

# Chapter 3 – Introduction to SQL

Databases lectures

Prof. Dr Kai Höfig



# SQL – Structured Query Language

- ♦ Language with a relatively simple structure
- ♦ Based on English colloquial language or slang (a lot of “syntactic sugar”)

## DDL – Data Definition Language\*

Manipulation of table schemas

- Creation/modification/deletion of
- Table schemas
- Databases
- Views
- Indexes

## DQL – Data Query Language\* \*\*

Retrieving tuples, sorting, formatting, calculations, combining data.

## DCL – Data Control Language\*\*

Rights management

## TCL – Transaction Control Language\*\*

Transaction management

## DML – Data Manipulation Language\*

Manipulation of tuples (data sets)

Change operations: Tuple  
Insertion/modification/deletion

\* In this chapter

\*\* In later chapters



# SQL versions

---

- ◆ History of the SQL language standard
  - SEQUEL (1974, IBM Research Labs San Jose)
  - SEQUEL2 (1976, IBM Research Labs San Jose)
  - SQL (1982, IBM)
  - ANSI-SQL (SQL-86; 1986)
  - ISO-SQL (SQL-89; 1989; three languages Level 1, Level 2, + IEF)
  - (ANSI / ISO) SQL2 (adopted as SQL-92)
  - (ANSI / ISO) SQL3 (adopted as SQL:1999)
  - (ANSI / ISO) SQL:2003
  - SQL/XML:2006 – adds XML handling
  - SQL:2008
  - SQL:2011 latest version (not yet widely used)
  
- ◆ Each DBMS implements SQL with different details and extensions
  - Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, IBM Db2,...
  - **Here: Microsoft SQL Server using Transact-SQL**
  - Many “dialects”, all similar to the SQL Standard, no 1:1 match



# Practical significance of SQL

---

- ◆ Diverse **uses of SQL** in business information systems, among other areas
  - Database development (create and maintain tables, views, rights, etc.)
  - Application development (manipulate and present data).
  - Website creation (dynamic websites, mostly accessed using scripting languages such as JavaScript, ASP.NET, PHP, etc.)
  - Mobile applications (iOS, Android, etc.)
  - Data Warehouses / Business Intelligence Systems
  - Corporate Information Management, esp. in ERP systems like SAP
  - etc.
  
- ◆ However, there are often **different, simultaneous access options** to the DB
  - DB management tools (such as SQL Server Management Studio) for creating DBs, granting/revoking access rights, backup, optimization, etc.
  - Access via SQL from the applications
  - In the exercise, server and client are installed on the same machine



# Why Transact-SQL?

- ◆ All SQL implementations are quite similar. It doesn't matter much, which specific dialect you learn.

Rank			DBMS	Database Model	Score		
Nov 2022	Oct 2022	Nov 2021			Nov 2022	Oct 2022	Nov 2021
1.	1.	1.	Oracle +	Relational, Multi-model	1241.69	+5.32	-31.04
2.	2.	2.	MySQL +	Relational, Multi-model	1205.54	+0.17	-5.98
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model	912.51	-12.17	-41.78
4.	4.	4.	PostgreSQL +	Relational, Multi-model	623.16	+0.44	+25.88
5.	5.	5.	MongoDB +	Document, Multi-model	477.90	-8.33	-9.45
6.	6.	6.	Redis +	Key-value, Multi-model	182.05	-1.33	+10.55
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	150.32	-0.74	-8.76
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	149.56	-0.10	-17.96
9.	9.	↑ 11.	Microsoft Access	Relational	135.03	-3.14	+15.79
10.	10.	↓ 9.	SQLite +	Relational	134.63	-3.17	+4.83

- ◆ Only two noSQL approaches in top-10
- ◆ “Despite the dominance of open source solutions on the market, MS Server is doing great. Don't be afraid that once you learn it, you will never use it. Every dollar you spend on learning MS SQL Server will pay off.”  
*learnsql.com about the most popular databases in 2020*

# Equivalent queries and query optimization

---

- ◆ Many queries in SQL or relational algebra have equivalent queries, i.e. queries that return the same result
  - In the exercises, you will learn many examples (or find them yourself)
- ◆ Some queries can be evaluated much more efficiently than (equivalent) others
- ◆ Advantage of relational algebra: expressions can be transformed (by means of mathematical rules), whereby the equivalence is guaranteed
- ◆ **Query optimiser** (part of the query processor)
  - Part of every DBMS
  - Task: transform every SQL query (usually after prior transformation into relational algebra) into an equivalent expression that can be executed as efficiently as possible
  - Are among the most complex software modules in existence!

Here: Keep is simple and readable!



