

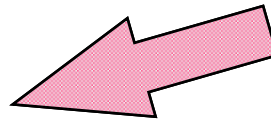
# **SQL Language: news from the 2003 standard**

---

**Jaroslav  
Pokorný**

# SQL:2003

- a lot of corrections and bug fixes
- several new features
  - data types
    - operations
    - predicates
  - operation MERGE
  - OLAP: TABLESAMPLE
  - generated columns
  - the identity columns and generators
- part 14 SQL/XML



# SQL:2003

Consists of 9 parts:

- part 1: SQL/Framework
  - part 2: SQL/Foundation
  - part 3: SQL/CLI (Call-Level Interface)
  - part 4: SQL/PSM (Persistent Stored Modules)
  - part 9: SQL/MED (Management of External Data)
  - part 10: SQL/OLB (Object Language Binding)
  - part 11: SQL/Schemes
  - part 13: SQL/JRT (Java Routines and Types)
  - part 14: SQL/XML
- parts 5, 6, 7, 8, and 12 do not exist

# New data types

- BIGINT
- MULTISSET

## Rejected types (from 1999)

- BIT
- BIT VARYING

# BIGINT

- Precision of **BIGINT**  $\geq$  precision of **INTEGER**  $\geq$  precision of **SMALLINT**
- based on **INT** and **SMALLINT**
- the same operators like **SMALLINT** and **INTEGER**

# MULTISET

- types of collections:
  - **MULTISET**
  - **ARRAY**
- multiset is a non-sorted, variable-length collection whose elements have a specified type
  - **MULTISET** - no maximal cardinality is specified
  - **ARRAY** - max. cardinality is not mandatory

# MULTISET - definition

- A INTEGER MULTISET
- B ROW( F1 BIGINT, F2 VARCHAR(4000) ) MULTISET
- C INTEGER MULTISET()
  - empty multiset of integers (*not* NULL!)
- D INTEGER MULTISET(2, 3, 5, 7)
  - non-empty multiset with several integers
- E INTEGER MULTISET(SELECT A  
FROM R WHERE A > 10)  
multiset of integers given by a  
SELECT

# MULTISET – ops. and functions

*/\* multi stands for a multiset \*/*

- **CARDINALITY**(*multi*)

- returns number of elements in *multi*

- **SET**(*multi*)

- returns content of *multi* without duplicities

- **ELEMENT**(*multi*)

- the cardinality must by 1

- returns the element (singleton)



# MULTISET - ops. and functions

- **UNNEST**(*multi*) **AS** *name*
  - returns the individual elements of *multi* as rows of a virtual table *name*

**UNNEST MULTISSET (2, 3, 5, 7) AS P**

<i><b>P</b></i>
2
3
5
7

# MULTISET – ops. and functions

- *multi1* **MULTISET** *op* [*quantifier*] *multi2*  
*op* — **UNION**, **EXCEPT**, **INTERSECT**  
*quantifier* — **ALL** or **DISTINCT**

Note: similar to the set operators

**UNION**, **EXCEPT** a **INTERSECT**

Note: quantifier **ALL** is implicit

# MULTISET – ops. and functions

SELECT

A MULTISET INTERSECT DISTINCT B

FROM R

WHERE CARDINALITY(B) > 50

# MULTISET – ops. and functions

New aggregation functions for multisets

Assumption: a group is given by **GROUP BY** or by a column

- **COLLECT** — transforms values in a group into the multiset
- **FUSION** — creates a union of all multisets in a group — amount of duplicities of a value = sum of duplicities of the value in each multiset in a group
- **INTERSECTION** — intersects all multisets in a group — amount of duplicities on a value = minimum of duplicities on the value in all multisets in a group

# MULTISET – ops. and functions

```
CREATE TABLE Logins(  
    session_id    INT NOT NULL PRIMARY KEY,  
    successful    BOOLEAN NOT NULL,  
    user_id       INT,  
    attempts      ROW(VARCHAR(128), VARCHAR(128))  
                   MULTISET);
```

username, password

```
SELECT user_id,  
       COLLECT (session_id) AS s_ids,  
       FUSION (attempts) AS all_attempts,  
       INTERSECTION (attempts) AS common_attempts  
FROM Logins  
WHERE Successful  
GROUP BY user_id;
```

# MULTISET – ops. and functions

A part of **Logins** for the user with **Id** = 8 and his/her successful attempts

Logins:	session_Id	user_Id	attempts
	a	8	multiset[(1,x),(2,y)]
	b	8	multiset[(1,x)]
	c	8	multiset[(1,x),(3,h)]

**Result** (1 row for the user with **Id** = 8 )

s_ids	all_attempts	common_attempts
multiset[a,b,c]	multiset[(1,x), (1,x), (1,x), (2,y), (3,h)]	multiset[(1,x)]

# New predicates

- comparison (multiset!) operators = and <>
- [NOT] MEMBER
- [NOT] SUBMULTISET
- IS SET, IS NOT A SET

# Predicate MEMBER

*h* [ NOT ] MEMBER [ OF ] *multi*

- *h* must be compatible with the type of elements in *multi*
- **FALSE** if *h* is not in *multi* or it is empty
- **TRUE** if *h* is equals to any element in *multi*
- **UNKNOWN** if any element in *multi* is **NULL**



# Predicate SUBMULTISET

*multi1* [ NOT ] SUBMULTISET [ OF ] *multi2*

...element types from multisets have to be compatible

- relation „be a submultiset“
- **TRUE** if  $|multi1| = |multi2|$  and each value in *multi1* has a correspondent value in *multi2*

# Predicate SET

*multi* IS [ NOT ] A SET

- *multi* is a multiset
- **TRUE** if there are no duplicities in multi
- max 1 **NULL** value in multiset

# MERGE

- combines **INSERT** and **UPDATE** statements
- rows of input (reference) table are divided into two groups according to predicate P:

*insert source table (IST)* if P is **FALSE** or **UNKNOWN**

*update source table (UST)*, if P is **TRUE**.

