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 Universität Karlsruhe

... some more Web-Pages

Data Integration has been done with FLORID

Literature

- Textbooks on Databases (in german):
  A. Kemper, A. Eickler: Datenbanksysteme - Eine Einführung, Oldenbourg, 1996

- Textbook on SQL (in german):

- The book on the practical DB training at Uni Karlsruhe with TERRA:

- Explanation of the SQL-2 Standard:

- Textbooks on relational databases and SQL:

and some more ...
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).
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,
- \(\{A_1, \ldots, A_n\}\), \(n \geq 0\) 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.

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 \(RO\).
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.

Generalization/Specialization

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

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

```
name -- Geo <1,*> in <0,*> Province
```

River Lake Sea Mountain Island Desert

Mapping ERM to RM

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

1. Entity types: $(E_{ER}, \{A_1, \ldots, A_n\}) \rightarrow E(A_1, \ldots, A_n)$,

2. Relationship types:
   $B_{ER}(\{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \rightarrow B(E_1.K_{11}, \ldots, E_1.K_{1p}, \ldots, E_k.K_{k1}, \ldots, E_k.K_{kp}, A_1, \ldots, A_m)$,
   where $\{K_{i1}, \ldots, K_{ip}\}$ 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 identifyi ng entity type must be added.

4. Aggregate types can be ignored if the underlying relationship type is mapped.

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,

- each attribute is associated with a Domain which specifies the possible values of the attribute. Often, attributes also can have a null value.

- elements of a relation are called tuples.

A (relational) database schema $R$ is given by a (finite) set of (relation) schemata.

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

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asia</td>
<td>4.5E7</td>
</tr>
<tr>
<td>Africa</td>
<td>3.02547e+07</td>
</tr>
<tr>
<td>Europe</td>
<td>9562489.6</td>
</tr>
<tr>
<td>America</td>
<td>3.9872e+07</td>
</tr>
<tr>
<td>Australia</td>
<td>8503474.56</td>
</tr>
</tbody>
</table>
```

Entity types $(E_{ER}, \{A_1, \ldots, A_n\}) \rightarrow E(A_1, \ldots, A_n)$
Relationship Types

\[(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \rightarrow B(E_1_{K_11}, \ldots, E_1_{K_1p_1}, \ldots, E_k_{K_k1}, \ldots, E_k_{K_kp_k}, A_1, \ldots, A_m)\]

where \(\{K_i_{11}, \ldots, K_i_{p_i}\}\) 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)

Weak Entity Types

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

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>area</th>
<th>pop.</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>D</td>
<td>24867</td>
<td>81170500</td>
<td>D</td>
</tr>
<tr>
<td>Stockholm</td>
<td>S</td>
<td>3575</td>
<td>14272069</td>
<td>S</td>
</tr>
</tbody>
</table>

The Relational Model

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freiburg</td>
<td>D</td>
</tr>
<tr>
<td>Berlin</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country1</td>
<td>Country2</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>D</td>
<td>CH</td>
</tr>
</tbody>
</table>

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.

Weak Entity Types

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

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.
SQL = Structured Query Language

- common query language
- 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' ≠ City='berlin'.
- every command has to be ended with a semicolon ";"
- comment lines are embraced in /∗. . . ∗/, or introduced by -- or rem.

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>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_ARGUMENTS</td>
<td>Arguments in objects accessible to the user</td>
</tr>
<tr>
<td>ALL_CATALOG</td>
<td>All tables, views, synonyms, sequences accessible to the user</td>
</tr>
<tr>
<td>ALL_CLUSTERS</td>
<td>Description of clusters accessible to the user</td>
</tr>
<tr>
<td>ALL_CLUSTER_HASH_EXPRESSIONS</td>
<td>Hash functions for all accessible clusters</td>
</tr>
</tbody>
</table>

Data Dictionary

Contains meta data about the database

Database Language:

**DDL**: Data Definition Language for defining schema
- tables
- views
- indexes
- integrity constraints

**DML**: Data Manipulation Language for manipulating database states
- Search/Read
- Insert
- Modify
- Delete

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>
<thead>
<tr>
<th>Table_name</th>
<th>Table_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BORDERS</td>
<td>ISLAND</td>
</tr>
<tr>
<td>CITY</td>
<td>LAKE</td>
</tr>
<tr>
<td>CONTINENT</td>
<td>LANGUAGE</td>
</tr>
<tr>
<td>COUNTRY</td>
<td>LOCATED</td>
</tr>
<tr>
<td>DESERT</td>
<td>IS_MEMBER</td>
</tr>
<tr>
<td>ECONOMY</td>
<td>MERGES_WITH</td>
</tr>
<tr>
<td>ENCOMPASSES</td>
<td>MOUNTAIN</td>
</tr>
<tr>
<td>ETHNIC_GROUP</td>
<td>ORGANIZATION</td>
</tr>
<tr>
<td>GEO_DESERT</td>
<td>POLITICS</td>
</tr>
<tr>
<td>GEO_ISLAND</td>
<td>POPULATION</td>
</tr>
<tr>
<td>GEO_LAKE</td>
<td>PROVINCE</td>
</tr>
<tr>
<td>GEO_MOUNTA IN</td>
<td>RELIGION</td>
</tr>
<tr>
<td>GEO_RIVER</td>
<td>RIVER</td>
</tr>
<tr>
<td>GEO_SEA</td>
<td>SEA</td>
</tr>
</tbody>
</table>

28 rows selected.

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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>C.</th>
<th>Province</th>
<th>Pop.</th>
<th>Long.</th>
<th>Lat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vienna</td>
<td>A</td>
<td>Vienna</td>
<td>1583000</td>
<td>16,3667</td>
<td>48.25</td>
</tr>
<tr>
<td>Innsbruck</td>
<td>A</td>
<td>Tyrol</td>
<td>118000</td>
<td>11,22</td>
<td>47.17</td>
</tr>
<tr>
<td>Stuttgart</td>
<td>D</td>
<td>Baden-W.</td>
<td>588482</td>
<td>9.1</td>
<td>48.7</td>
</tr>
<tr>
<td>Freiburg</td>
<td>D</td>
<td>Germany</td>
<td>198496</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokyo</td>
<td>J</td>
</tr>
<tr>
<td>Stockholm</td>
<td>S</td>
</tr>
<tr>
<td>Warsaw</td>
<td>PL</td>
</tr>
<tr>
<td>Cochabamba</td>
<td>BOL</td>
</tr>
<tr>
<td>Hamburg</td>
<td>D</td>
</tr>
<tr>
<td>Berlin</td>
<td>D</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data Dictionary

1. **Table**: A named collection of logically related data.
2. **Column**: A named piece of data stored in a table.
3. **Row**: A set of related columns.
4. **Value**: An item of data stored in a column.

**Key Terms**

- **Attribute**: A characteristic or feature of an entity.
- **Entity**: A class or collection of objects that share the same characteristics.
- **Relationship**: A connection between entities.
- **Entity Set**: A collection of entities.

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