## Remark

---

- ◆ In this chapter, we will learn about the creation of tables (relations) using SQL, about applying changes to existing tables and about the deletion of tables.
- ◆ Furthermore, about the creation of tuples, applying changes to existing tuples and the deletion of tuples.
- ◆ To provide a compact example to all the interactions with the database system, this slide set does not contain one single consistent example over the entire chapter. instead, every slide provides a minimalistic example, so only the relevant data and attributes are shown.



# SQL DDL – important instructions (statements)

---

## ♦ create table

- Create a new (empty) relation
- Specify the integrity constraints
- Store the information in the data dictionary

## ♦ drop table

- Delete a/an (empty) relation
- Delete the information from the data dictionary

## ♦ alter table

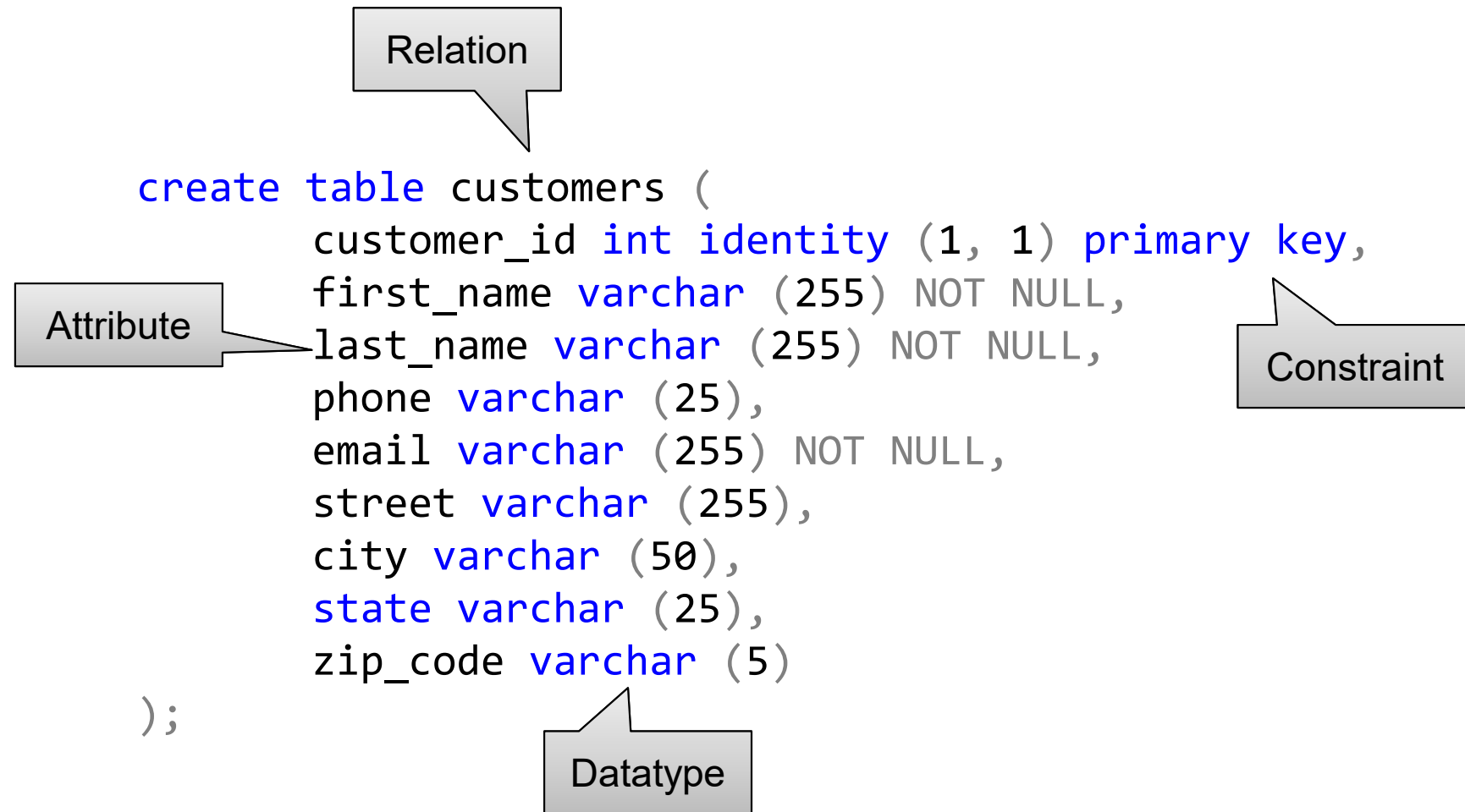
- Add/delete attributes/integrity constraints of an existing relation
- Update the information in the data dictionary
- See exercise!





# Example of `create table`

- ◆ Each table definition defines the schema of a relation. It has a name, some attributes with data types and constraints such as a primary key constraint.





