a single tuple* can assign teir results to PL/SQL variables by using `RETURNING`:

```
UPDATE ...  SET ...  WHERE ...
RETURNING <expr-list>
INTO <variable-list>;
```

E.g., return the row-ID of the affected tuple:

```
DECLARE rowid ROWID;
BEGIN
    ⋮
   INSERT INTO Politics (Country,Independence)
     VALUES (Code,SYSDATE)
     RETURNING ROWID
     INTO rowid;
    ⋮
END;
```

- DDL-Statements are not supported directly by PL/SQL: DBMS_SQL-Package.

Control Structures

- `IF THEN - [ELSIF THEN] - [ELSE] - END IF`,
- several kinds of loops:
- Simple `LOOP`: `LOOP ... END LOOP;`
- `WHILE LOOP`:

  ```
  WHILE <condition> LOOP ... END LOOP;
  ```
- Numeric `FOR LOOP`:

  ```
  FOR <loop_index> IN
    [REVERSE] <from> .. <to>
   LOOP ... END LOOP;
  ```

  The variable `<loop_index>` is declared *automatically* as `INTEGER`.
- `EXIT [WHEN <condition>]`: leave `LOOP`.
- the well-known `GOTO` statement with labels:

  ```
  <<label_i>> ... GOTO label_j;
  ```
- `NULL` values always lead into the `ELSE` branch.
- `GOTO`: it is not allowed to jump into an `IF`, a `LOOP`, or a local block; also not from one `IF` branch into another.
- after a label, an executable statement must follow;
- `NULL` Statement (is executable).

Database Programming in SQL/ORACLE

Nested Blocks

Inside the *execution section*, *anonymous blocks* can be used for structuring. Here, the *Declaration Section* is introduced by `DECLARE` (there is no block header):

```
BEGIN
  -- statements of the outer block --
  DECLARE
   -- declarations of the inner block
  BEGIN
   -- statements of the inner block
  END;
  -- statements of the outer block --
END;
```

Cursor-Based Database Access

Row-wise access to a relation from an PL/SQL program.

Cursor declaration in the *declaration section*:

```
CURSOR <cursor-name> [(<parameter-list>)]
IS
  <select-statement>;
```

- (<parameter-list>): parameter list.
- only IN allowed for parameter communication.
- between SELECT and FROM, PL/SQL variables and PL/SQL-Functions are allowed. PL/SQL variables can also be used in the WHERE, GROUP, and HAVING clauses.

Example

Compute all cities which are located in the country specified by the variable the_country:

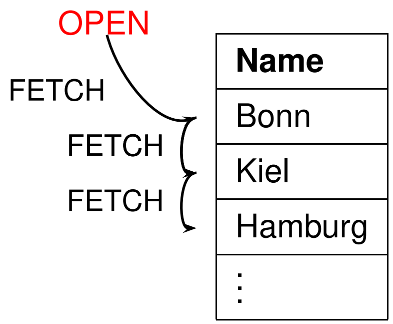
```
DECLARE CURSOR cities_in
   (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```


Cursors

- OPEN <cursor-name>[(<argument-list>)];

  creates a *virtual table* for the result of the given SELECT statement and defines a “window” that is placed over one of the tuples and can be moved forwards stepwise. OPEN executes the query and initializes the cursor:

  ```
  OPEN cities_in ('D');
  ```



Cursors

- FETCH <cursor-name> INTO <record-variable>; or
  FETCH <cursor-name> INTO <variable-list>;
  moves the cursor to the next row of the result of the query and copies this row into the given record variable or variable list.

  The variable can e.g. be declared with the record type of the cursor by using <cursor-name>%ROWTYPE:

  ```
  <variable> <cursor-name>%ROWTYPE;
  ```

- CLOSE <cursor-name>; closes the cursor.

Example

```
DECLARE CURSOR cities_in
   (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
city_in cities_in%ROWTYPE;
BEGIN
   OPEN cities_in ('D');
   FETCH cities_in INTO city_in;
   CLOSE cities_in;
END;
```


Cursors

not allowed:

```
OPEN cities_in ('D');
OPEN cities_in ('CH');
FETCH cities_in INTO <variable>;
```

- *one* parameterized cursor,
- *not* a family of cursors!

Cursors: Attributes

- <cursor-name>%ISOPEN: Cursor open?
- <cursor-name>%FOUND: as long as the preceding FETCH operation has been successful (i.e., the cursor has been moved to a valid tuple), <cursor-name>%FOUND = TRUE.
- <cursor-name>%NOTFOUND: TRUE if all rows of a cursor have been FETCHed.
- <cursor-name>%ROWCOUNT: number of tuples that have already been read from the cursor.
- not allowed inside SQL expressions.

Cursor FOR LOOP

```
FOR <record_index> IN <cursor-name>
LOOP ... END LOOP;
```

- `<record_index>` is *automatically* declared as a variable of the type `<cursor-name>%ROWTYPE`,
- `<record_index>` is *always* of a record type (including one-column records).
- `OPEN` is executed automatically.
- for each execution of the loop body, `FETCH` is done *automatically*,
- → loop body does *not* contain a `FETCH` statement,
- at the end, `CLOSE` is also executed automatically,
- columns must be addressed explicitly.

Cursor FOR LOOP

Example: for every city in a given country, a procedure "request_Info" should be invoked:

```
DECLARE CURSOR cities_in
   (the_country country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country = the_country;
BEGIN
   the_country:='D'; % or something else
   FOR the_city IN cities_in(the_country)
   LOOP
      request_Info(the_city.name);
   END LOOP;
END;
```

Cursor FOR LOOP

- SELECT statement can also be written directly into the FOR clause.

```
CREATE TABLE big_cities
(name VARCHAR2(25));

BEGIN
  FOR the_city IN
   SELECT Name
   FROM City
   WHERE Country = the_country
   AND Population > 1000000
  LOOP
   INSERT INTO big_cities
      VALUES (the_city.Name);
  END LOOP;
END;
```


Writing on a Cursor

With WHERE CURRENT OF <cursor-name>, the most recently FETCHed tuple of <cursor-name> can be accessed:

```
UPDATE <table-name>
SET <set_clause>
WHERE CURRENT OF <cursor_name>;

DELETE FROM <table-name>
WHERE CURRENT OF <cursor_name>;
```

Note that the placement of the cursor over a base table tuple uniquely gives the position of the update (in contrast to View Updates).

Access Permissions

Invocation permission for functions/procedures:

- `GRANT EXECUTE ON <procedure/function> TO <user>;`
- procedures and functions are always executed with the access permissions of the *owner*.
- after

  `GRANT EXECUTE ON <procedure/function> TO <user>;`

  the user can execute this procedure/function, even if he has no access permission for the tables that are used by the procedure.
- possibility for defining access permissions that are more strict than `GRANT ... ON <table> TO ...`:
  access is allowed only in a special context that is defined by the procedure/function.

Database Programming in SQL/ORACLE

Nested Tables under PL/SQL

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

The use of nested tables in ORACLE causes some problems:
"Give all countries where german is spoken, and give the percentage of the german language in these countries"

Such a query has to search the inner table for *every* tuple in *Nested_Languages*.

- `SELECT THE` returns only a single object,
- no correlation with the surrounding tuple.
- use a (Cursor) loop.

Nested Tables under PL/SQL

```
CREATE TABLE tempCountries
 (Country    VARCHAR2(4),
  Language   VARCHAR2(20),
  Percentage NUMBER);

CREATE OR REPLACE PROCEDURE Search_Countries
      (the_Language IN VARCHAR2)
IS CURSOR countries IS
   SELECT Code
   FROM Country;
BEGIN
   DELETE FROM tempCountries;
   FOR the_country IN countries
   LOOP
      INSERT INTO tempCountries
      SELECT the_country.code,Name,Percentage
      FROM THE(SELECT Languages
               FROM Nested_Language
               WHERE Country = the_country.Code)
      WHERE Name = the_Language;
   END LOOP;
END;
/

EXECUTE Search_Countries('German');
SELECT * FROM tempCountries;
```


Database Programming in SQL/ORACLE

- Up to now: functions and procedures are explicitly called by the user.
- Triggers: invocation is caused by an event inside the database.

Intermezzo: integrity constraints

- column constraints and table constraints,
- domain constraints,
- prohobiting Null values,
- uniqueness and primary key constraints,
- `CHECK`-constraints,

! these are only conditions on a *single* row of a *single* table.

Assertions

- conditions that are concerned with the whole database state.

  `CREATE ASSERTION <name> CHECK (<condition>)`
- not supported by ORACLE8.

⇒ other solution?

Trigger

- special form of PL/SQL procedures,
- are invoked when a certain event takes place.
- Special case of *active rules* according to the **E**vent-**C**ondition-**A**ction paradigm.
- assigned to a table (often, to a certain column of this table).
- invocation is caused by detection of some event in the table (insertion, modification, or deletion of a row).
- execution also depends on a condition on the database state.
- action:
- *before* or *after* execution of the activating statement
- executed once per activating statement (statement trigger) or once for each effected row (Row-Trigger).
- the body of the trigger can read the old and the new value of the tuple,
- the body of the trigger can *write the new value of the tuple*.

Trigger

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  BEFORE | AFTER
  {INSERT | DELETE | UPDATE} [OF <column-list>]
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  :
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  ON <table>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH ROW]
  [WHEN (<condition>)]
  <pl/sql-block>;
```

- BEFORE, AFTER: trigger is invoked before/after the activating operation.
- OF <column> (only for UPDATE) restricts the activating event to the specified column.
- access to the fields of the tuple before and after executing the activating action by :OLD or :NEW. (Aliasing by REFERENCING OLD AS ... NEW AS ...).