```
DESC City;
```

```
<table>
<thead>
<tr>
<th>Name</th>
<th>NULL?</th>
<th>Typ</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>COUNTRY</td>
<td>NOT NULL</td>
<td>VARCHAR2(4)</td>
</tr>
<tr>
<td>PROVINCE</td>
<td>NOT NULL</td>
<td>VARCHAR2(35)</td>
</tr>
<tr>
<td>POPULATION</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LONGITUDE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LATITUDE</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

SQL: Queries
**DISTINCT**

\[
\text{SELECT * FROM Island;}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Islands</th>
<th>Area ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jersey</td>
<td>Channel Islands</td>
<td>:</td>
</tr>
<tr>
<td>Mull</td>
<td>Inner Hebrides</td>
<td>:</td>
</tr>
<tr>
<td>Montserrat</td>
<td>Antilles</td>
<td>:</td>
</tr>
<tr>
<td>Grenada</td>
<td>Antilles</td>
<td>:</td>
</tr>
</tbody>
</table>

\[
\text{SELECT Islands FROM Island;}
\]

**Selections: Choose some rows**

\[
\text{SELECT <attr-list> FROM <table> WHERE <predicate>;}
\]

<p><tt><var><operator><value><operator><value></operator></value></operator><value><operator><operator><value></operator></value></operator><operator><value><operator><value><operator><value></operator></value></operator></value></operator></operator></var></tt></p>

- <var><operator><value><operator><value></operator></value></operator><value><operator><operator><value></operator></value></operator></operator></var>
- <var><operator><value><operator><operator><value></operator></value></operator><value><operator><operator><value></operator></value></operator></operator></operator></var>
- <var><operator><value><operator><value><operator><value></operator></value></operator></operator></operator></var>
- <var><operator><value><operator><value><operator><value></operator></value></operator></operator></operator></var>
- <var><operator><value><operator><value><operator><value></operator></value></operator></operator></operator></var>
- <var><operator><value><operator><value><operator><value></operator></value></operator></operator></operator></var>

**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

**Example:**

\[
\text{SELECT Name, Country, Population FROM City WHERE Country = 'J';}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokyo</td>
<td>J</td>
<td>7843000</td>
</tr>
<tr>
<td>Kyoto</td>
<td>J</td>
<td>1415000</td>
</tr>
<tr>
<td>Hiroshima</td>
<td>J</td>
<td>1099000</td>
</tr>
<tr>
<td>Yokohama</td>
<td>J</td>
<td>3256000</td>
</tr>
<tr>
<td>Sapporo</td>
<td>J</td>
<td>1748000</td>
</tr>
</tbody>
</table>

**Example:**

\[
\text{SELECT Name, Country, Population FROM City WHERE Country = 'J' AND Population > 2000000}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokyo</td>
<td>J</td>
<td>7843000</td>
</tr>
<tr>
<td>Yokohama</td>
<td>J</td>
<td>3256000</td>
</tr>
</tbody>
</table>
**Example:**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kingston</td>
<td>JA</td>
<td>101000</td>
</tr>
<tr>
<td>Amman</td>
<td>JOR</td>
<td>777500</td>
</tr>
<tr>
<td>Suva</td>
<td>FJI</td>
<td>69481</td>
</tr>
</tbody>
</table>

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

**ORDER BY, Alias**

```sql
SELECT Name, Population/Area AS Density
FROM Country
ORDER BY 2 ;
```

### Aggregate functions

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

**Example:** How many cities are stored in the database?

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

<table>
<thead>
<tr>
<th>Count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3114</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Count(DISTINCT(Country))</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
</tr>
</tbody>
</table>
Aggregate functions

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

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

<table>
<thead>
<tr>
<th>SUM(Population)</th>
<th>MAX(Population)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2434525</td>
<td>1583000</td>
</tr>
</tbody>
</table>

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

Grouping

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

```sql
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).

Conditions over Groups

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