# Numeric datatypes in Transact-SQL

---

- ◆ `bigint int smallint tinyint bit`

Exact number data.

- ◆ `decimal`

Fixed precision and scale numbers. `decimal(5,2)` is for example 123.45

- ◆ `float`

Double precision ISO floating point number.

E.g. 0.1234 or `CAST('2E-44' as float)`

- ◆ `time date datetime`

For example `CAST('17:56:12.1234' as time)`

`CAST('2012-08-29' as date)`

`CAST('2007-05-08 12:35:29.123' AS datetime)`

*For more, see*

*<https://learn.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql?view=sql-server-ver16>*



# String datatypes in Transact-SQL

---

- ◆ **char nchar**

Latin or Unicode (*n* for *national*) string with fixed reserved size in bytes. Since Latin encoding is one byte per character, **char**(4) reserves 4 characters as maximal size. For a two-byte encoding, such as UTF-16, **char**(4) reserves 2 characters.

Use this datatype for low variance in string lengths, for example car number plates, serial numbers or soccer team abbreviations.

- ◆ **varchar nvarchar**

Latin or Unicode string of variable size up to 2GB. The space is not reserved, but extended accordingly if data is entered or changed. So, **varchar**(100) requires less space than **char**(100), when the string size is smaller or empty most of the time.

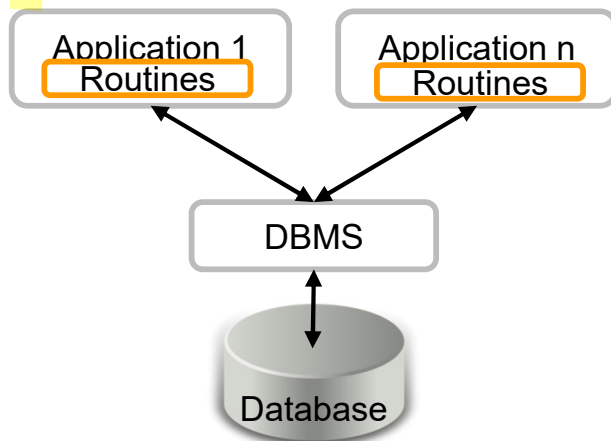
Use this datatype, when the string size varies much, such as names, streets, cities or text in general. Use **varchar**(max) with care, since a minimum 24 bytes is used.



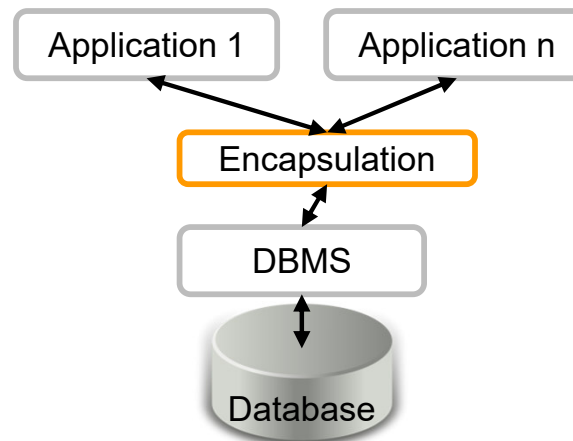
# Integrity constraints

- ◆ An **Integrity constraint** is a condition for the correctness of the database. That can refer to
  - the domain, e.g. a grade is a number between 1.0 and 5.0
  - a key integrity, e.g. a key is unique
  - a referential integrity, e.g. a foreign key points to existing elements.
- ◆ Integrity can be assured by the application using the database, a special application server or the database management system itself. Only the assurance of integrity by the DBMS is considered here.
- ◆ Each interaction that violates integrity constraints is rejected by the DBMS

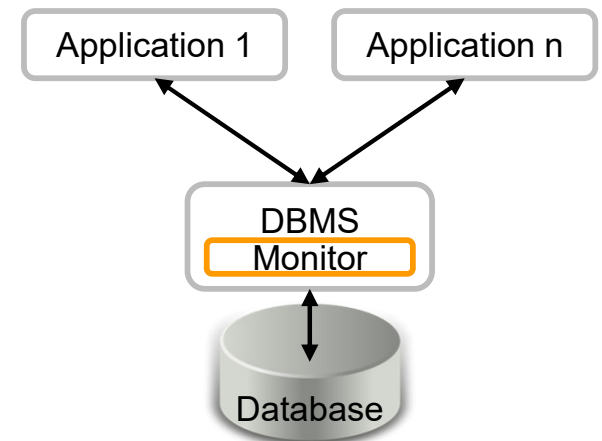
integrity constraints cost performance (checking if it is upheld), so only use as many as required