  Writing the :NEW values only with BEFORE triggers.
- FOR EACH ROW: row-Trigger, otherwise statement trigger.
- WHEN (<condition>): additional condition; OLD and NEW are allowed in <condition>.

Trigger: Example

If a country code is changed, this modification is propagated to the relation *Province*:

```
CREATE OR REPLACE TRIGGER change_Code
BEFORE UPDATE OF Code ON Country
FOR EACH ROW
BEGIN
  UPDATE Province
  SET Country = :NEW.Code
  WHERE Country = :OLD.Code;
END;
/

UPDATE Country
SET Code = 'UK'
WHERE Code = 'GB';
```

Trigger: Example

If a country is created, an entry in *Politics* is created with the current date:

```
CREATE TRIGGER new_Country
AFTER INSERT ON Country
FOR EACH ROW
BEGIN
   INSERT INTO Politics (Country,Independence)
   VALUES (:NEW.Code,SYSDATE);
END;
/

INSERT INTO Country (Name,Code)
  VALUES ('Lummerland', 'LU');

SELECT * FROM Politics;
```


Trigger: Mutating Tables

- row-based trigger are always called immediately before/after changing the row
- each invocation of the triggers sees another database state of the table on which it is defined, and of the tables which are changed by the trigger
- $\leadsto$ result *depends on the order of tuples*.

ORACLE: affected tables are marked as *mutating* during the whole action. They cannot be read by the trigger.

Problem: a too strict criterion.

- if a trigger should access the table on which it is defined:
  - only the activating tuple should be read/written by the trigger: Use a `BEFORE` trigger and the `:NEW` and `:OLD` variables
  - additional tuples must be used: if possible, use a statement trigger
  - otherwise, use auxiliary tables.

INSTEAD OF Triggers

- *view updates*: updates must be translated to base tables.
- view updating mechanisms are restricted.
- INSTEAD OF-Trigger: modification of a view is *replaced* by other SQL statements.

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  INSTEAD OF
  {INSERT | DELETE | UPDATE} ON <view>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH STATEMENT]
  <pl/sql-block>;
```

- cannot be restricted to columns
- no WHEN clause
- Default: FOR EACH ROW

View Updates and INSTEAD OF Triggers

```
CREATE OR REPLACE VIEW AllCountry AS
SELECT Name, Code, Population, Area,
       GDP, Population/Area AS Density,
       Inflation, population_growth,
       infant_mortality
FROM Country, Economy, Population
WHERE Country.Code = Economy.Country
  AND Country.Code = Population.Country;

INSERT INTO AllCountry
 (Name, Code, Population, Area, GDP,
  Inflation, population_growth, infant_mortality)
 VALUES ('Lummerland','LU',4,1,0.5,0,25,0);
```

Error message: ”Uber ein Join-View kann nur *eine* Basistabelle modifiziert werden.

View Updates and INSTEAD OF Triggers

```
CREATE OR REPLACE TRIGGER InsAllCountry
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
  INSERT INTO
    Country (Name,Code,Population,Area)
  VALUES (:NEW.Name, :NEW.Code,
          :NEW.Population, :NEW.Area);
  INSERT INTO Economy (Country,Inflation)
  VALUES (:NEW.Code, :NEW.Inflation);
  INSERT INTO Population
      (Country, Population_Growth,infant_mortality)
  VALUES (:NEW.Code, :NEW.Population_Growth,
          :NEW.infant_mortality);
END;
/
```

- updates *Country*, *Economy* and *Population*.
- trigger *New_Country* (AFTER INSERT ON COUNTRY) also updates *Politics*.

Database Programming in SQL/ORACLE

Error Handling

- Declaration Section: declaration (of names) of user-defined exceptions.

  ```
  DECLARE <exception> EXCEPTION;
  ```

- Exception Section: Definition of actions that have to be executed in case of an exception.

  ```
  WHEN <exception>
     THEN <PL/SQL-Statement>;
  WHEN OTHERS THEN <PL/SQL-Statement>;
  ```

- Exceptions can be raised on arbitrary places on the PL/SQL block by the RAISE statement.

  ```
  IF <condition>
    THEN RAISE <exception>;
  ```

Execution

- raise of an exception
- execute the corresponding action in the WHEN
- leave innermost block (use anonymous blocks)

Triggers/Error Handling: Example

In the afternoon, it is not allowed to delete cities:

```
CREATE OR REPLACE TRIGGER bla
BEFORE DELETE ON City
BEGIN
   IF TO_CHAR(SYSDATE,'HH24:MI')
      BETWEEN '12:00' AND '18:00'
   THEN RAISE_APPLICATION_ERROR
      (-20101,'Unerlaubte Aktion');
    END IF;
END;
/
```


Example

```
CREATE OR REPLACE TRIGGER bla
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
 IF user='may'
  THEN NULL;
 END IF;
...
END;
/

INSERT INTO AllCountry
 (Name, Code, Population, Area, GDP, Inflation,
  population_growth, infant_mortality)
 VALUES ('Lummerland','LU',4,1,0.5,0,25,0);

1 Zeile wurde erstellt.

SQL> select * from allcountry where Code='LU';

Es wurden keine Zeilen ausgewaehlt
```

(from A. Christiansen, M. Höding, C. Rautenstrauch and G. Saake, ORACLE 8 effizient einsetzen, Addison-Wesley, 1998)

Further PL/SQL Features

- *Packages*: encapsulate data and programs;
- `FOR UPDATE` option in cursor declarations;
- *cursor variables*;
- *exception handlers*;
- *named* parameter passing;
- PL-SQL built-in functions: parsing, string operations, date operations, numerical functions;
- built-in packages.

- definition of complex transactions,
- usage of `SAVEPOINTs` for transactions,

Object-Relational Database Systems

Integration of relational concepts and object orientation:

- complex data types: extend the domain concept of SQL-2
- abstract data types (“Object types”): object identity and encapsulation of internal functionality.
- specialization: class hierarchy; subtypes as specialization of more general types.
- subtables.
- functions as parts of ADT’s or tables, or free functions.
- method calls inside of `SELECT` statements

Object Orientation

- distinction between the *state* and *behavior* of an *object*.
- in ORACLE 8: tables of tuples vs. *object tables* (which contain objects)
- in contrast to a *tuple*, an object has *attributes* (which describe its state) and *methods* (for querying and changing its state).
- type defines signature of a set of instances (objects)
- already mentioned: complex attribute types, having only *value attributes*, no methods.
- methods: *procedures* and *functions*
- `MAP/ORDER`-function: order of instances of an object type
- columns in a relational table can be *object-valued* or *reference-valued*.
- Objects: *value attributes* and *reference attributes*.
- ORACLE8: no subtypes, no inheritance.

Type declaration: attributes, *signatures* of methods, `READ/WRITE` access characteristics.
Type Body: implementation of the methods in PL/SQL.

Object Type Declarations

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   :
   <attr> REF <object-datatype>,
   :
   MEMBER FUNCTION <func-name> [(<parameter-list>)]
      RETURN <datatype>,
   :
   MEMBER PROCEDURE <proc-name> [(<parameter-list>)],
   :
   [ MAP MEMBER FUNCTION <func-name>
         RETURN <datatype>, |
     ORDER MEMBER FUNCTION <func-name>(<var> <type>)
         RETURN <datatype>,]
   [ <pragma-declaration-list>]
   );
```

- `<parameter-list>` as in PL/SQL,
- similar to `CREATE TABLE`, but *no* integrity constraints (are done later with the definition of (object) tables)

PRAGMA Clauses:

Read/Write Access Characteristics

<pragma-declaration-list>:

for every method, a PRAGMA clause is given:

```
PRAGMA RESTRICT_REFERENCES
  (<method_name>,<feature-list>);
```

<feature-list>:

WNDS	Writes no database state,
WNPS	Writes no package state,
RNDS	Reads no database state,
RNPS	Reads no package state.

Functions: are only executed if it is *explicitly asserted* that they do not change the database state:

```
PRAGMA RESTRICT_REFERENCES
  (<function_name>,WNPS,WNDS);
```

MAP/ORDER **functions:** no database access allowed

```
PRAGMA RESTRICT_REFERENCES
  (<function-name>,WNDS,WNPS,RNPS,RNDS)
```

⇒ uses only the state of the object itself.