```sql
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.
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;
```

<table>
<thead>
<tr>
<th>Country</th>
<th>SUM(Population)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUS</td>
<td>12153500</td>
</tr>
<tr>
<td>BR</td>
<td>77092190</td>
</tr>
<tr>
<td>CDN</td>
<td>10791230</td>
</tr>
<tr>
<td>CO</td>
<td>18153631</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Set Operations

SQL queries can be joined by set operations:

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

- `UNION [ALL]`
- `MINUS [ALL]`
- `INTERSECT [ALL]`
- automatical elimination 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);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Einwohner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>3249136</td>
</tr>
<tr>
<td>Andorra</td>
<td>72766</td>
</tr>
<tr>
<td>Liechtenstein</td>
<td>31122</td>
</tr>
<tr>
<td>Slovakia</td>
<td>5374362</td>
</tr>
<tr>
<td>Slovenia</td>
<td>1951443</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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'
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Einwohner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>3249136</td>
</tr>
<tr>
<td>Andorra</td>
<td>72766</td>
</tr>
<tr>
<td>Liechtenstein</td>
<td>31122</td>
</tr>
<tr>
<td>Slovakia</td>
<td>5374362</td>
</tr>
<tr>
<td>Slovenia</td>
<td>1951443</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Equijoin

Example: For all organizations, give the continents where they are seated.

encompasses: Country, Continent, Percentage.
Organization: Abbreviation, Name, City, Country, Province.

SELECT Continent, Abbreviation
FROM encompasses, Organization

<table>
<thead>
<tr>
<th>Name</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>America</td>
<td>UN</td>
</tr>
<tr>
<td>Europe</td>
<td>UNESCO</td>
</tr>
<tr>
<td>Europe</td>
<td>CCC</td>
</tr>
<tr>
<td>Europe</td>
<td>EU</td>
</tr>
<tr>
<td>America</td>
<td>CACM</td>
</tr>
<tr>
<td>Australia/Oceania</td>
<td>ANZUS</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>A.Name</th>
<th>A.Country</th>
<th>B.Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexandria</td>
<td>ET</td>
<td>RO</td>
</tr>
<tr>
<td>Alexandria</td>
<td>ET</td>
<td>USA</td>
</tr>
<tr>
<td>Alexandria</td>
<td>RO</td>
<td>USA</td>
</tr>
<tr>
<td>Barcelona</td>
<td>E</td>
<td>YV</td>
</tr>
<tr>
<td>Valencia</td>
<td>E</td>
<td>YV</td>
</tr>
<tr>
<td>Salamanca</td>
<td>E</td>
<td>MEX</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

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> ∈ {<, >, <, <=}, <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');

<table>
<thead>
<tr>
<th>Country.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
</tr>
<tr>
<td>Seychelles</td>
</tr>
</tbody>
</table>
Uncorrelated Subquery with \textit{IN}

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

\begin{verbatim}
SELECT *
FROM CITY
WHERE (Name,Country,Province)
    IN (SELECT City,Country,Province
         FROM located);
\end{verbatim}

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Province</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajaccio</td>
<td>F</td>
<td>Corse</td>
<td>53500</td>
</tr>
<tr>
<td>Karlstad</td>
<td>S</td>
<td>Värmland</td>
<td>74669</td>
</tr>
<tr>
<td>San Diego</td>
<td>USA</td>
<td>California</td>
<td>1171121</td>
</tr>
</tbody>
</table>

Subqueries 52

Subquery with \textit{ALL}

\textbf{Example:} \textit{ALL} can e.g. be used for computing all countries that are smaller than all countries that have more than 10 million inhabitants:

\begin{verbatim}
SELECT Name,Area,Population
FROM Country
WHERE Area < ALL
    (SELECT Area
     FROM Country
     WHERE Population > 10000000);
\end{verbatim}

<table>
<thead>
<tr>
<th>Name</th>
<th>Area</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>28750</td>
<td>3249136</td>
</tr>
<tr>
<td>Macedonia</td>
<td>25333</td>
<td>2104035</td>
</tr>
<tr>
<td>Andorra</td>
<td>450</td>
<td>72766</td>
</tr>
</tbody>
</table>

Subqueries 53

The \textit{EXISTS} Operator

\textit{EXISTS} and \textit{NOT EXISTS} simulate the existential quantifier.

\begin{verbatim}
SELECT <attr-list>
FROM <table>
WHERE [NOT] EXISTS
    (<select-clause>);
\end{verbatim}

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

\begin{verbatim}
SELECT Name
FROM Country
WHERE EXISTS
    ( SELECT *
        FROM City
        WHERE Population > 1000000
        AND City.Country = Country.Code);
\end{verbatim}

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serbia and Montenegro</td>
</tr>
<tr>
<td>France</td>
</tr>
<tr>
<td>Spain</td>
</tr>
<tr>
<td>Austria</td>
</tr>
</tbody>
</table>

Subqueries 54

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,
- \textit{imported} attributes must be qualified.

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

\begin{verbatim}
SELECT Name, Country
FROM City
WHERE Population * 4 >
    (SELECT Population
     FROM Country
     WHERE Code = City.Country);
\end{verbatim}

Subqueries 55
**Transformation EXISTS, Subquery, Join**

Equivalent to the previous one are the following queries:

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

```sql
SELECT DISTINCT Country.Name
FROM Country, City
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?

```sql
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
GROUP BY Country.Name, Country.Population
HAVING (SUM(City.Population)/Country.Population) > 0.1;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Population-Urban_Residents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Austria</td>
<td>4620065771</td>
</tr>
<tr>
<td>Denmark</td>
<td></td>
</tr>
<tr>
<td>Germany</td>
<td></td>
</tr>
<tr>
<td>Ireland</td>
<td></td>
</tr>
<tr>
<td>Italy</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td></td>
</tr>
<tr>
<td>Spain</td>
<td></td>
</tr>
<tr>
<td>United Kingdom</td>
<td></td>
</tr>
</tbody>
</table>

**Subqueries in the FROM Clause**

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.

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

| Population-Urban_Residents | 4620065771 |

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

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

| sum(pop_biggest) | 273837106 |

... especially suitable for nested computations with aggregate functions

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

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

| sum(pop_biggest) | 273837106 |
**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 ≤ n.

||: string concatenation.

**NUMBER:** numbers. for NUMBER, the usual operators +, −, *, and /, and the comparisons =, >, >=, <, and < are allowed. In addition there is BETWEEN x AND y.

**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.

**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: 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');

<table>
<thead>
<tr>
<th>Country</th>
<th>Organization</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CZ</td>
<td>EU</td>
<td>membership applicant</td>
</tr>
<tr>
<td>D</td>
<td>EU</td>
<td>member</td>
</tr>
</tbody>
</table>

Definition of Tables: Constraints (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|ON DELETE SET NULL]: 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
  
  ```sql
  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.

<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>b</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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:

```sql
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|ON DELETE SET NULL]: 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|ON DELETE SET NULL: referential action (later).