Through the application



Through encapsulation



Through monitor



# Key constraints in Transact-SQL

## ♦ primary key

Unique key property(es), stored on disk by this attribute(s) → Only one primary key.

$$\forall t_i(key) \in R : \nexists t_j(key) = t_i(key), i \neq j$$

## ♦ unique

Also unique property(es), but storage not optimized by this attribute(s). Many unique keys per table possible.

```
create table profs(  
  login char(20) primary key,  
  surname varchar(256),  
  familyname varchar(256))
```

single attribute

```
create table lectures(  
  name varchar(256),  
  semester varchar(256),  
  primary key (name, semester))
```

multiple attributes

combination is important and must be unique

```
create table cars(  
  number_plate char(16) primary key,  
  serial_number char(256) unique)
```

primary and unique key



# Artificial keys in Transact-SQL

created for the sole purpose of uniquely identifying a row in a table  
automatically generated

- ◆ `identity(5,1)`

Used to automatically generate numeric, mostly primary, keys, so called **surrogate keys**. 5 marks the start and 1 the increment per new tuple.

```
create table person(  
    pid int identity(1,1) primary key,  
    name varchar(256)  
)
```

- ◆ Inserts automatically create new increments of that ID.

```
insert into person (name) values ('Clarissa')  
insert into person (name) values ('Alexander')
```

- ◆ Inserts into an identity field (e.g. for restoring data from backup) must be explicitly allowed. Collisions with existing records still possible.

```
set identity_insert persons on
```



# Surrogate vs natural keys, what is better?

natural key: constructed from a data that is already present in the table, <https://sqlstudies.com/2016/08/29/natural-vs-artificial-primary-keys/>

- ◆ Surrogate keys take less space, especially when using references.
- ◆ Natural keys bring information into other tables using foreign keys. Lesser joins required for some applications.
- ◆ Using natural keys, the data-model is better understood.
- ◆ Natural keys can extend queries, i.e. when using multiple attributes.
- ◆ Short answer: surrogate keys. Long answer: it depends.

```
create table cars(  
  id int identity(1,1) primary key,  
  plate char(8) unique,  
  serial_number int unique  
);
```

In practice: surrogate keys and natural keys marked as unique, maybe with index.

<https://www.sqlservercentral.com/articles/using-a-surrogate-vs-natural-key#:~:text=Storage%20space,for%20the%20most%20common%20applications>



# Foreign key constraints in Transact-SQL

- ◆ The foreign key constraint enforces the foreign key property of an attribute against the referenced attribute. Also possible with multiple attributes in combination.

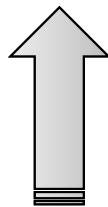
$$\pi_A(R_1) \subseteq \pi_A(R_2)$$

*“A in R1 fulfills the foreign key constraint for A in R2”*

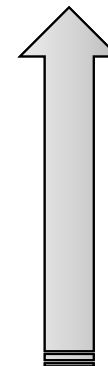
```
create table student(  
  matnr int primary key,  
  name varchar(256),  
  bdate date  
)
```

```
create table lectures(  
  name varchar(256),  
  semester varchar(256),  
  primary key (name, semester))
```

```
create table grades(  
  grade decimal(2,1),  
  student int foreign key references student(matnr),  
  lecture_name varchar(256),  
  lecture_semester varchar(256),  
  foreign key (lecture_name, lecture_semester) references lectures(name, semester))
```



...fulfills the foreign  
key constrain for...



...fulfills the foreign  
key constrain for...





# Propagating changes to foreign keys

- Especially when using natural keys, we want to be able to change the values efficiently. When deleting or changing a referenced element, we can decide to remove the referencing tuple, set to a default value or to propagate the changes. No action is the default.

```
create table cars(  
  plate char(8) primary key,  
  serial_number int unique  
);
```

```
create table owner(  
  name varchar(200) primary key,  
  his_car char(8) foreign key references cars(plate) on delete set null on update cascade  
);
```

```
insert into cars (plate, serial_number) values ('RO-SE-01', 100232), ('HH-AB-12', 7766544);  
insert into owner (name, his_car) values ('Maria', 'RO-SE-01'), ('Klaus', 'HH-AB-12');
```

```
update cars set plate='RO-TT-11' where plate='RO-SE-01';  
delete from cars where plate='HH-AB-12';
```

```
select * from owner;
```

on delete set null: set null in case of deletion  
on update cascade: update foreign key in case of change  
(i.e. propagate change to other tables)

The deletion of a car will not result in a reject, but the car of an owner will be set to null instead.



# Verification modes of conditions (1)

---

## ◆ **on update | delete**

- Specifies a trigger event that initiates verification of the condition. The update condition is executed when the tuple changes, the delete condition is executed when the entire tuple is removed.