Database Programming in SQL/ORACLE

Example: Geo-Coordinates

- method *Distance*(geo-coord-value)
- MAP method: distance from Greenwich.

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
  (Longitude NUMBER,
   Latitude NUMBER,
   MEMBER FUNCTION
      Distance (other IN GeoCoord)
      RETURN NUMBER,
   MAP MEMBER FUNCTION
      Distance_Greenwich RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance,WNPS,WNDS,RNPS,RNDS),
   PRAGMA RESTRICT_REFERENCES
      (Distance_Greenwich,WNPS,WNDS,RNPS,RNDS)
   );
/
```

Type Body

- Implementation of object methods,
- has to conform with the signature given for `CREATE TYPE`,
- for *all* declared methods, an implementation must be given.
- method `SELF` for accessing the attributes of the host object.

Type Body

```
CREATE [OR REPLACE] TYPE BODY <type>
AS
   MEMBER FUNCTION <func-name> [(<parameter-list>)]
      RETURN <datatype>
   IS
      [<var-decl-list>;]
      BEGIN <PL/SQL-code> END;
   ⋮
   MEMBER PROCEDURE <proc-name> [(<parameter-list>)]
   IS
      [<var-decl-list>;]
      BEGIN <PL/SQL-code> END;
   ⋮
   [MAP MEMBER FUNCTION <func-name>
        RETURN <datatype> |
    ORDER MEMBER FUNCTION <func-name>(<var> <type>)
        RETURN <datatype>
    IS
        [<var-decl-list>;]
        BEGIN <PL/SQL-code> END;]
END;
/
```

Object Creation

- Constructor method:

  `<type>(<arg_1>, ..., <arg_n>)`

Method Invocation

(from a PL-SQL program)

`<object>.<method-name>(<argument-list>)`

using `SELF`, <object> can invoke its own methods.

Example: Geo-Coordinates

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS

MEMBER FUNCTION Distance (other IN GeoCoord)
  RETURN NUMBER
  IS
  BEGIN
   RETURN 6370 * ACOS(cos(SELF.latitude/180*3.14)
               * cos(other.latitude/180*3.14)
               * cos((SELF.longitude-
                      other.longitude)/180*3.14)
               + sin(SELF.latitude/180*3.14)
               * sin(other.latitude/180*3.14));
  END;

MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0,51));
  END;

END;
/
```

Column Objects

- Attribute of a tuple (or of an object) can be object-valued,
- *no* OID, i.e., *not* referencable.

Example: Geo-Coordinates

```
CREATE TABLE Mountain
  (Name VARCHAR2(20) CONSTRAINT MountainKey PRIMARY KEY,
   Height NUMBER CONSTRAINT MountainHeight
      CHECK (Height > = 0),
   Coordinates GeoCoord CONSTRAINT MountainCoord
      CHECK ((Coordinates.Longitude >= -180) AND
             (Coordinates.Longitude <= 180) AND
             (Coordinates.Latitude >= -90) AND
             (Coordinates.Latitude <= 90)));
```

- Constraints are given as usual with the table definition:

```
INSERT INTO Mountain
   VALUES ('Feldberg', 1493, GeoCoord(8,48));
SELECT Name, mt.coordinates.distance(geocoord(0,90))
FROM Mountain mt;
```

- use the tuple-variable *mt* for disambiguating the navigation path to *coordinates.distance*.

Database Programming in SQL/ORACLE

Row Objects

- elements of *Object tables*,
- have a unique OID and are referencable.
- OID corresponds to the *primary key* and is specified together with (further) integrity constraints in the table definition.
- seamless combination with referential integrity constraints from object tables to existing relational tables.

```
CREATE TABLE <name> OF <object-datatype>
  [(<constraint-list>)];
```

<constraint-list>:

- attribute constraints correspond to column constraints:

```
<attr-name> [DEFAULT <value>]
   [<colConstraint> ... <colConstraint>]
```

- table constraints: syntax as for relational tables.

Row Objects

Example: *City_Type*

```
CREATE OR REPLACE TYPE City_Type AS OBJECT
  (Name VARCHAR2(35),
   Province VARCHAR2(32),
   Country VARCHAR2(4),
   Population NUMBER,
   Coordinates GeoCoord,
   MEMBER FUNCTION Distance (other IN City_Type)
      RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance,WNPS,WNDS,RNPS,RNDS));
/
```

```
CREATE OR REPLACE TYPE BODY City_Type
AS
  MEMBER FUNCTION Distance (other IN City_Type)
  RETURN NUMBER
  IS
  BEGIN
   RETURN self.coordinates.distance(other.coordinates);
  END;
END;
/
```


Object Tables: Row Objects

- the (multi-column) primary key is specified as a table condition,
- primary key must not contain reference attributes,
- the foreign key constraint to the relational table *Country* is also specified as a table condition:

```
CREATE TABLE City_ObjTab OF City_Type
  (PRIMARY KEY (Name, Province, Country),
   FOREIGN KEY (Country) REFERENCES Country(Code));
```

- Objects are inserted into object tables by using the object constructor `<object-datatype>`:

```
INSERT INTO City_ObjTab
SELECT City_Type
  (Name,Province,Country,Population,
   GeoCoord(Longitude,Latitude))
FROM City
WHERE Country = 'D'
  AND NOT Longitude IS NULL;
```

Using Objects

- select a row object *as a whole*,

  `VALUE (<var>)`

  in combination with aliasing

  `FROM <table> <var>`

- e.g. for a comparison or in an `ORDER BY` clause.

Example

```
SELECT VALUE(cty)
FROM City_ObjTab cty;
```

VALUE(Cty)(Name, Province, Country, Population, Coordinates(Longitude, Latitude))
City_Type('Berlin', 'Berlin', 'D', 3472009, GeoCoord(13, 52))
City_Type('Bonn', 'Nordrh.-Westf., 'D', 293072, GeoCoord(8, 50))
City_Type('Stuttgart', 'Baden-Wuertt., 'D', 588482, GeoCoord(9, 49))
⋮

Database Programming in SQL/ORACLE

Using Objects: `VALUE`

- check equality of objects
- object as argument of a method

```
SELECT cty1.Name, cty2.Name,
   cty1.coordinates.Distance(cty2.coordinates)
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

```
SELECT cty1.Name, cty2.Name,
   cty1.Distance(VALUE(cty2))
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

- assignment of an object to a PL/SQL variable by using a `SELECT INTO` statement:

```
SELECT VALUE(<var>) INTO <PL/SQL-Variable>
FROM <tabelle> <var>
WHERE ...  ;
```

Object References

- Additional datatype for attributes: references to objects

  ```
  <ref-attr> REF <object-datatype>
  ```

- `PRIMARY KEY`s must not contain `REF` attributes.
- *object type* as target of a reference
- only objects that have an OID – i.e., row objects in an object table – can be referenced.
- object type can be used in several tables
- restriction to a certain table can can be specified by constraints using the `SCOPE` concept:
  - as column constraint (only for relational tables):

    ```
    <ref-attr> REF <object-datatype>
      SCOPE IS <object-table>
    ```

  - as table constraint:

    ```
    SCOPE FOR (<ref-attr>) IS <object-table>
    ```

- generation of a reference (selection of an OID):

  ```
  SELECT ..., REF(<var>), ...
  FROM <object-table> <var>
  WHERE ... ;
  ```


Database Programming in SQL/ORACLE

Example: Object Type *Organization*

```
CREATE TYPE Member_Type AS OBJECT
 (Country VARCHAR2(4),
  Type VARCHAR2(30));
/
CREATE TYPE Member_List_Type AS
 TABLE OF Member_Type;
/

CREATE OR REPLACE TYPE Organization_Type AS OBJECT
  (Name VARCHAR2(80),
   Abbrev VARCHAR2(12),
   Members Member_List_Type,
   Established DATE,
   has_headq_in REF City_Type,
   MEMBER FUNCTION is_member (the_country IN VARCHAR2)
  -- EU.is_member('SL') = 'membership applicant'
      RETURN VARCHAR2,
   MEMBER FUNCTION people RETURN NUMBER,
   MEMBER FUNCTION number_of_members RETURN NUMBER,
   MEMBER PROCEDURE add_member
      (the_country IN VARCHAR2, the_type IN VARCHAR2),
   PRAGMA RESTRICT_REFERENCES (is_member,WNPS,WNDS),
   PRAGMA RESTRICT_REFERENCES (people,WNDS,WNPS));
   PRAGMA RESTRICT_REFERENCES (number_of_members,WNDS,WNPS))
/
```

Example: Object Type *Organization*

Table Definition:

```
CREATE TABLE Organization_ObjTab OF Organization_Type
  (Abbrev PRIMARY KEY,
   SCOPE FOR (has_headq_in) IS City_ObjTab)
   NESTED TABLE Members STORE AS Members_nested;
```

Inserting objects via the object constructor:

```
INSERT INTO Organization_ObjTab VALUES
  (Organization_Type('European Community','EU',
                     Member_List_Type(),NULL,NULL));
```

Reference attribute *has_headq_in*:

```
UPDATE Organization_ObjTab
SET has_headq_in =
  (SELECT REF(cty)
   FROM City_ObjTab cty
   WHERE Name = 'Brussels'
     AND Province = 'Brabant'
     AND Country = 'B')
WHERE Abbrev = 'EU';
```


Selecting Object Attributes

- value attributes

```
SELECT Name, Abbrev, Members
FROM Organization_ObjTab;
```

Name	Abbrev	Members
European Community	EU	Member_List_Type(...)