# MERGE

- IST rows are inserted into result table R.
- each row in R which equals to a row in UST is updated.
  - if there are more equal rows in R for one row from UST, an error is raised
- Syntax is done by **MATCHED** and **NOT MATCHED** keywords

# MERGE

```
MERGE INTO table [AS name]
  USING reference_table
  ON condition
  WHEN MATCHED THEN
    SET column = value
```

```
MERGE INTO table [AS name]
  USING reference_table
  ON condition
  WHEN NOT MATCHED THEN
    INSERT [(a_list_of_columns)]
    VALUES (a_list_of_values)
```

# MERGE

```
store(prod_id, descr, amount)  
import(prod_id, descr, amount, price)
```

```
MERGE INTO store AS ST  
  USING (SELECT prod_id, descr, amount  
        FROM import) AS IM  
  ON (ST.prod_id = IM.prod_id)  
WHEN MATCHED THEN  
  UPDATE SET amount = ST.amount + IM.amount  
WHEN NOT MATCHED THEN  
  INSERT (prod_id, descr, amount)  
  VALUES (IM.prod_id, IM.descr, IM.amount)
```

# TABLESAMPLE

- new feature for OLAP
- evaluation of aggregation functions in **samples**
- faster application development
- two different sampling methods:  
**BERNOULLI** and **SYSTEM**

# TABLESAMPLE

```
TABLESAMPLE {BERNOULLI | SYSTEM}  
(%amount) [REPEATABLE(amount_op)]
```

- **BERNOULLI**: sample table consists of appr. **%amount** of original table; probability of appearance a given row in the sample is **%amount** independently of every other row.
- **SYSTEM**: sample table consists of appr. **%amount** of original table; probability of appearance a given row in the sample can depend on rows already inserted into the sample
- **REPEATABLE**: amount of repeated operation calls (**amount\_op**) generates the same sample for the same source.



# TABLESAMPLE

Q.: Guess appr. estimation of the total salary for each department

```
SELECT dept, SUM(salary) * 10  
FROM employees  
    TABLESAMPLE BERNOULLI (10)  
REPEATABLE (5)  
GROUP BY dept
```

# Generated columns

- original columns of table: *base columns*
- *generated columns* - their value is computed from 0 or more base columns of the same row

```
CREATE TABLE employees (  
  emp_ID          INTEGER,  
  dept            string(6)  
  salary          DECIMAL(7,2),  
  addition        DECIMAL(7,2),  
  total_salary    GENERATED ALWAYS AS  
                  (salary + addition),  
  user            GENERATED ALWAYS AS  
                  (CURRENT_USER ))
```

# Identity columns & generators

- *identity column*: mechanism for automatic key population
- *generator*: used for generation of the next value of a sequence
- together provides the mechanism for automatic key generation for identity columns

# Sequence generators

## Parameters:

- data type (numeric)
- start value
- increment (positive or negative, 1 by default)
- minimal and maximal values
- cycle (when the maximum value is reached, it starts from the beginning)
- external (explicit object of the schema) or internal (part of another schema object, column for example)

# External generators

```
CREATE SEQUENCE s_name AS type
  START WITH value
  INCREMENT BY value
  MAXVALUE value
  CYCLE
```

} generator options

## ■ possibilities:

- NO CYCLE
- NO MAXVALUE, MINVALUE, NO MINVALUE

# Sequence generators

- is initialized to a base value  $Z$
- generation of the next value:  
`NEXT VALUE FOR s_name`
- returns  $Z + N * \text{incremental\_value}$ , for  $N \geq 0$ 
  - if computed value  $> \text{MAXVALUE}$  (or  $< \text{MINVALUE}$ ) and `NO CYCLE`, then raise exception

# Sequence generators

Examples:

```
Order(order_id, prod, amount)
```

```
INSERT INTO Order  
VALUES (NEXT VALUE FOR seqgen, 'prod1', 2 );
```

```
CALL myproc(NEXT VALUE FOR seqgen);
```

```
SET J = J + NEXT VALUE FOR seqgen;
```

# Sequence generators

- value of start, max, min, increment, and cycle/nocycle can be changed by alter statement

**ALTER SEQUENCE** s\_name

**RESTART WITH** new\_base\_value

- removing of sequence

**DROP SEQUENCE** s\_name



# Internal sequence generators

- **GENERATED ALWAYS** or **GENERATED BY DEFAULT**
  - **ALWAYS** — means **UPDATE** on column is not allowed; **INSERT** requires **OVERRIDING SYSTEM VALUE** (privilege)
  - **BY DEFAULT** — **INSERT** or **UPDATE** allowed; the value is generated during **INSERT**, if it is not specified in statement

# Internal sequence generators

```
CREATE TABLE Employees (  
    em_id  INTEGER  
        GENERATED ALWAYS AS IDENTITY  
        START WITH 100  
        INCREMENT 1  
        MINVALUE 100  
        NO MAXVALUE  
        NO CYCLE,  
    salary DECIMAL(7,2), ...,  
)
```

# Conclusion

- These extensions support creating analytical functions in SQL, i.e., they are usable for OLAP and now for so called Big Analytics.