## ◆ **cascade | set null | set default | no action**

- **cascade** the changes of the referenced value are transferred to the local value. If the referenced tuple is removed, the local tuple is also removed.
- **set null** when the referenced tuple is removed or changed, the attribute value of the local tuple is set to null
- **set default** if the referenced tuple is removed or changed, the attribute value of the local tuple is set to a default value
- **no action** default value, nothing is done locally (can result in rejecting changes)



# Null constraints in Transact-SQL

## ◆ Special value *null*

- represents the meaning “*value unknown*”, “*value not applicable*” or “*value does not exist*”, but does not belong to any value range
- In SQL, null values are denoted by **null** or  $\perp$

## ◆ null

If set for an attribute, the value can also be null or left unset in inserts. This is the default setting for all attributes.

## ◆ not null

If set for an attribute, the value cannot be null. Every insert must specify a value. This is the default for primary key or unique.

```
create table person(  
    pid int identity(1,1) primary key,  
    name varchar(256),  
    email varchar(256) not null)
```





## default and check in Transact-SQL – domain integrity

---

### ♦ default

The default clause specifies the default value, that is used, when no value is specified.

### ♦ check

The check clause is used to define value related constraints. It can be applied to one or more attributes and always relates to one specific tuple. The check constraint is evaluated every time a tuple changes.

```
create table drink(  
name varchar(255) default 'not set',  
alk int check (alk>=0 and alk<=100),  
check(name not like '%Zombie%' or (name like '%Zombie%' and alk > 0))  
)
```



# Comparison to other SQL Languages

- ◆ In other SQL languages, some more integrity constraints can be formulated.
- ◆ **create domain**: specification of a user-defined reusable datatype

```
create domain WineColour  
varchar(4)  
default 'Red'  
check (value in ('Red', 'White', 'Rose'))
```

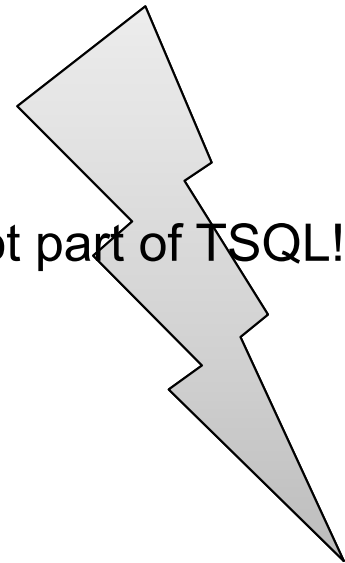
- ◆ An **assertion** expresses a condition that must always be met by the database. In TSQL we can use triggers for that.

```
create assertion Rentals  
check ((select sum (Plate) from Cars) < 1000)
```

assertions can dramatically decrease performance since it now checks with every insert/delete non-transparent to the user when these assertions are evaluated, yet decrease performance

e.g. if a car rental company does not want to manage more than 1000 cars.

Not part of TSQL!





# drop table in Transact-SQL

## ♦ drop table

removes relation data and schema from the database. The DBMS will enforce integrity constraints, such as referential integrity. Dropping a table whose tuples are referenced by another table will be rejected.

```
create table A (a int primary key);  
create table B (b int foreign key references A(a));  
drop table A;  
drop table B;
```



Could not drop object 'A' because it is referenced by a FOREIGN KEY constraint.



## alter table in Transact-SQL

---

- ◆ Existing table definition can be changed using the keyword `alter table`.

```
alter table students add addr varchar(256)
```

- ◆ Existing columns can also be adjusted or removed, e.g. different data type or `null` / `not null`, but the integrity must be preserved at all times.

```
update students set addr = ''  
alter table students alter column addr varchar(256) not null  
alter table students drop column addr
```

- ◆ Also constraints can be set, but they need a name

```
alter table students add constraint  
addr_and_name_is_unique unique(addr, name)
```



# Insert in Transact-SQL

## ♦ insert

The insert statement creates new tuples. Integrity constraints are enforced by the DBMS and so, inserts can also be rejected.

```
create table students(  
  matnr int primary key,  
  name varchar(256),  
  bdate date  
);
```

Not all values need to be specified.  
Omitted values will be filled up with  
null, default values or the  
automatically incremented key.

```
insert into students (matnr, name) values  
(1234, 'Fred'),  
(1235, 'Sarah'),  
(1236, 'Violet'),  
(1237, 'Max');
```

If the statement conflicts with  
integrity constraints, the entire  
statement is terminated.

Cannot insert duplicate key in object 'dbo.student'.  
The statement has been terminated.