- Reference attributes:

  `SELECT <ref-attr-name>`

  yields an OID:

```
SELECT Name, Abbrev, has_headq_in
FROM Organization_ObjTab;
```

Name	Abbrev	has_headq_in
European Community	EU	<oid>

- `DEREF(<oid>)` yields the corresponding object:

```
SELECT Abbrev, DEREF(has_headq_in)
FROM Organization_ObjTab;
```

Abbrev	has_headq_in
EU	City_Type('Brussels', 'Brabant', 'B', 951580, GeoCoord(4, 51))

Usage of Reference Attributes

- Attributes and methods of a referenced object are addressed by *path expressions* of the form

  `SELECT <ref-attr-name>.<attr-name>`

  ("*navigational access*").
- aliasing with an object variable to disambiguate the path expression.

```
SELECT Abbrev, org.has_headq_in.name
FROM Organization_ObjTab org;
```

Abbrev	has_headq_in.Name
EU	Brussels

`REF` and `DEREF` can be used instead of `VALUE`:

```
SELECT VALUE(cty) FROM City_ObjTab cty;
```

and

```
SELECT DEREF(REF(cty)) FROM City_ObjTab cty;
```

are equivalent.

Database Programming in SQL/ORACLE

Cyclic References

- `City_Type: country REF Country_Type`
- `Country_Type: capital REF City_Type`
- declaration of each of the datatypes requires the definition of some other.
- Definition of *incomplete* types "forward declaration"

  ```
  CREATE TYPE <name>;
  /
  ```
- is replaced later by a complete type declaration

Cyclic References: Example

```
CREATE OR REPLACE TYPE City_Type
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/
CREATE OR REPLACE TYPE Province_Type AS OBJECT
  (Name VARCHAR2(32),
   Country REF Country_Type,
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/
CREATE OR REPLACE TYPE City_Type AS OBJECT
  (Name VARCHAR2(35),
   Province REF Province_Type,
   Country REF Country_Type,
   Population NUMBER,
   Coordinates GeoCoord);
/
```


Incomplete Datatypes: Usage and Example

Incomplete datatypes can only be used for defining *references* to them, not for defining columns or nested tables:

```
CREATE TYPE City_type;
/
allowed:
CREATE TYPE city_list AS TABLE OF REF City_type;
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital REF City_Type);
/
only allowed if city_type is complete:
CREATE TYPE city_list AS TABLE OF City_type;
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital City_Type);
/
```

Referential Integrity

- Cf. `FOREIGN KEY ... REFERENCES ... ON DELETE/UPDATE CASCADE`
- modifications of objects:
  OID remains unchanged
  → referential integrity is preserved.
- deletion of objects:
  dangling references possible.

  Check with

  ```
  WHERE <ref-attribute> IS DANGLING
  ```

  Usage e.g. in an `AFTER` trigger:

  ```
  UPDATE <table>
    SET <attr> = NULL
    WHERE <attr> IS DANGLING;
  ```


Methods: Functions and Procedures

- `TYPE BODY` contains the implementations of the methods in PL/SQL
- PL/SQL is adapted to nested tables and some object-oriented features.
- PL/SQL does not support navigation along path expressions (which is allowed in SQL).
- every `MEMBER METHOD` has an *implicit* parameter `SELF` that references the host object itself.
- table-valued attributes can be handled inside PL/SQL like PL/SQL-tables:

  Built-in methods for collections (PL/SQL-Tables) can also be applied to table-valued attributes:

  `<attr-name>.COUNT`: number of tuples in the nested table

  Not allowed in SQL statements that are embedded into the PL/SQL body – e.g. `SELECT <attr>.COUNT`.
- future extension: Java

```
CREATE OR REPLACE TYPE BODY Organization_Type IS

MEMBER FUNCTION is_member (the_country IN VARCHAR2)
  RETURN VARCHAR2
IS
BEGIN
  IF SELF.Members IS NULL OR SELF.Members.COUNT = 0
   THEN RETURN 'no'; END IF;
  FOR i in 1 ..  Members.COUNT
  LOOP
   IF the_country = Members(i).country
      THEN RETURN Members(i).type; END IF;
  END LOOP;
  RETURN 'no';
END;

MEMBER FUNCTION people RETURN NUMBER IS
p NUMBER;
BEGIN
  SELECT SUM(population) INTO p
  FROM Country ctry
  WHERE ctry.Code IN
  (SELECT Country
   FROM THE (SELECT Members
                FROM Organization_ObjTab org
                WHERE org.Abbrev = SELF.Abbrev));
  RETURN p;
END;
```



```
MEMBER FUNCTION number_of_members RETURN NUMBER
IS
BEGIN
  IF SELF.Members IS NULL THEN RETURN 0; END IF;
  RETURN Members.COUNT;
END;

MEMBER PROCEDURE add_member
   (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
  IF NOT SELF.is_member(the_country) = 'no'
   THEN RETURN; END IF;
  IF SELF.Members IS NULL THEN
   UPDATE Organization_ObjTab
   SET Members = Member_List_Type()
   WHERE Abbrev = SELF.Abbrev;
  END IF;
  INSERT INTO
  THE (SELECT Members
   FROM Organization_ObjTab org
   WHERE org.Abbrev = SELF.Abbrev)
  VALUES (the_country,the_type);
END;
END;
/
```

- `FROM THE(SELECT ...)` cannot be replaced by `FROM SELF.Members` (PL/SQL vs. SQL).

Method Calls: Functions

- MEMBER FUNCTIONS can be invoked from SQL and PL/SQL by `<object>.<function>(<argument-list>)`.
- parameterless functions: `<object>.<function>()`
- from SQL: `<object>` is given as a path expression with alias.

```
SELECT Name, org.is_member('D')
  FROM Organization_ObjTab org
  WHERE NOT org.is_member('D') = 'no';
```

- MEMBER PROCEDURES can be invoked only from PL/SQL by `<object>.<procedure>(<argument-list>)`.
- free procedures in PL/SQL have to be used for invoking MEMBER PROCEDURES from SQL.

Method Calls: Procedures

```
CREATE OR REPLACE PROCEDURE make_member
  (the_org IN VARCHAR2, the_country IN VARCHAR2,
   the_type IN VARCHAR2) IS
  n NUMBER;
  c Organization_Type;
BEGIN
  SELECT COUNT(*) INTO n
   FROM Organization_ObjTab
   WHERE Abbrev = the_org;
  IF n = 0
  THEN INSERT INTO Organization_ObjTab
   VALUES(Organization_Type(NULL,
      the_org,Member_List_Type(),NULL,NULL));
  END IF;
  SELECT VALUE(org) INTO c
   FROM Organization_ObjTab org
   WHERE Abbrev = the_org;
  IF c.is_member(the_country)='no' THEN
   c.add_member(the_country,the_type);
  END IF;
END;
/
EXECUTE make_member('EU','USA','special member');
EXECUTE make_member('XX','USA','member');
```

Copying all data from the relational tables *Organization* and *is_member* to the object table *Organization_ObjTab*:

```
INSERT INTO Organization_ObjTab
 (SELECT Organization_Type
     (Name,Abbreviation,NULL,Established,NULL)
  FROM Organization);
```

```
CREATE OR REPLACE PROCEDURE Insert_All_Members IS
BEGIN
 FOR the_membership IN
     (SELECT * FROM is_member)
 LOOP make_member(the_membership.organization,
                  the_membership.country,
                  the_membership.type);
 END LOOP;
END;
/
EXECUTE Insert_All_Members;
```

```
UPDATE Organization_ObjTab org
SET has_headq_in =
  (SELECT REF(cty)
   FROM City_ObjTab cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
   AND cty.Name = old.City
   AND cty.Province = old.Province
   AND cty.Country = old.Country);
```


Using Objects

```
CREATE OR REPLACE FUNCTION is_member_in
     (the_org IN VARCHAR2, the_country IN VARCHAR2)
 RETURN is_member.Type%TYPE IS
 c is_member.Type%TYPE;
BEGIN
  SELECT org.is_member(the_country) INTO c
  FROM Organization_ObjTab org
  WHERE Abbrev=the_org;
  RETURN c;
END;
/
```

The system-owned table `DUAL` can be used for displaying the result of free functions.

```
SELECT is_member_in('EU','SL')
FROM DUAL;
```

is_member_in('EU','SL')
applicant

It is not (at least not in ORACLE 8.0) possible to change table contents by using path expressions:

```
UPDATE Organization_ObjTab org
SET org.has_headq_in.Name = 'UNO City'   -- NOT ALLOWED
WHERE org.Abbrev = 'UN';
```

`ORDER`- and `MAP` Methods

- in contrast to most data types, object types do not have an inherent order.
- an order on objects of some type can be defined via functional methods.
- ORACLE 8: for each object type, a `MAP FUNCTION` or an `ORDER FUNCTION` can be specified.