```sql
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:

```sql
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.

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>);
```

• 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;
```

Inserting Information

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';
```

```
<table>
<thead>
<tr>
<th>Land</th>
<th>Datum</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>01 01 1419</td>
</tr>
<tr>
<td>NL</td>
<td>01 01 1579</td>
</tr>
<tr>
<td>E</td>
<td>01 01 1492</td>
</tr>
<tr>
<td>THA</td>
<td>01 01 1238</td>
</tr>
</tbody>
</table>
```

```
```

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 the month to which d belongs.
- MONTHS_BETWEEN(d1, d2) returns the number of months between two dates.
Object Orientation in ORACLE 8

- complex data types:

```
CREATE TYPE GeoCoord AS OBJECT
  (Longitude NUMBER,
   Latitude NUMBER);
/
CREATE TABLE Mountain
  (Name VARCHAR2(20),
   Height NUMBER,
   Coordinates GeoCoord);
```

- nested tables:

```
CREATE TYPE <type> AS OBJECT (...)
```

- objects, methods, object tables, object references ...

(later)

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;
```

Complex Data Types

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

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)
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;

SELECT *
FROM NLanguage
WHERE Country='CH';

<table>
<thead>
<tr>
<th>Country</th>
<th>Languages(Name, Percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))</td>
</tr>
</tbody>
</table>

SELECT Languages
FROM NLanguage
WHERE Country='CH';

<table>
<thead>
<tr>
<th>Languages(Name, Percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))</td>
</tr>
</tbody>
</table>

Again: constructor methods

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

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

```sql
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 \((n = \text{number of languages in this country})\) !!

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

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

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:

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

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.

<table>
<thead>
<tr>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>CH</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>NAM</td>
</tr>
</tbody>
</table>
Working with Nested Tables

CURSOR-Operator:

Example:

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

| Country | CURSOR(SELECT...)
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>CURSOR STATEMENT : 2</td>
</tr>
<tr>
<td>Name</td>
<td>Percentage</td>
</tr>
<tr>
<td>French</td>
<td>18</td>
</tr>
<tr>
<td>German</td>
<td>65</td>
</tr>
<tr>
<td>Italian</td>
<td>12</td>
</tr>
<tr>
<td>Romansch</td>
<td>1</td>
</tr>
</tbody>
</table>

⇒ Cursors etc. in PL/SQL.

Transactions in ORACLE

Begin of a Transaction

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

Safepoints

For a long transaction, savepoints can be set:

```sql
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.

Complex Data Types

```sql
SELECT * FROM USER_TYPES
```

<table>
<thead>
<tr>
<th>Type_name</th>
<th>Type_oid</th>
<th>Typecode</th>
<th>Attributes</th>
<th>Methods</th>
<th>Pre Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoCoord</td>
<td>-</td>
<td>0</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Language_T</td>
<td>-</td>
<td>0</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Languages_List</td>
<td>0 Collection</td>
<td>0</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

Delete:

```sql
DROP TYPE [FORCE]
```

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

Same scenario:

```sql
DROP TYPE Language_T
```

SQL> desc Languages_List;FEHLER:ORA-24372: Ungültiges Objekt für Beschreibung

Nested Tables

| Type_name | Type_oid | Typecode | Attributes
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages_T</td>
<td>2</td>
<td>Object</td>
<td>0</td>
</tr>
<tr>
<td>Languages_List</td>
<td>0 Collection</td>
<td>0</td>
<td>NO</td>
</tr>
</tbody>
</table>

Using a table All_Languages that contains all languages:

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

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

Transactions

Nested Tables

Nested Tables

Nested Tables
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.

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.

```sql
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>;
```
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));

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>).

ADD (PRIMARY KEY (Country, Continent));
ADD (CONSTRAINT DesertArea CHECK (Area > 10));

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

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));
Referential Integrity

Referential integrity occurs 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);

• as column condition:
```
<attr> [CONSTRAINT <name>] REFERENCES <table>(<attr>)
```

CREATE TABLE Country
VALUES ('Lummerland','LU',...);

DELETE FROM is_member ('D','EU');

• as table condition:
```
[CONSTRAINT <name>]
FOREIGN KEY (<attr-list>) REFERENCES <table>(<attr-list>)
```

CREATE TABLE Country
VALUES ('Karl-Marx-Stadt','DDR',...);

If the target key exists, there is no problem:
UPDATE City SET Country='A' WHERE Name='Munich';

Foreign key values that do not exist in the referenced table:
UPDATE Country SET Code='UK' WHERE Code='GB';
or
DELETE FROM Country WHERE Code='I';

Referential Integrity

Referential Actions

• if the contents of a table changes, actions are carried out for preserving referential integrity.

1. INSERT into a referenced table or DELETE from a referencing table does not cause any problems:
```
INSERT INTO Country
VALUES ('Lummerland','LU',...);
```

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 and 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,...) ~
(United Kingdom,UK,...)
Province:(Yorkshire,GB,...) ~ (Yorkshire,UK,...)
City: (London,GB,Greater London,...) ~
(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,NULL,North Sea)
DELETE * FROM River WHERE Name='Weser';
located(Bremerhaven,Nds.,D,NULL,NULL,North Sea)

Referential Actions

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>Code</th>
<th>Capital</th>
<th>Province</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>D</td>
<td>Berlin</td>
<td>Berlin</td>
<td></td>
</tr>
<tr>
<td>United States</td>
<td>USA</td>
<td>Washington</td>
<td>Distr. Columbia</td>
<td></td>
</tr>
</tbody>
</table>

Referential Actions

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,...) ~
(United Kingdom,UK,...)
Province:(Yorkshire,GB,...) ~ (Yorkshire,UK,...)
City: (London,GB,Greater London,...) ~
(London,UK,Greater London,...)

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Province</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>D</td>
<td>Berlin</td>
</tr>
<tr>
<td>Washington</td>
<td>USA</td>
<td>Distr. Columbia</td>
</tr>
</tbody>
</table>

1. DELETE FROM City WHERE Name='Berlin';
2. DELETE FROM Country WHERE Name='Germany';
Referential Actions in ORACLE

- ORACLE 9: only ON DELETE/UPDATE NO ACTION, ON DELETE CASCADE, and ON DELETE SET NULL 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|ON DELETE SET NULL]

Syntax as <tableConstraint>:
CONSTRAINT <name>
    FOREIGN KEY [ (<attr-list>)]
    REFERENCES <table> (<attr-list>)
    [ON DELETE CASCADE|ON DELETE SET NULL]

Referential Actions: UPDATE without CASCADE

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

ALTER TABLE Province
    DISABLE CONSTRAINT ProvRefsCountry;
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>],
    ...
    [<tableConstraint>],
    ...
DISABLE ...
    ...
ALTER TABLE <table>
    ENABLE ...;

Referential Integrity

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';
  ~ “dangling reference” of the old tuple ('Yorkshire','GB').
- UPDATE Province SET Code='UK' WHERE Code='GB';
  ~ “dangling reference” of the new tuple ('Yorkshire','UK').
Referential Actions: Cyclic References

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>Code</th>
<th>Capital</th>
<th>Province</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td></td>
<td>D</td>
<td>Berlin</td>
<td>Berlin</td>
</tr>
</tbody>
</table>

Referential Actions: Problems with ON UPDATE

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>Code</th>
<th>Capital</th>
<th>Province</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td></td>
<td>D</td>
<td>Berlin</td>
<td>Berlin</td>
</tr>
</tbody>
</table>

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:

```sql
CREATE VIEW <name> AS ...
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = '<NAME>';
```
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>
<thead>
<tr>
<th>Table_Name</th>
<th>Column_Name</th>
<th>UPD</th>
<th>INS</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>Name</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>temp</td>
<td>Code</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>temp</td>
<td>Area</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>temp</td>
<td>Population</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>temp</td>
<td>Density</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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 120

View Updates

Example:

CREATE VIEW CityCountry (City, Country) AS
SELECT City.Name, Country.Name
FROM City, Country

SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = 'CITYCOUNTRY';

<table>
<thead>
<tr>
<th>Table_Name</th>
<th>Column_Name</th>
<th>UPD</th>
<th>INS</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CityCountry</td>
<td>City</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>CityCountry</td>
<td>Country</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- city names can be changed:

  direct mapping to City:

  UPDATE CityCountry
  SET Country = 'Poland'
  WHERE City = 'Berlin';

  DELETE FROM CityCountry
  WHERE City = 'Berlin';

View Updates 121

View Updates

Example:

CREATE VIEW CityCountry (City, Country) AS
SELECT City.Name, Country.Name
FROM City, Country

SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = 'CITYCOUNTRY';

<table>
<thead>
<tr>
<th>Table_Name</th>
<th>Column_Name</th>
<th>UPD</th>
<th>INS</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CityCountry</td>
<td>City</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>CityCountry</td>
<td>Country</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- City names can be changed:

  direct mapping to City:

  UPDATE CityCountry
  SET Country = 'Deutschland'
  WHERE City = 'Berlin';

  DELETE FROM CityCountry
  WHERE City = 'Berlin';

Mapping to base table would be ambiguous:

UPDATE CityCountry
SET Country = 'Poland'
WHERE City = 'Berlin';

DELETE FROM CityCountry
WHERE City = 'Berlin';

View Updates 122

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
View Updates

**Example:**

```sql
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';
```

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Nordrhein-Westfalen</td>
<td>17816079</td>
</tr>
</tbody>
</table>

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!

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

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Bayern</td>
<td>11921944</td>
</tr>
</tbody>
</table>

**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 Authentication**

- 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>`.
System Privileges

- Entitle for schema operations
- CREATE [ANY]
  TABLE/VIEW/TABLE/INDEX/CLUSTER/TRIGGER/PROCEDURE: user is allowed to generate schema objects of these types,
- ALTER [ANY] TABLE/TABLE/INDEX/CLUSTER/TRIGGER/PROCEDURE: user is allowed to change schema objects of these types,
- DROP [ANY]
  TABLE/VIEW/TABLE/INDEX/CLUSTER/TRIGGER/PROCEDURE: user is allowed to delete schema objects of these types,
- SELECT/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
- CREATE INDEX, CREATE TRIGGER, CREATE PROCEDURE
- CREATE USER, CREATE ROLE, CREATE SESSION
- CREATE TERMINAL

Permissions for accessing and changing the own tables are not mentioned explicitly (SELECT TABLE).

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,
  EXECUTE for procedures, functions, and TYPES.
- ALL: all privileges that one has for the corresponding object.
- GRANT OPTION: the grantee can grant the permission to other users.

Revoke permissions:

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

User roles are defined as prototypical patterns for maintaining permissions (e.g., student, dba, ...).
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 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
  - access paths: indexes, hashing
- try to access data that semantically belongs together with a single access to secondary memory
  - 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.

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>58317450</td>
<td>83536115</td>
<td>15568034</td>
</tr>
<tr>
<td></td>
<td>547030</td>
<td>356910</td>
<td>37330</td>
</tr>
</tbody>
</table>

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:
• minimize 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.

Optimization 136
Optimization 138
Optimization 137
Optimization 139
Clustering

Sea and geo_Sea with cluster key Sea.Name:

<table>
<thead>
<tr>
<th>Cl_Sea</th>
<th>Depth</th>
<th>Province</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mediterranean Sea</td>
<td>5121</td>
<td>Catalonia</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valencia</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Murcia</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Andalusia</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Languedoc-R.</td>
<td>F</td>
</tr>
<tr>
<td>Baltic Sea</td>
<td>459</td>
<td>Schleswig-H.</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Szczecin</td>
<td>PL</td>
</tr>
</tbody>
</table>

Creating a Cluster in ORACLE

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

```
CREATE TABLE <table>
    (<col> <datatype>,
     ...);

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:

```
CREATE INDEX CSeaInd ON CLUSTER Cl_Sea;
```
Procedural 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.

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>,
  ...
- 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>.

Block Structure of PL/SQL

- 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 and Functions

- Invocation of procedures in a PL/SQL body:
  `<procedure> (arg1,...,argn);`
  (if a formal parameter is declared as OUT or INOUT, the respective argument must be a variable)

- 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 the return value of functions:

```sql
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.