# Insert using select in Transact-SQL

- ◆ The insert statement can also process calculated data from a select.

```
create table students(  
  matnr int primary key,  
  name varchar(256),  
  bdate date  
)
```

```
create table old_students(  
  matnr int primary key,  
  name varchar(256),  
  bdate date,  
  backup_date date  
)
```

```
insert into old_students (matnr, name, bdate, backup_date)  
Select  matnr,  
        name,  
        bdate,  
        cast(CURRENT_TIMESTAMP as date)  
From    students  
Where   bdate < cast('1900-01-01' as date)
```

Again, if the statement conflicts with integrity constraints, the entire statement is terminated. Example?

conflict: key constraint



## delete in Transact-SQL

---

- ◆ The delete statement removes tuples from the database preserving the integrity constraints. All tuples that match the where clause are being removed. To prevent accidentally deleting data, a primary key should be addressed, or the statement should be limited to a single tuple (not in TSQL).

```
delete from students where bdate < cast('1900-01-01' as date);  
delete from students where matnr=1234;
```

- ◆ Using the delete statement, the entire relation can be emptied. Alternatively, the truncate statement can be used. The schema is not removed.

//table is emptied but not deleted, table structure, attributes and indexes stay intact

```
delete from students;  
truncate table students;
```



## The **update** statement

---

- ◆ Using the update statement, the attributes of existing tuples can be changed. To change the attributes of a single tuple, a key needs to be addressed in the where clause.

```
create table profs(  
login char(20) primary key,  
name varchar(256),  
office varchar(256));
```

```
update profs set office='B0.9a' where login='HoKa';
```

- ◆ If multiple tuples are addressed by the where clause, all tuples are updated accordingly. Integrity constraints are preserved.

```
update profs set office='THRO-'+office;
```

*Caution: here, all offices get the prefix **THRO-***



## output Clause

---

```
create table students(  
  id int identity(1,1) primary key,  
  name varchar(200)  
);
```

- ◆ When working with artificial keys, it is important to get the key of the element just inserted. Using the **output** clause, this (and other attributes if required) can be returned. This makes surrogate keys easy to handle.

```
insert into students (name) output inserted.id values ('Klaus');
```

- ◆ That is also possible for multiple inserts and the **output** clause can also be part of **update** and **delete**.

inserted: has all data from the last insert saved

```
insert into students (name) output inserted.id  
values ('Tick'),('Trick'),('Track');  
update students set name=name+' the awesome'  
output inserted.id, inserted.name where students.name='Tick';  
delete from students output deleted.id where id=3 ;
```



# SQL DML – practical implementation of relational algebra

---

- ◆ Queries to relational DBMS are not made directly in relational algebra. They are implemented as part of the SQL DQL instead, but with substantial syntactic sugar:

```
select name as 'Human'  
from person
```

vs

$$\beta_{Human \leftarrow name}(person)$$

- ◆ SQL implements the relational algebra operations relatively directly, but SQL is declarative, relational algebra imperative.
- ◆ SQL DQL additionally provides functionality such as grouping, aggregate functions or sorting. SQL DML additionally consists of change operations: insert/update/delete statements.
- ◆ SQL has multiset semantics, relational algebra has set semantics.



# SQL core – the SFW block

$\pi, \beta$  **select** projection list  
arithmetic operations & aggregate functions

select \* = select all

$\times, \bowtie$  **from** relations to be used, possible renaming,  
join conditions

cross product: select \* from table1, table2

$\sigma$  **where** selection conditions,  
(join conditions also possible here)  
nested queries (once again an SFW block)

group by grouping for aggregate functions

having selection conditions for groups

order by output order

here

Chapter “Advanced SQL”



# SQL core – the SFW block defines a new relation, called a **view**

permanent tables are called base tables, the SFW block creates a view

$\pi$ ,  $\beta$  **select**

projection list  
defining the schema of the view  
**functions are applied to each attribute value of each tuple of the new relation**

1) should be considered first:

$\times$ ,  $\bowtie$  **from**

which tables are the data source?

data source (base-)relations, join conditions

2) should be considered second

$\sigma$  **where**

e.g. what is the join condition?

selection conditions,  
(join conditions also possible here)  
nested queries (once again an SFW block)  
**Condition is evaluated for every tuple of the new relation**

i.e. filtering, if the condition is true, tuple stays, else it is discarded

Defines a new relation  
(also called a “view”)



# from in Transact-SQL: the cross-product

- ◆ The from clause, in its simple form, creates the cross product of addressed tables.

profs:

	name	office
1	HoKa	B1.18
2	TiMa	B1.18
3	FIKe	NULL

offices:

	name
1	B0.9a
2	B1.18
3	B1.20

`select * from profs, offices`

	name	office	name
1	HoKa	B1.18	B0.9a
2	TiMa	B1.18	B0.9a
3	FIKe	NULL	B0.9a
4	HoKa	B1.18	B1.18
5	TiMa	B1.18	B1.18
6	FIKe	NULL	B1.18
7	HoKa	B1.18	B1.20
8	TiMa	B1.18	B1.20
9	FIKe	NULL	B1.20

- ◆ Tables can be renamed to shorten the query or to address the same table multiple times.