`MAP` function:

- no parameters,
- maps each object to a number.
- linear order on an object type, "absolute value"
- suitable both for comparisons <, >, and `BETWEEN`, and for `ORDER BY`.

`ORDER` function:

- has *one* argument of the same object type that is compared to the host object.
- suitable for comparisons <, >, but in general not for sorting.

- `MAP` and `ORDER` functions require `PRAGMA RESTRICT_REFERENCES (<name>,WNDS,WNPS,RNPS,RNDS)`, i.e., they *must not contain any database access*.

`MAP` Methods: Example

`MAP` method on *GeoCoord*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
...
MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0,51));
  END;
END;
/
```

```
SELECT Name, cty.coordinates.longitude,
          cty.coordinates.latitude
FROM City_ObjTab cty
WHERE NOT coordinates IS NULL
ORDER BY coordinates;
```

`MAP` Methods

Some operations are not allowed in the body of `MAP` methods:

- no database queries:

  In *Organization_Type*, *People* cannot be used as `MAP`.
- no built-in methods of nested tables:

  In *Organization_Type*, *number_of_members* can also not be used as `MAP` method.

Database Programming in SQL/ORACLE

`ORDER` Methods

- comparison between `SELF` an another object of the same type that is given as a parameter.
- result: -1 (`SELF` < parameter), 0 (equality), or 1 (`SELF` > parameter)
- in case of `ORDER BY`, the objects are compared pairwise and output according to the results of the `ORDER` method.
- an example for this a soccer league table: a team is placed higher than another it it has more points. In case of an equal number of point, the goal difference decides. If this also coincides, the number of scored goals decides (cf. exercises).

Indexes on Attributes of Objects

Indexes can also be created over attributes of objects:

```
CREATE INDEX <name>
  ON <object-table-name>.<attr>[.<attr>]*;
```

- indexes *cannot* be created for complex attributes:

```
-- not allowed:
CREATE INDEX city_index
   ON City_ObjTab(coordinates);
```

- indexes can be created for atomic components of complex attributes:

```
CREATE INDEX city_index
   ON City_ObjTab(coordinates.Longitude,
                  coordinates.Latitude);
```

Access Permissions for Objects

Permission to use an object type:

```
GRANT EXECUTE ON <Object-datatype> TO ...
```

- when using an object type, its methods (including its constructur method) play the major role.

Database Programming in SQL/ORACLE

Modifications of Object Types

- using object types and reference attributes induces a network that is similar to the one defined by keys and referential integrity constraints.
- modifications of object types in ORACLE 8.0 are restricted: `CREATE OR REPLACE TYPE` and `ALTER TYPE` are (at least in ORACLE 8.0) not allowed if the object type is used somewhere.

! it is not possible to add some attribute (or even only a method!) to an object type that is used somewhere.

“In conclusion, carefully plan the object types for your database so that you get things right the first time. Then keep your fingers crossed and hope that things do not change once you have everything up and running (ORACLE 8: Architecture)”.

A First Conclusion

- Data management in an object-oriented schema is problematic already for minor schema modifications.
- application-oriented (non-relational) representation by methods and free procedures and functions.
- integration of application-specific functionality is supported by object methods.

⇒ Data management: relational model
user interface: object-oriented model.

Object-Views

- powerful object views tailored to application-specific requirements

Legacy Databases: integration of already existing databases into a “modern”, object-oriented model:

define *object views* over the relational level:
“object abstractions”

Efficiency + user friendliness:
relational representation is often more efficient:

- nested tables internally stored as separate tables.
- $n : m$-Relationships: require pairs of nested tables.

⇒ definition of a relational base schema (conceptual model) with object views (external schemata).

Modifiability: `CREATE OR REPLACE TYPE` and `ALTER TYPE` are restricted

⇒ changes are captured by the redefinition of the object view level.

Object Views

User updates are given wrt. the external schema that is given by object views:

- mapping of generic updates (`INSERT`, `UPDATE`, and `DELETE`) to the conceptual/physical schema by `INSTEAD OF`-Triggers, or
- generic updates are disallowed. Instead, the functionality is provided by methods of object types that execute the changes directly on the base tables.

Object-Relational Views

- Tuple-views without methods:

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
  <select-clause>;
```

- `SELECT`-clause: additional constructor method for objects and nested tables.
- for creating nested tables for object views, the `CAST` and `MULTISET` constructs are used.

Example

```
CREATE TYPE River_List_Entry AS OBJECT
 (name VARCHAR2(20),
  length NUMBER);
/
CREATE TYPE River_List AS
 TABLE OF River_List_Entry;
/
CREATE OR REPLACE VIEW River_V
  (Name, Length, Tributary_Rivers)
AS SELECT
   Name, Length,
   CAST(MULTISET(SELECT Name,Length FROM River
                 WHERE River = A.Name) AS River_List)
   FROM River A;
```

Object Views

- contain row objects, i.e., in this case, new objects are *defined*,
- `WITH OBJECT OID <attr-list>` specifies how the object-ID is computed based on the object state.
- use `CAST` and `MULTISET`.

```
CREATE [OR REPLACE] VIEW <name> OF <type>
  WITH OBJECT OID (<attr-list>)
  AS <select-statement>;
```

- in `<select-statement>` the object constructor is *not* used explicitly!

Database Programming in SQL/ORACLE

Object Views: *Country*

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
 (Name       VARCHAR2(32),
  Code       VARCHAR2(4),
  Capital    REF City_Type,
  Area       NUMBER,
  Population NUMBER);
/
```

```
CREATE OR REPLACE VIEW Country_ObjV OF Country_Type
WITH OBJECT OID (Code)
AS
SELECT Country.Name, Country.Code, REF(cty),
       Area, Country.Population
FROM Country, City_ObjTab cty
WHERE cty.Name = Country.Capital
  AND cty.Province = Country.Province
  AND cty.Country = Country.Code;
```

```
SELECT Name, Code, c.capital.name, Area, Population
FROM Country_ObjV c;
```

Object Views: what's not (yet?) allowed

- Object View must not contain nested tables,
- and it must not contain any result of a functional method of objects of the base table.

Object View based on *Organization_ObjTab*:

```
CREATE OR REPLACE TYPE Organization_Ext_Type AS OBJECT
  (Name VARCHAR2(80),
   Abbrev VARCHAR2(12),
   Members Member_List_Type,
   established DATE,
   has_headq_in REF City_Type,
   number_of_people NUMBER);
/
```

```
CREATE OR REPLACE VIEW Organization_ObjV
  OF Organization_Ext_Type
AS
SELECT Name, Abbrev, Members, established,
       has_headq_in, org.people()
  FROM Organization_ObjTab org;
```

```
FEHLER in Zeile 3:
ORA-00932:  nicht "ubereinstimmende data types
```

Both attributes are also not allowed alone.

Conclusion

\+ Compatibility with the basic concepts of ORACLE 7.
  E.g., foreign key constraints from object tables to relational tables.

\+ *object views* allow for an object-oriented external schema.
  User interaction can be mapped to the internal schema by methods and `INSTEAD OF`-Triggers.

– Flexibility/Maturity:
  types cannot be changed/extended.
  (incremental!) adaptions of the schema not possible.

Embedded SQL, JDBC

Coupling Modes between Database and Programming Languages

- extending the database language with programming constructs (e.g., PL/SQL)
- extending programming languages with database constructs:
  persistent programming languages, database programming languages
- embedding a database programming language into a programming language: "Embedded SQL"
- database access from the programming language with specialized constructs

Embedded SQL

- C, Pascal, C++

Impedance Mismatch with the SQL Embedding

- type systems do not fit
- different paradigms:
  set-oriented vs. individual, scalar variables

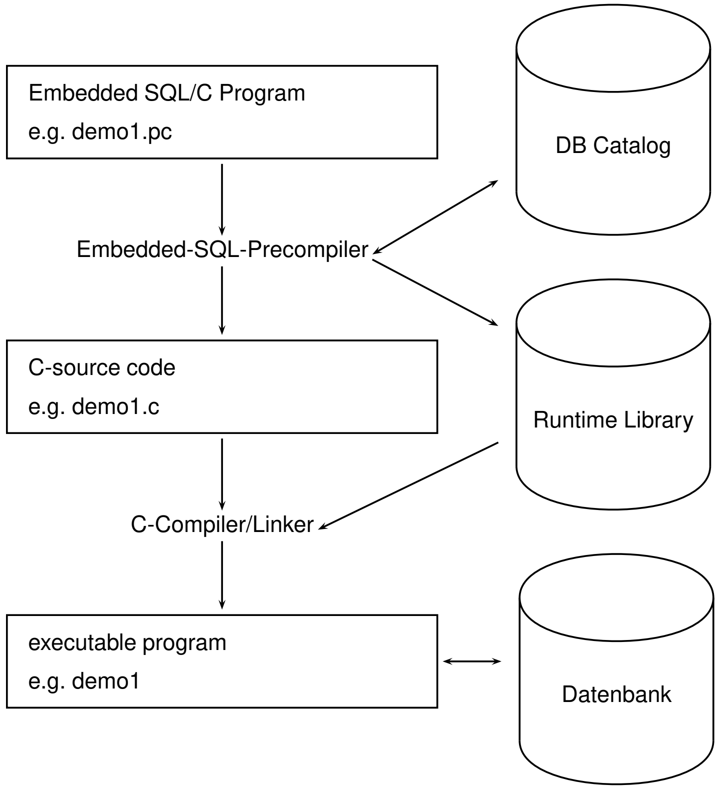
Practical Solution

- Mapping of tuples/attributes to data types of the host language
- iterative processing of the result set by a cursor

Effects on the Host Language

- Structure of the host language remains unchanged
- Every SQL statement can be embedded
- SQL statements are simply prefixed by `EXEC SQL`
- How to communicate between application program and database?

Development of an Embedded SQL Application



Connection

Application with embedded SQL: database connection must be established explicitly.