```sql
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

```sql
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:

```sql
<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.

```sql
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:

\[
\text{<variable> <variable'>%TYPE} \\
\text{[NOT NULL] [DEFAULT <value>];}
\]

or

\[
\text{<variable> <table>.<col>%TYPE} \\
\text{[NOT NULL] [DEFAULT <value>];}
\]

- 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:
  \[
  \text{SELECT ...} \\
  \text{INTO <PL/SQL-Variable>} \\
  \text{FROM ... ;}
  \]

Example:

\[
\text{the_name country.name%TYPE} \\
\text{... :} \\
\text{SELECT name} \\
\text{INTO the_name} \\
\text{FROM country} \\
\text{WHERE name='Germany';}
\]

PL/SQL Data Types: Records

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

\[
\text{TYPE city_type IS RECORD} \\
\text{(Name City.Name%TYPE,} \\
\text{Country VARCHAR2(4),} \\
\text{Province VARCHAR2(32),} \\
\text{Population NUMBER,} \\
\text{Longitude NUMBER,} \\
\text{Latitude NUMBER);} \\
\text{the_city city_type;}
\]

anchored Type Declaration for Records

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

\[
\text{<variable> <table-name>%ROWTYPE;} \\
\]
equivalent to the above example:

\[
\text{the_city city%ROWTYPE;}
\]

Assignment to Records

- Aggregate assignment: two variables of the same record type:
  \[
  \text{<variable> := <variable'>;} \\
  \]
- assignment of a single field:
  \[
  \text{<record.field> := <variable>|<value>;} \\
  \]
- SELECT INTO: result of a query that yields a single tuple:
  \[
  \text{SELECT ...} \\
  \text{INTO <record-variable>} \\
  \text{FROM ... ;} \\
  \]

\[
\text{the_country country%ROWTYPE} \\
\text{... :} \\
\text{SELECT *} \\
\text{INTO the_country} \\
\text{FROM country} \\
\text{WHERE name='Germany';}
\]

Comparison of Records:
For comparing records, each field must be compared.
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>(<int>)
  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>;
```

```
<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

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 their 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).

PL/SQL
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;
```

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:

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

```
<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bonn</td>
</tr>
<tr>
<td>Kiel</td>
</tr>
<tr>
<td>Hamburg</td>
</tr>
</tbody>
</table>
```

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

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

Example

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

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

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.

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;
```

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.
• 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,
- prohibiting 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 Event-Condition-Action 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.

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>;

- 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:

```sql
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: 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
- 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.

```sql
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

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


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;
/

Error Handling

- Declaration Section: declaration (of names) of user-defined exceptions.
- Exception Section: Definition of actions that have to be executed in case of an exception.
- Exceptions can be raised on arbitrary places on the PL/SQL block by the RAISE statement.

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;
/

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

```sql
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);
```

```
SQL> select * from allcountry where Code='LU';
Es wurden keine Zeilen ausgewählt
```

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

**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.

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 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)

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.
• variable SELF for accessing the attributes of the host object.

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.
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;
/

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)));

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.
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.

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

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;
/

Example

SELECT VALUE(cty)
FROM City_ObjTab cty;

VALUE(City)(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)) |
Using Objects: VALUE

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

```sql
SELECT cty1.Name, cty2.Name,
       cty1.coordinates.Distance(cty2.coordinates)
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:

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

Object References

- Additional datatype for attributes: references to objects

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

- PRIMARY KEYS 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 be specified by constraints using the SCOPE concept:
  - as column constraint (only for relational tables):
    ```sql
    <ref-attr> REF <object-datatype>
    SCOPE IS <object-table>
    ```
  - as table constraint:
    ```sql
    SCOPE FOR (<ref-attr>) IS <object-table>
    ```

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

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

Example: Object Type `Organization`

```sql
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_hq_in REF City_Type,
     MEMBER FUNCTION is_member (the_country IN VARCHAR2)
          -- EU.is_member('SLO') = '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));
/
```

Table Definition:

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

Inserting objects via the object constructor:

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

Reference attribute `has_hq_in`:

```sql
UPDATE Organization_ObjTab
SET has_hq_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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbrev</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>European Community</td>
<td>EU</td>
<td>Member_List_Type(...)</td>
</tr>
</tbody>
</table>

- **Reference attributes:**

```
SELECT <ref-attr-name>
  yields an OID:
```

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbrev</th>
<th>has_hq_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>European Community</td>
<td>EU</td>
<td>&lt;oid&gt;</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Abbrev</th>
<th>has_hq_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU</td>
<td>City_Type('Brussels', 'Brabant', 'B', 951580, GeoCoord(4, 51))</td>
</tr>
</tbody>
</table>

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

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_hq_in.name
FROM Organization_ObjTab org;
```

<table>
<thead>
<tr>
<th>Abbrev</th>
<th>has_hq_in.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU</td>
<td>Brussels</td>
</tr>
</tbody>
</table>

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.

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);
/

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

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;
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;
EXECUTE make_member('EU', 'USA', 'special member');
EXECUTE make_member('XX', 'USA', 'member');
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', 'SLO')
FROM DUAL;

is_member_in('EU', 'SLO')

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_hq_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,
• 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;

Object Orientation in Oracle 8

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.
**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:

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

- indexes cannot be created for complex attributes:
  ```sql
  -- not allowed:
  CREATE INDEX city_index
  ON City_ObjTab(coordinates);
  ```
- indexes can be created for atomic components of complex attributes:
  ```sql
  CREATE INDEX city_index
  ON City_ObjTab(coordinates.Longitude, coordinates.Latitude);
  ```

**Access Permissions for Objects**

Permission to use an object type:

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

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

**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.

  > **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.