`select * from profs as p1, profs as p2 where p1.office=p2.office and p1.name != p2.name`

from can add the same table more than once but to avoid conflict, they must be given unique names (see above)

(second condition added to filter out entries where it's the same prof in both columns)

What is the purpose, what is the output?

list of persons that share the same office, since it is a cross product, it will initially include entries with the same prof, needs to be filtered (see left)



## from in Transact-SQL: named sub-queries

- ◆ The from clause addresses a comma-separated list of relations or tables. Such a table can also be generated, e.g. if it is not a base-table, using a select. This sub-query requires to have a name, since it has none.

```
select * from (select * from profs) as p
```

- ◆ Which makes less sense in simple examples, so here is another one:

```
select grades.*, best_grades.best
from grades,
      (select MIN(grade) as best, lecture
       from grades group by lecture) as best_grades
where grades.lecture=best_grades.lecture
```

What is the purpose,  
what is the output?



## from in Transact-SQL: natural join

---

- ◆ The natural join is a combination of a cross product and the where clause.

```
create table offices(  
name varchar(256) primary key,  
floor varchar(256)  
)
```

```
create table profs(  
name varchar(256),  
office varchar(256) foreign key references offices(name)  
)
```

```
select * from profs, offices where profs.office=offices.name
```

- ◆ The joint notation cleans up the where clause by putting the join criteria into the from clause.

```
select * from profs inner join offices on (profs.office=offices.name)
```



## where in Transact-SQL: simple condition(s)

- ◆ The where clause is evaluated for every tuple in the resulting relation from the cross join in the from clause.
- ◆ When the condition is true, the tuple stays, when the condition is false, the tuple is removed from the result.

```
select * from grades where grade < 1.3
```

- ◆ Constants and attributes can be compared using =, <>, >, <, >=, <=, is null or is not null and statements can be combined using and, or, not and ()

```
select *  
from   grades as g1, grades as g2  
where  g1.lecture = g2.lecture and  
       g1.grade > g2.grade
```

What is the  
purpose, what  
is the output?

```
create table grades (  
  matnr int,  
  grade decimal(2,1),  
  lecture varchar(256)  
)
```



## where in Transact-SQL: Range and wildcard selection

---

- ◆ A range of numerical values can be selected using the keyword `between`, which is just an abbreviation.

```
select * from grades where grade between 1.0 and 2.0
```

```
select * from grades where grade >= 1.0 and grade <= 2.0
```

- ◆ The wildcard selection using the keyword `like` is used to search in strings.
  - `'%'` stands for no character or any number of characters
  - `'_'` stands for exactly one character

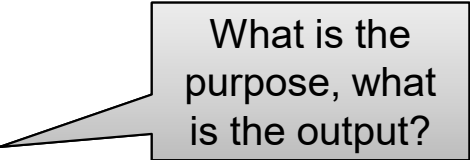
```
select * from grades where lecture like '_atabase%'
```



## where in Transact-SQL: Quantifiers and set comparisons (1)

- ◆ An attribute can also be compared against multiple values using the keyword `any`, `some`, `exists` and `in`. This makes only sense, when the values are being generated using a `select`.
- ◆ `Some` evaluates true, if the condition is true for at least one value and `any` is only true, if the condition is true for all values.  
(So `any` and `all` is the same. → Forget about `any` : )

```
select *  
from   grades  
where  grade >= all (select grade from grades)
```



What is the purpose, what is the output?

- ◆ `Exists` evaluates true, if the set of values is not empty.  

```
select * from grades where exists (select * from profs)
```



## where in Transact-SQL: Quantifiers and set comparisons (2)

- ◆ This inner select can also refer to the outer select. Important is here, that the where clause is being evaluated for every tuple separately. So, the list of values is different for every tuple. Mostly the case for *exists*.

```
select *  
from grades as g  
where grade <= all  
      (select grade  
       from grades  
       where g.lecture = grades.lecture)
```

What is the purpose, what is the output?

```
select *  
from offices  
where not exists  
      (select office  
       from profs  
       where office=offices.name)
```

Correlated query:  
tuple variable from  
outer query is used  
in inner query



## where in Transact-SQL: Quantifiers and set comparisons (3)

---

- ◆ The `in` quantifier can be used either with a fixed list like

```
select name from profs
where office in ('B1.18', 'B1.20') and office not in ('A0.1a')
```