```
EXEC SQL CONNECT :username IDENTIFIED BY :passwd;
```

- `username` and `passwd` host variables of the types `CHAR` or `VARCHAR`..
- strings are not allowed!

Equivalent:

```
EXEC SQL CONNECT :uid;
```

where `uid` is a string of the form `"name/passwd"`.

Host Variables

- Communication between database and application program
- output-variables for communication of values from the database to the application program
- input-variables for the communication of values from the application program to the database.
- assigned to each ost variable: *indicator variable* for handling NULL values.
- to be declared in the *Declare Section*:

```
EXEC SQL BEGIN DECLARE SECTION;
 int population;        /* host variable */
 short population\_ind;   /* indicator variable */
EXEC SQL END DECLARE SECTION;
```

- in SQL-Statements, host variables and indicator variables are prefixed with a colon (":")
- data types if the database and the programming language must be compatible

Database Programming in SQL/ORACLE

Indicator Variables

Handling of Null values

Indicator Variables for Output-Variables:

- -1 : the attribute value is `NULL`, thus, the value of the host variable is undefined.
- 0 : die host variable contains a valid attribute value.
- $>$0 : die host variable contains only a part of the attribute value. The indicator variable gives the original length of the attribute value.
- -2 : the host variable contains only a part of the attribute value, where the original length is not known.

Indicator Variables for Input-Variables:

- -1 : independent from the value of the host variable, the value `NULL` is inserted in the corresponding column.
- $>=$0 : the value of the host variable is inserted in the corresponding column.

Cursors

- Analogous to PL/SQL
- required for processing a resut set that contains more than one tuple

Cursor operations

- `DECLARE <cursor-name> CURSOR FOR <sql statement>`
- `OPEN <cursor-name>`
- `FETCH <cursor-name> INTO <varlist>`
- `CLOSE <cursor-name>`

Error Situations

- cursor has not been declared or not opened
- no (further) data has been found
- cursor has been closed, but not reopened

`Current of` **clause** analogous to PL/SQL

Example

```
int main() {
   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[25];    /* output host var */
      int cityEinw;         /* output host var */
      char* landID = "D";    /* input host var */
      short ind1, ind2;      /* indicator var */
      char* uid = "/";
   EXEC SQL END DECLARE SECTION;
   /* Establish connection to the database */
   EXEC SQL CONNECT :uid;
   /* Cursor declarieren */
   EXEC SQL DECLARE StadtCursor CURSOR FOR
      SELECT Name, Einwohner
      FROM Stadt
      WHERE Code = :landID;
   EXEC SQL OPEN StadtCursor;  /* open cursor */
   printf("Stadt                  Einwohner\n");
   while (1)
   {EXEC SQL FETCH StadtCursor INTO :cityName:ind1 ,
                                   :cityEinw INDICATOR :ind2;
    if(ind1 != -1 && ind2 != -1)
    {  /* keine NULLwerte ausgeben */
       printf("%s     %d \n", cityName, cityEinw);
    }};
   EXEC SQL CLOSE StadtCursor; }
```

Host Arrays

- useful if the size of the result set is known, or only a predefined portion is relevant.
- simplifies the programming, since no cursor is required.
- reduces communication overhead between client and server.

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[25][20];    /* host array */
   int cityPop[20];         /* host array */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT Name, Population
         INTO :cityName, :cityPop
         FROM City
         WHERE Code = 'D';
```

fetches 20 tuples to the two host arrays.

PL/SQL

- Oracle Pro*C/C++ precompiler supports PL/SQL blocks.
- PL/SQL block can be used in place of an SQL statement.
- PL/SQL block reduces communication overhead between client and server.
- Frame for communication:

```
EXEC SQL EXECUTE
DECLARE
  ...
BEGIN
  ...
END;
END-EXEC;
```

Static vs. Dynamic SQL

SQL statements can be composed by string operations. Depending on the statements, there are several commands how to submit these statements to the database.

Transactions

- Application program is regarded as a closed transaction, if it is not divided by `COMMIT`- or `ROLLBACK`-commands
- In Oracle, after leaving a program, `COMMIT` is executed automatically
- DDL statements execute `COMMIT` automatically before being executed themselves
- the database connection is shut down by
  `EXEC SQL COMMIT RELEASE;` or
  `EXEC SQL ROLLBACK RELEASE;`

Savepoints

- Transaction can be divides by *savepoints*.
- Syntax : `EXEC SQL SAVEPOINT <name>`
- `ROLLBACK` to an earlier savepoint deleted all savepoints in-between.

Exception Handling Mechanism

- SQL Communications Area (SQLCA)
- `WHENEVER`-Statement

SQLCA

contains status information about the execution of the most recent SQL statement:

```
struct sqlca {
   char   sqlcaid[8];
   long   sqlcabc;
   long   sqlcode;
   struct { unsigned short sqlerrml;
            char sqlerrmc[70];
   } sqlerrm;
   char   sqlerrp[8];
   long   sqlerrd[6];
   char   sqlwarn[8];
   char   sqlext[8];
};
```

Semantics of `sqlcode`:

- 0: statement has been processes without any problems.
- $>$0: statement has been executed, but a warning occurred.
- $<$0: statement has not been executed due to a serious error message.

`WHENEVER`-Statement

specifies actions that have to be executed automatically by the DBMS in case of an error.

```
EXEC SQL WHENEVER <condition> <action>;
```

`<condition>`

- `SQLWARNING`: the most recent statement caused a warning different from “no data found” (cf. `sqlwarn`). This corresponds to `sqlcode` > 0, but $\neq 1403$.
- `SQLERROR`: the most recent statement caused a serious error. Tis corresponds to `sqlcode` < 0.
- `NOT FOUND`: `SELECT INTO` or `FETCH` did not return any more answer tuple. This corresponds to `sqlcode` 1403.

`<action>`

- `CONTINUE`: the program continues with the subsequent statement.
- `DO flq proc_name>`: invoke a procedure (error handling); `DO break` for exiting a loop.
- `GOTO <label>`: jump to the given label.
- `STOP`: the program is left without `commit` (exit()), a `rollback` is executed.