Example

```sql
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

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.

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

Object Orientation in ORACLE 8

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_hq_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_hq_in, org.people()
FROM Organization_ObjTab org;
ERROR in line 3:
ORA-00932: inconsistent datatypes
Both attributes are also not allowed alone.

New Object Relational Features in ORACLE 9

- SQL type inheritance
- Object view hierarchies
- Type evolution
- User defined Aggregate Functions
- Generic and transient datatypes
- Function-based indexes
- Multi-level collections
- C++ interface to Oracle
- Java object storage

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.
SQL Type Inheritance

- Type hierarchy:
  - supertype: parent base type
  - subtype: derived type from the parent
  - inheritance: connection from subtypes to supertypes in a hierarchy
- Subtype:
  - adding new attributes and methods
  - overriding: redefining methods
- Polymorphism: object instance of a subtype can be substituted for an object instance of any of its subtypes

Hierarchy example

```
plant
tree
flower
conifer
```

Creating Subtypes

Supertype is given by `UNDER` parameter:

```
CREATE TYPE coord_with_height UNDER coord (height NUMBER)
) NOT FINAL;
/
```

FINAL and NOT FINAL Types and Methods

- Whole type marked as `FINAL`:
  no subtypes can be derived
- Function marked as `FINAL`:
  no overriding in subtypes

Example:

```
CREATE TYPE coord AS OBJECT (latitude NUMBER, longitude NUMBER) FINAL;
/
ALTER TYPE coord NOT FINAL;
CREATE TYPE example_typ AS OBJECT (...
  MEMBER PROCEDURE display(),
  FINAL MEMBER FUNCTION move(x NUMBER, y NUMBER), ...
) NOT FINAL;
/
```
NOT INSTANTIABLE Types and Methods

- Types declared as NOT INSTANTIABLE:
  - objects of this type cannot instantiated
  - no constructor
  - “abstract class”
- Methods declared as NOT INSTANTIABLE:
  - implementation need not to be given
  - also NOT INSTANTIABLE declaration of the whole type

Examples:
CREATE TYPE generic_person_type AS OBJECT ( ...
  ) NOT INSTANTIABLE NOT FINAL;
/
CREATE TYPE example_type AS OBJECT ( ...
  NOT INSTANTIABLE MEMBER FUNCTION foobar(...) RETURN NUMBER
  ) NOT INSTANTIABLE NOT FINAL;
/
ALTER TYPE example_type INSTANTIABLE;

Attribute Substitutability

- At different places object types can be used:
  - REF type attributes
  - Object type attributes
  - Collection type attributes
- Declared type can be substituted by any of its subtypes
- Special type forced by TREAT

TREAT

- Function TREAT tries to modify the declared type into the specified type,
  e.g. a supertype into a subtype
- Returns NULL if conversion not possible
- Supported only for SQL, not for PL/SQL

Examples:
-- types: generic_shape and subtype circle_type
-- table xy:
-- column generic_col of type generic_shape
-- column circle_col of type circle_type
UPDATE xy SET circle_col =
  TREAT generic_col AS circle_type)

-- Accessing functions:
SELECT TREAT(VALUE(x) AS circle_type).area() area
FROM graphics_object_table x;
IS OF, SYS_TYPEID

- IS OF type: object instance can be converted into specified type?
  (same type or one of its subtypes)

Example:
```
-- type hierarchy:
-- plant_type ← tree_type ← conifer_type
SELECT VALUE(p)
FROM plant_table p
WHERE VALUE(p) IS OF (tree_type);
-- Result:
-- objects of type tree_type and conifer_type
```
- SYS_TYPEID: returns most specific type (subtype), syntax:
  ```
  SYS_TYPEID(<object_type_value>)
  ```

Type Evolution

Now user-defined type may be changed:
- Add and drop attributes
- Add and drop methods
- Modify a numeric attribute (length, precision, scale)
- VARCHAR may be increased in length
- Changing FINAL and INSTANTIABLE properties

Summary of SQL Type Inheritance

- Type hierarchy: supertype, subtype
- FINAL, NOT FINAL types and methods
- INSTANTIABLE, NOT INSTANTIABLE types and methods
- Overloading, overriding
- Polymorphism, substitutability
- New functions: TREAT, IS OF, SYS_TYPEID

Type Evolution: Dependencies

- Dependents: schema objects that reference a type, e.g.:
  - table
  - type, subtype
  - PL/SQL: procedure, function, trigger
  - indextype
  - view, object view
- Changes: ALTER TYPE
- Propagation of type changes: CASCADE
- Compilable dependents (PL/SQL units, views, ...):
  Marked invalid and recompiled at next use
- Table: new attributes added with NULL values, ...
Type Evolution: Example

CREATE TYPE coord AS OBJECT (
    longitude NUMBER,
    latitude NUMBER,
    foobar VARCHAR2(10),
    name VARCHAR2(10)
);
/

ALTER TYPE coord
    ADD ATTRIBUTE (height NUMBER),
    DROP ATTRIBUTE foobar,
    MODIFY ATTRIBUTE (name VARCHAR2(20));

Type Evolution: Revalidation

Fine tuning of the time for revalidation:

- ALTER TYPE:
  - INVALIDATE: bypasses all checks
  - CASCADE: propagation of type change to dependent types and tables
  - CASCADE (NOT) INCLUDING TABLE DATA: user-defined columns
- ALTER TABLE:
  - UPGRADE: conversion to latest version of each referenced type
  - UPGRADE (NOT) INCLUDING DATA: user-defined columns

Type Evolution: Limitations

- Pass of validity checks
- All attributes from a root type cannot be removed
- Inherited attributes, methods cannot be dropped
- Indexes, referential integrity constraints of dropped attributes are removed
- Change from NOT FINAL to FINAL if no subtypes exist
- ...

User Defined Aggregate Functions

- Set of pre-defined aggregate functions: MAX, MIN, SUM, ...
  They work on scalar data.
- New aggregate functions can be written for use with complex data (object types, ...):
  - feature of Extensibility Framework
  - registered with the server
  - usable in SQL DML statements (SELECT, ...)

Object Orientation in ORACLE 9

244

245

246

247
Function-based Indexes

- Index based on the return values of a function or expression:
  - Return values pre-computed and stored in the index.
- Functions have to be **DETERMINISTIC**:
  - return the same value always
  - no aggregate functions inside
  - nested tables, REF, ... are not allowed
- Additional privileges:
  - EXECUTE for the used functions
  - QUERY REWRITE
  - Some settings for Oracle to use function-based indexes
- Speed-up of query evaluation that use these functions

Java Object Storage

- Mapping of Oracle objects and collection types into Java classes with automatically generated get and set functions.
- Other direction (new in Oracle 9):
  - SQL types that map to existing Java classes
  - SQLJ = SQL types of Language Java
  - SQL types that map to existing Java classes
  - usable as object, attribute, column, row in object table
  - querying and manipulating from SQL

Function-based Indexes: Example