- ◆ The `in` list can also be filled by a subquery

```
select name from profs
where office not in (
    select name
    from offices
    where name like 'A%')
```



# Correlated subqueries

---

- ♦ `in` often with **(correlated) subqueries** (synchronised subqueries) i.e. in the inner query, the relation name or tuple variable name from the `from` part of the outer query is used.

```
select name from profs
where office not in ( select name
                     from offices
                     where name like 'A%')
```

- ♦ Those can be replaced by joins, which are more easy to read

```
Select profs.name
From   profs, offices
where  profs.office=offices.name and
       offices.name not like 'A%'
```





## select in Transact-SQL

- ◆ In the select clause, a comma-separated list of columns that shall appear in the resulting table is defined. All attributes from the addresses tables in the from clause can be used.

```
select name, name, name, office from profs
```

- ◆ Naming the columns is also possible, but in contrast to the names in the from clause, the names in the select clause are just labels.

cannot be used like variables (except to create column, see below)

```
select name as 'Name', name as 'The same name again' from profs
```

- ◆ Names must be unambiguous

```
select p1.name, *, p2.* from profs as p1, profs as p2
```

How does this look like?



# select distinct in Transact-SQL

- ◆ Relational algebra has set semantics, for example

R: v1  
a  
b  
c

S: v2  
a  
b  
c

$$\pi_{v1}(R \times S) = R$$

	v1
1	a
2	b
3	c
4	a
5	b
6	c
7	a
8	b
9	c

- ◆ SQL has multiset semantics  
meaning it is possible to have double entries

select v1 from R,S

- ◆ To restore the set semantics from RA, there is a special keyword

select distinct v1 from R,S

Why is that  
“special”

sorting out doubles is expensive

	v1
1	a
2	b
3	c



## select in Transact-SQL: Scalar expressions

---

- ◆ Scalar expressions in the select clause are used to format, calculate and manipulate the result accordingly.

- ◆ Numeric calculations

```
select (grade + 1) * 0.5, grade + grade from grades
```

- ◆ Strings

```
select name + ' is in room ' + office,  
CONCAT('His office plate is this long: ', LEN(office), ' characters'),  
'And he works on floor ' + SUBSTRING(office, 0, CHARINDEX('.', office, 0))  
from profs
```

- ◆ Dates

```
select DAY(CURRENT_TIMESTAMP), CURRENT_TIMESTAMP + '00:00:20'
```



# Scalar expressions outside of `select`

---

- ◆ Scalar expressions can also be used in the `where` clause

```
select *  
from students  
where DATEDIFF(year, bdate, CURRENT_TIMESTAMP) > 10
```

- ◆ ..in `set`

```
update bikes  
set price = price * 1.1  
where CHARINDEX('Bikecompany', name, 0) > 0
```

- ◆ ..or in `having` (later) or `case`



## case in Transact-SQL: Conditional expressions

---

- ◆ Case is used in a select to conditionally generate a single column.

```
select
    name,
    case
        when office is null
        then 'No office'
        when office like 'B%' or office like 'A%'
        then 'Main building'
        else office
    end as 'Location guidance',
    office
from profs
```



## cast in Transact SQL

---

- ◆ Sometimes, typecasts are necessary. We need them to concatenate strings using +, since there is no toString() method like there is in java

```
select CAST(matnr as char(4)) + ' has a ' + cast(grade as char(3))  
from grades
```

- ◆ Alternatively CONCAT can be used

```
select CONCAT(matnr, ' has a ', grade) from grades
```

- ◆ Or we can use it to adjust results

```
select CAST(grade as int) from grades
```

- ◆ Or extend the data type for set operations

```
select cast(name as char(256)) from profs  
union  
select cast(name as char(256)) from students
```



# Set operations in Transact-SQL


- ♦ Intuitive set operations. Important: The first select defines the data types of the attributes and the number and datatypes of all sets should be the same

```
select name from profs  
union  
select name from students
```


```
select name from profs  
Intersect  
select name from students
```

```
select name from profs  
except  
select name from students
```

Schema  
mismatch



```
select name, office from profs  
union  
select name from students
```



```
select name from profs  
intersect  
select matnr from students
```

Type mismatch (is int  
expected varchar)

- ♦ **Distinct** is default, duplicates are eliminated in set operations. If duplicates are required, use **all** (so for example **union all**)



# Summary, power of the SQL core,

---

Relational algebra	SQL
Projection	<code>select (distinct)</code>
Selection	<code>where</code> without nesting
Cross product	<code>from</code> with comma-separated list
Join	<code>from, where</code>
Renaming	<code>from</code> with tuple variable; <code>as</code>
Difference	<code>except</code>
Intersection	<code>intersect</code>
Union	<code>union</code>