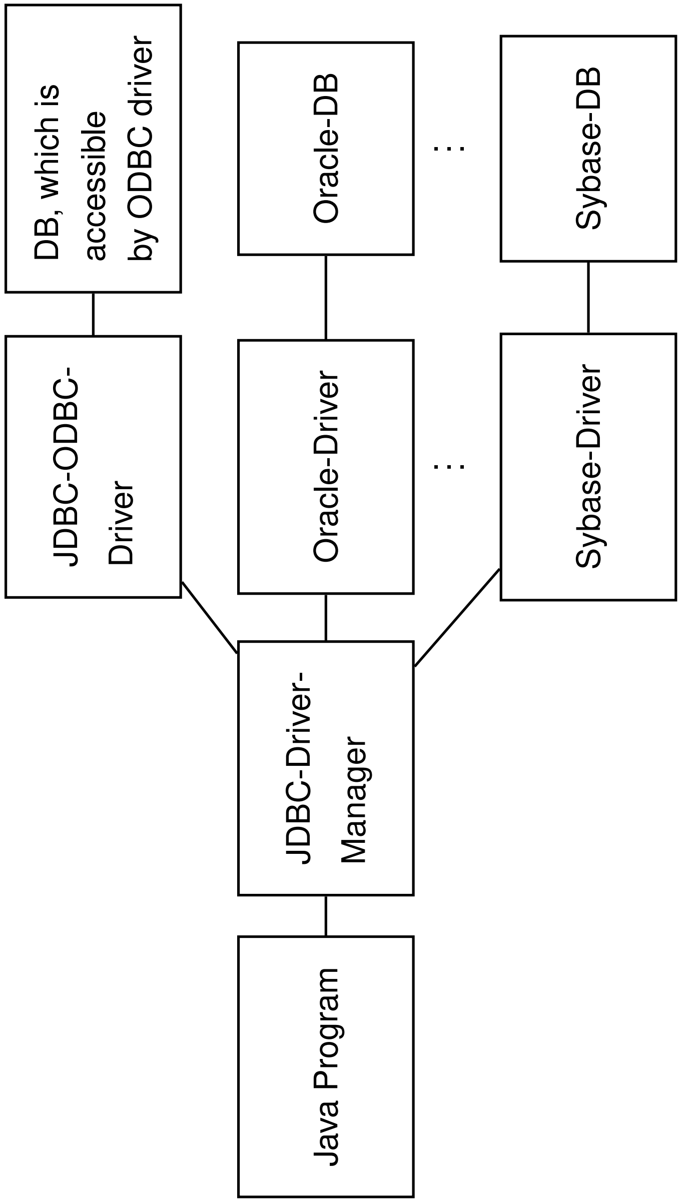
Java and Databases

- Java: platform-independent
- if a *Java Virtual Machine* is available, Java programs can be executed.
- API's: Application Programming Interfaces; collections of classes and interfaces that provide a certain functionality.

JDBC: API for database access (Java DataBase Connectivity)

- interface for (remote) access to a database from Java programs
- application can be programmed independently from the underlying DBMS
- translates the ODBC idea to Java
- common base is the X/Open SQL CLI (Call Level Interface) Standard

JDBC Architecture



JDBC Architecture

- core: driver manager
- below: driver for individual DBMSs

Types of drivers:

- Goal:
  - DBMS-Client-Server-Network-Protocol with pure Java drivers: JDBC-calls are translated to the DBMS-Network protocol. JDBC-client directly calls the DBMS server.
  - JDBC-Net with pure Java driver: JDBC calls are translated to the JDBC-Network protocol. At the server, they are translated into a certain DBMS-Protocol.
- as temporary solution:
  - JDBC-ODBC-Bridge and ODBC-Driver: ODBC driver is used via a JDBC-ODBC-Bridge.
  - Native API: JDBC calls are translated into calls of the client-APIs of the corresponding database vendors.

JDBC-API

- flexible:
  - Application can be programmed independently from the underlying DBMS
  - de facto: portability only in the SQL-2 standard (stored procedures, object-relational features)
- “low-level”:
  - statements are submitted as strings
  - in contrast to Embedded SQL, program variables in SQL commands are not allowed

Under development:

- Embedded SQL for Java
- direct mapping of tables and tuples to Java classes

JDBC-Functionality

- Establishing a connection to the database (`DriverManager, Connection`)
- submission of SQL statements to the database (`Statement` and subclasses)
- processing of the result set (`ResultSet`)

JDBC Driver Manager

`DriverManager`

- registration and administration of drivers
- selects a suitable driver when a connection to some DB is requested
- establishes a connection to the requested DB

- Only one `DriverManager` required.

⇒ class `DriverManager`:
  - only `static` methods (operating on the class)
  - constructor is `private` (impossible to create instances)

Required drivers must be registered:

```
DriverManager.registerDriver(driver*)
```

In the SQL training for the Oracle driver:

```
DriverManager.registerDriver
     (new oracle.jdbc.driver.OracleDriver());
```

creates a new instance of the Oracle driver and “gives” it to the Driver manager.

Database Programming in SQL/ORACLE

Establishing a Connection

- Invocation of the `DriverManager`:

```
Connection <name> =
  DriverManager.getConnection
   (<jdbc-url>, <user-id>, <passwd>);
```

- Database is uniquely identified by the JDBC-URL

JDBC-URL:

- `jdbc:<subprotocol>:<subname>`
- `<subprotocol>` identifies the driver and access mechanism
- `<subname>` identifies the database

SQL training:

```
jdbc:oracle:<driver-name>:
  @<IP-Address DB Server>:<Port>:<SID>
```

```
String url =
  'jdbc:oracle:thin:@132.230.150.11:1521:o901';
Connection conn =
  DriverManager.getConnection(url,'jdbc_1','jdbc_1');
```

returns an opened connection instance `conn`.

Close a connection: `conn.close();`

Submitting SQL Statements

Statement objects:

- are created by invocation of methods of an existing connection <connection>.
- Statement: simple SQL statements without parameters
- PreparedStatement: precompiled queries, queries with parameters
- CallableStatement: invocation of stored procedures (PL/SQL)

Class “Statement”

```
Statement <name> = <connection>.createStatement();
```

Let <string> an SQL statement *without semicolon*.

- ResultSet <statement>.executeQuery(<string>):
  queries against the database. A result set is returned.
- int <statement>.executeUpdate(<string>):
  SQL statements that change the database. The return value indicates how many tuples have been effected.
- <statement>.execute(<string>):
  (sequences of) statements that return more than one result set.
  Result sets are then processed by invoking methods of the statement object (see later).

A statement object can be reused for submitting SQL statements arbitrarily often.

A statement object can be closed by its close() method.

Handling of Result Sets

Class "`ResultSet`":

```
ResultSet <name> = <statement>.executeQuery(<string>);
```

- virtual table that is accessible from the "Host language" – in this case, Java.
- `ResultSet` object maintains a cursor which can be moved by

  `<result-set>.next();`

  to the subsequent tuple.
- `<result-set>.next()`

  returns the value `false` if all tuples have been processed.

```
ResultSet countries =
  stmt.executeQuery(``SELECT Name, Code, Population
                        FROM Country'');
```

Name	**code**	**Population**
Germany	D	83536115
Sweden	S	8900954
Canada	CDN	28820671
Poland	PL	38642565
Bolivia	BOL	7165257
..	..	..

Database Programming in SQL/ORACLE

Handling Result Sets

- access to individual columns of the tuple where the cursor is currently placed by

  `<result-set>.get<type>(<attribute>)`
- where `<type>` is a Java data type,

Java type	get method
INTEGER	getInt
REAL, FLOAT	getFloat
BIT	getBoolean
CHAR, VARCHAR	getString
DATE	getDate
TIME	getTime

  `<getString>` does always work.
- `<attribute>` can be given by the attribute name or the column index.

  ```
  countries.getString(``Code'');
  countries.getInt(``Population'');
  countries.getInt(3);
  ```
- For `get<type>`, the values of the result tuple (SQL-data types) are converted into Java types.

Handling of Result Sets

```
class Hello {
public static void main (String args []) throws SQLException {
  // load the ORACLEdriver
  DriverManager.registerDriver(new oracle.jdbc.dnlddriver.OracleDriver());
  // connect to the database
  String url = "jdbc:oracle:dnldthin:@132.230.150.161:1521:test";
  Connection conn = DriverManager.getConnection(url,:username,:passwd);
  // submit a query to the database
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery("SELECT Name, Population FROM City");
  while (rset.next ()) { // process the result set
   String s = rset.getString(1);
   int i = rset.getInt("Population");
   System.out.println (s + " " + i "\ n");
   }
  }
}
```


Handling of Result Sets

JDBC Data Types

- JDBC stands in-between Java (object types) and SQL (several types).
- java.sql.types defines *generic* SQL types which are used by JDBC:

Java type	JDBC-SQL type
String	CHAR, VARCHAR
java.math.BigDecimal	NUMBER, NUMERIC, DECIMAL
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	FLOAT, DOUBLE
java.sql.Date	DATE (day, month, year)
java.sql.Time	TIME (hour, minute, second)

These are also used for describing metadata.

Handling of Result Sets

Informations about columns of the result set:

```
ResultSetMetaData <name> = <result-set>.getMetaData();
```

creates a `ResultSetMetaData` object that contains information about the result set:

Method	Description
int getColumnCount()	number of columns of the result set
String getColumnLabel(int)	attribute name of the ith column <int>
String getTableName(int)	table name of the ith column <int>
String getSchemaName(int)	schema name of the ith column <int>
int getColumnType(int)	JDBC type of the ith column <int>
String getColumnTypeName(int)	underlying DBMS type of the ith column <int>

Handling of Result Sets

- no `NULL` values in Java:

  `<resultSet>.wasNULL()`

  tests whether the most recently read column value was `NULL`.