```sql
CREATE TYPE emp_t AS OBJECT (
    name VARCHAR2(30),
    salary NUMBER,
    MEMBER FUNCTION bonus RETURN NUMBER DETERMINISTIC );
/
CREATE OR REPLACE TYPE BODY emp_t IS
    MEMBER FUNCTION bonus RETURN NUMBER IS
        BEGIN
            RETURN SELF.salary * .1;
        END;
END;
/
CREATE TABLE emps OF emp_t;
CREATE INDEX emps_bonus_idx ON emps x (x.bonus());
CREATE INDEX emps_upper_idx ON emps (UPPER(name));
SELECT e
    FROM emps e
    WHERE e.bonus() > 2000
    AND UPPER(e.name) = 'ALICE';
```

Java Object Storage: Example

```sql
CREATE TYPE person_t AS OBJECT
    EXTERNAL NAME 'Person' LANGUAGE JAVA
    USING SQLData (
        ss_no NUMBER(9) EXTERNAL NAME 'socialSecurityNo',
        name VARCHAR(30) EXTERNAL NAME 'name',
        ...
    MEMBER FUNCTION age () RETURN NUMBER
        EXTERNAL NAME 'age () return int',
    ...
STATIC create RETURN person_t
    EXTERNAL NAME 'create () return Person',
    ...
ORDER FUNCTION compare (in_person person_t)
    RETURN NUMBER
        EXTERNAL NAME 'isSame (Person) return int'
    );
/
```

The corresponding Java class Person implements the interface SQLData.

⇒ Next unit contains more about JDBC.
Summary of New Features in Oracle 9

+ Introduction of inheritance
- Still missing OO features, e.g.:
  - multiple inheritance
  - data encapsulation (private, protected, public), but partially possible by the view concept
+ Flexibility improved: types can now be changed/extended

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

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
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 host 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

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`: the host variable contains a valid attribute value.
- `>0`: the 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 result 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

```c
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, :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.

```c
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:
  ```c
  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.

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

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

```
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 ≠ 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.
```
**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-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-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 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:

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

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

- Invocation of the DriverManager:
  ```java
  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:
```java
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>.
- 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");

<table>
<thead>
<tr>
<th>Name</th>
<th>code</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>D</td>
<td>83536115</td>
</tr>
<tr>
<td>Sweden</td>
<td>S</td>
<td>8900954</td>
</tr>
<tr>
<td>Canada</td>
<td>CDN</td>
<td>28820671</td>
</tr>
<tr>
<td>Poland</td>
<td>PL</td>
<td>38642565</td>
</tr>
<tr>
<td>Bolivia</td>
<td>BOL</td>
<td>7165257</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
Handling of Result Sets

- access to individual columns of the tuple where the cursor is currently placed by
  \[\text{result-set}.\text{get}<\text{type}\>(<\text{attribute}\>)\]
- where \(<\text{type}\>\) is a Java data type,

<table>
<thead>
<tr>
<th>Java type</th>
<th>get method</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>getInt</td>
</tr>
<tr>
<td>REAL, FLOAT</td>
<td>getFloat</td>
</tr>
<tr>
<td>BIT</td>
<td>getBoolean</td>
</tr>
<tr>
<td>CHAR, VARCHAR</td>
<td>getString</td>
</tr>
<tr>
<td>DATE</td>
<td>getDate</td>
</tr>
<tr>
<td>TIME</td>
<td>getTime</td>
</tr>
</tbody>
</table>

\(<\text{getString}\>\) does always work.

- \(<\text{attribute}\>\) can be given by the attribute name or the column index.

  countries.getString("Code");
  countries.getInt("Population");
  countries.getInt(3);

- For \(<\text{get}<\text{type}\>\>\), the values of the result tuple (SQL-data types) are converted into Java types.

### 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:

<table>
<thead>
<tr>
<th>Java type</th>
<th>JDBC-SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>CHAR, VARCHAR</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>NUMBER, NUMERIC, DECIMAL</td>
</tr>
<tr>
<td>boolean</td>
<td>BIT</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
</tr>
<tr>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT, DOUBLE</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE (day, month, year)</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME (hour, minute, second)</td>
</tr>
</tbody>
</table>

These are also used for describing metadata.
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

```java
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

- Input parameters are represented by "?"

```java
PreparedStatement pstmt = conn.prepareStatement("SELECT Population
    FROM Country
    WHERE Code = ?");
```

- "?"-parameters are assigned to values by
  <preparedStatement>.set(<type>,<pos>,<value>);
  before a PreparedStatement is submitted.

- <type>: Java data type,
- <pos>: position of the parameter to be set,
- <value>: value.

```java
pstmt.setString(1,"D");
ResultSet rset = pstmt.executeQuery();
```

- Null values are set by
  setNULL(<pos>,<type>);
  where <type> is the JDBC type of this column:

```java
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 ...)
  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)

```java
CallableStatement cstmt = conn.prepareCall("call bla()");
```
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);

```
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**
  resultSet() or getUpdateCount(): gets the next return value or update count.
  getMoreResults() and then again
  resultSet() or getUpdateCount(): proceed to the next result.

```
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);

```
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))
Handling a Sequence of Results

```java
stmt.execute(queryStringWithUnknownResults);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) {
        System.out.println("Rows changed = " + count);
        stmt.getMoreResults();
        continue;
    } else 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."
• 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.