Example: output the current row of the result set

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
for(i=1; i<=numCols; i++) {
  String returnValue = rset.getString(i);
  if (rset.wasNull())
   System.out.println ("null");
  else
   System.out.println (returnValue);
}
```

- The method `close()` closes a ResultSet object explicitly.

Prepared Statements

```
PreparedStatement <name> =
  <connection>.prepareStatement(<string>);
```

- SQL statement `<string>` is precompiled.
- thus, the statement is contained in the state of the object
- more efficient than `Statement` if some statement has to be executed several times.
- depending on `<string>`, only one of the (parameterless!) methods
  - `<prepared-statement>.executeQuery()`,
  - `<prepared-statement>.executeUpdate()` or
  - `<prepared-statement>.execute()`

  is applicable.

Prepared Statements: Parameters

- Input parameters are represented by “?”

```
PreparedStatement pstmt =
  conn.prepareStatement("SELECT Population
                         FROM Country
                         WHERE Code = ?");
```

- “?”-parameters are assigned to values by

  `<prepared-statement>.set<type>(<pos>,<value>);`

  before a `PreparedStatement` is submitted.
- `<type>`: Java data type,
- `<pos>`: position of the parameter to be set,
- `<value>`: value.

```
pstmt.setString(1,"D");
ResultSet rset = pstmt.ExecuteQuery();
...
pstmt.setString(1,"CH");
ResultSet rset = pstmt.ExecuteQuery();
...
```

- Null values are set by

  `setNULL(<pos>,<type>);`

  where `<type>` is the JDBC type of this column:

  `pstmt.setNULL(1,Types.String);`

Callable Statements: Invoke Stored Procedures

- Stored procedures and functions are created by

  ```
  <statement>.executeUpdate(<string>);
  ```

  (<string> is of the form `CREATE PROCEDURE ...`)

  ```
  s = 'CREATE PROCEDURE bla() IS BEGIN ... END';
  stmt.executeUpdate(s);
  ```

- the *procedure invocation* is then created as a `CallableStatement` object:
- invocation syntax of procedures differs amongst the DBMS products

⇒ JDBC uses a *generic* syntax via an escape-sequence (which is translated by the driver)

```
CallableStatement <name> =
  <connection>.prepareCall("{call <procedure>}");
cstmt = conn.prepareCall("{call bla()}");
```


Database Programming in SQL/ORACLE

Callable Statements with Parameters

```
s = 'CREATE FUNCTION
            distance(city1 IN Name, city2 IN Name)
     RETURN NUMBER IS BEGIN ... END';
stmt.executeUpdate(s);
```

- Parameters:

```
CallableStatement <name> =
  <connection>.prepareCall("{call <procedure>(?,...,?)}");
```

- Return value of functions:

  ```
  CallableStatement <name> =
    <connection>.prepareCall
        ("{? = call <procedure>(?,...,?)}");
  cstmt = conn.prepareCall("{? = call distance(?,?)}");
  ```

- for `OUT`-parameters and the return value, the JDBC data type of the parameters must first be registered by

  ```
  <callable-statement>.registerOutParameter
        (<pos>,java.sql.Types.<type>);
  cstmt.registerOutParameter(1,java.sql.types.number);
  ```

Callable Statements with Parameters

- Preparations (see above)

```
cstmt = conn.prepareCall("{? = call distance(?,?)}");
cstmt.registerOutParameter(1,java.sql.types.number);
```

- `IN` parameters are set by `set<type>`:

```
cstmt.setString(2,'Freiburg');
cstmt.setString(3,'Berlin');
```

- invocation by

```
ResultSet <name> =
  <callable-statement>.executeQuery();
```

or

```
<callable-statement>.executeUpdate();
```

or

```
<callable-statement>.execute();
```

```
in our example:    cstmt.execute();
```

- `OUT`-parameters are read by `get<type>`:

```
int distance = cstmt.getInt(1);
```


Database Programming in SQL/ORACLE

Sequential Execution

- SQL-Statements that return a sequence of result sets:
- `<statement>.execute(<string>)`, `<prepared-statement>.execute()`, `<callable-statement>.execute()`
- Often `<string>` is generated dynamically
- `getResultSet()` or `getUpdateCount()`: gets the next return value or update count.
- `getMoreResults()` and then again `getResultSet()` or `getUpdateCount()`: proceed to the next result.

Sequential Execution

- `getResultSet()`: if the next result is a result set, this is returned. If no next result is available, or the next result is an update count, `null` is returned.
- `getUpdateCount()`: if the next result is an update count, this ($n \geq 0$) is returned. If no next result is available, or the next result is a result set, -1 is returned.
- `getMoreResults()`: `true`, if the next result is a result set, `false`, if it is an update count or there are no more results.
- test if all results are processed:

  ```
  ((<stmt>.getResultSet() == null) &&
   (<stmt>.getUpdateCount() == -1))
  ```

  or

  ```
  ((<stmt>.getMoreResults() == false) &&
   (<stmt>.getUpdateCount() == -1))
  ```


Database Programming in SQL/ORACLE

Handling a Sequence of Results

```
stmt.execute(queryStringWithUnknownResults);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) {
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue;
    }
    if (rowCount == 0) {
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue;
    }
    ResultSet rs = stmt.getResultSet();
    if (rs != null) {
       .....  // process metadata
       while (rs.next())
          { ....}  // process result set
       stmt.getMoreResults();
       continue;
    }
    break;
}
```

Further SQL/Oracle Tools

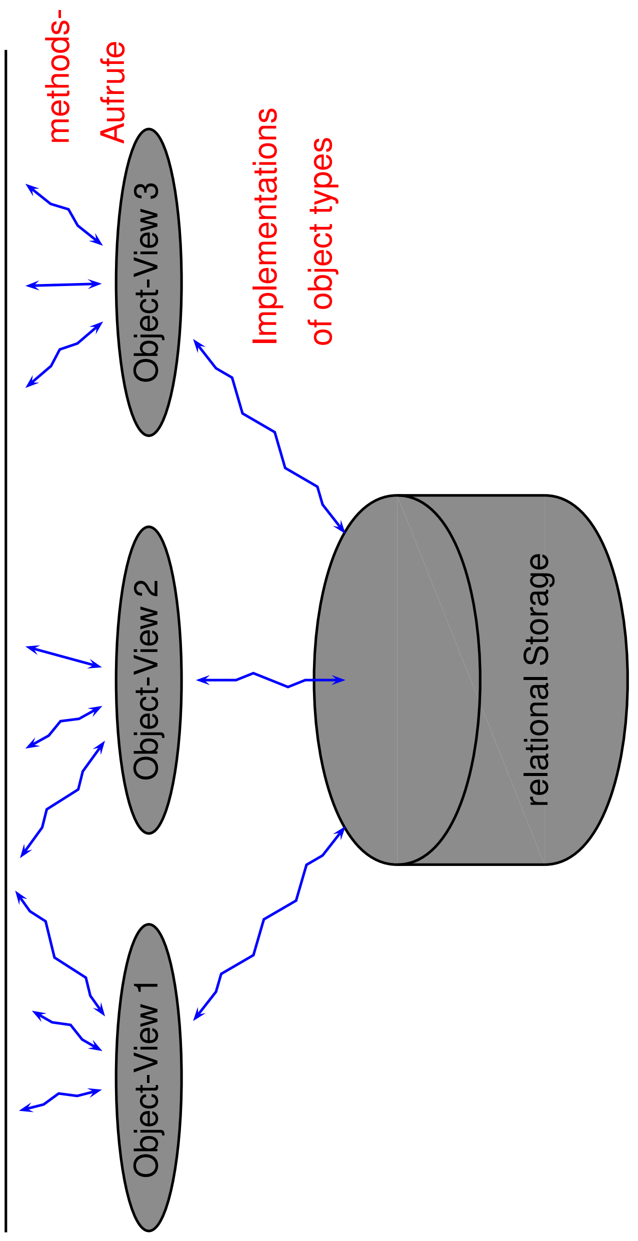
- Dynamic SQL: SQL statements are generated in *in PL-SQL* at runtime as strings, and are then submitted to the database.
- ORACLE8i: built-in Java Virtual Machine, access to the file system,
  i= internet: XML-interface, Web-Application-Server etc.
- ORACLE-Web Server/Internet Application Server (9i): HTML pages can be generated depending on the database contents.
- by the most recent packages and extensions (IAS, Internet File System Server) the difference between the database and the operating system diminishes.

ORACLE8 ?

+ complex data types
+ Objects: object methods, object references, path expressions
  ⇒ user-friendly interface possible
  (vgl. *add_member*, *is_member*)
- Nested Tables:
  - Storage: as separate tables (`STORE AS ...`)
  - DML: cumbersome `SELECT FROM THE`, `TABLE ...`, `CAST MULTISET`
  - usage: query must only consider a single nested table
    ⇒ cursor requires
  - no advantages ??
- modifications of object types not supported
  ⇒ object types not suitable for *storage*.

- *"I think this is the power of the system. Object Views."*

Database-Architecture



- Modifications of the relational storage: easy. Implementations of the object types can be adapted without changing the user interface (external schema).
- Modifications of the object types: independent of the storage (Views). Possible to delete object types completely and rebuild new ones without losing data.
- Adding functionality: redefine or add suitable object types.