Data Types

Oracle and PostgreSQL do not use the same data types. When porting the schema (table creation scripts), I tried to translate each Oracle data-type to its equivalent PostgreSQL data-type.

However, Oracle is very easy going about numeric data types because everything is a “NUMBER” and allows implicit casting of FLOAT or DOUBLE PRECISION float data-types to NUMBER in pretty much all contexts. PostgreSQL does not. Furthermore, PostgreSQL does not do any implicit data type casts when you pass an argument to a function. That argument has to be the exact type that the function expects.

The Jiggle code casts almost all numerical variables to double, which the JDBC by default turns into a java.sql.Types.DOUBLE type, which in Oracle turns into a NUMBER data-type, but in PostgreSQL turns into a DOUBLE PRECISION float type, rather than NUMERIC. Therefore, it first needs to be explicitly cast to a java.math.BigDecimal before writing them to PostgreSQL using JDBC or before passed them to a PL/pgSQL stored function if that function expects NUMERICs as input.

The C++ code use OTL as abstraction layer between the C++ codes and the databases. The Java code uses the JDBC (ODBC and PostgreSQL JDBC) drivers. The C++ code casts numbers to LONG or DOUBLE, both are 8 bytes long, it does not know of Decimal numbers. Java does have a data type that can represent very high precision numbers as Decimals, called java.math.BigDecimal. The question now is, shall I change Jiggle to use BigDecimals that get converted to NUMERIC or shall I change the schema to use DOUBLE PRECISION instead? I decided to learn more about the pros and cons.

Conclusion: *I changed the schema to use DOUBLE PRECISION rather than NUMERIC because NUMERICS take more storage and none of the programs that write to the database currently write higher precision than doubles and longs.*

Numeric DataTypes

Oracle docs about NUMBER type:
Oracle Database stores numeric data in variable-length format. Each value is stored in scientific notation, with 1 byte used to store the exponent and up to 20 bytes to store the mantissa. The resulting value is limited to 38 digits of precision. Oracle Database does not store leading and trailing zeros. For example, the number 412 is stored in a format similar to 4.12 x 10^2, with 1 byte used to store the exponent(2) and 2 bytes used to store the three significant digits of the mantissa(4,1,2). Negative numbers include the sign in their length.

Taking this into account, the column size in bytes for a particular numeric data value NUMBER(p), where p is the precision of a given value, can be calculated using the following formula:

ROUND((length(p)+s)/2))+1
where s equals zero if the number is positive, and s equals 1 if the number is negative.

Zero and positive and negative infinity (only generated on import from Oracle Database, Version 5) are stored using unique representations. Zero and negative infinity each require 1 byte; positive infinity requires 2 bytes.

PostgreSQL docs about NUMERIC type:
Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the numeric type is more akin to varchar(n) than to char(n).) The actual storage requirement is two bytes for each group of four decimal digits, plus three to eight bytes overhead.

These descriptions *sound* very similar, but in practice they are not, because of the mysterious “overhead”. For most numbers, the storage needed for a NUMERIC is much more than needed for a INTEGER, REAL, BIGINT, or DOUBLE PRECISION. See here some examples:

In PostgreSQL:
Smallest NUMERIC TYPE number stored: 2+3 = 5 bytes, example is 0.62.
But 8.8 as NUMERIC takes 10 bytes, vs 8 as a DOUBLE PRECISION and 4 as a REAL
NUMERIC(15,0) stored: 4*2 + 8 = 16 bytes a BIGINT only takes 8.

In Oracle:
0.62 uses 2 bytes, 8.8 uses 3 bytes, an integer with 10 digits takes 6 bytes. It really only uses [(significant digits)/2 plus 1] byte or less.

In the table below I’m comparing the size of a NUMERIC to that of a DOUBLE PRECISION. You can see that the NUMERIC datatypes usually take a lot more space than a DOUBLE PRECISION. Notice the number 8.8 taking up 10 bytes as a NUMERIC vs. 8 bytes as a DOUBLE PRECISION data type, and even when the 8.8 is stored specifically as a NUMERIC(8,1), i.e. a single decimal digit, it uses 10 bytes. Oddly enough, 0.62 only uses 5 bytes, which is the smallest possible NUMERIC.

```
rtdb=> select n.num,s.num as size,n.dbl,s.dbl as size, n.remark as input_format from numbers
n, sizes s where n.num=s.this_number and n.remark=s.input_type order by n.num;
                  numeric                 | size |        double        | size |
input_format
------------------------------------+------+----------------------+------+-------------------
               0.0000000000452541334 |    9 |       4.52541334e-12 |    8 | NUMERIC
 0.000000000004525413341245266799123 |   17 | 4.52541334124527e-12 |    8 | NUMERIC
                       0.000452541334 |   12 |       0.000452541334 |    8 | DOUBLE PRECISION
```

```
                0.000452541334 |     9 |       0.000452541334 |     8 | NUMERIC
                          0.62 |     5 |                 0.62 |     8 | DOUBLE PRECISION
                           8.8 |    10 |                  8.8 |     8 | DOUBLE PRECISION
                           8.8 |    10 |                  8.8 |     8 | NUMERIC
 12.2345787249881353497414972475 |    22 |     12.2345787249881 |     8 | NUMERIC
                           123 |     8 |                  123 |     8 | DOUBLE PRECISION
                           123 |     8 |                  123 |     8 | INTEGER
                       123.000 |     8 |                  123 |     8 | NUMERIC
                245678911.2345 |    14 |       245678911.2345 |     8 | NUMERIC
                    2451375572 |    12 |           2451375572 |     8 | NUMERIC
                    5555555555 |    12 |           5555555555 |     8 | BIGINT
```

We have to conclude that for large integers it is best to use BIGINT. And unless a very high precision is needed (more than 15 significant digits) or you plan to to calculations that will propagate the round-off errors associated with float types, it is better to use DOUBLE PRECISION than NUMERIC.

To confirm, I did another small test by inserting 101060 records into two different tables, numeric_table, and double_table. The input values were the “same” (doubles truncated at 15 significant digits max), but the data-types were different (NUMERIC vs DOUBLE PRECISION). Indeed, NUMERIC takes a lot more space.

```
 oid  |    table_schema    |       table_name         | row_estimate | total_bytes |
index_bytes | toast_bytes | table_bytes |   total    |   index    |   toast    |   table
-------+--------------------+--------------------------+--------------+-------------+----------
---+-------------+-------------+------------+------------+------------+------------
 22063 | public             | numeric_table            |       101060 |     6119424 |
0 |        8192 |     6111232 | 5976 kB    | 0 bytes    | 8192 bytes | 5968 kB
 22069 | public             | double_table             |       101060 |     3694592 |
0 |             |     3694592 | 3608 kB    | 0 bytes    |            | 3608 kB
```

Character DataTypes

Oracle	PostgreSQL	C++	Java	JDBC	Description	Used in AQMS Schema?
CHAR	CHAR	char	java.lang.String	Types.CHAR	Fixed length character string, zero-padded	NO
VARCHAR2[1]	VARCHAR[2]	char[]	java.lang.String	Types.VARCHAR	Variable length character string (up to 256)	YES
LONG	TEXT	char[]	java.lang.String	Types.LONGVARCHAR	Longer variable length string	NO
NCHAR	?	char[]?	java.lang.String	Types.NCHAR	Unicode fixed length	NO
NVARCHAR	?	char[]?	java.lang.String	Types.NVARCHAR	Unicode variable length	NO

Common Integer DataTypes

Oracle	PostgreSQL	C++	Java	JDBC	Description	Used in AQMS Schema?
NUMBER(1,0)[3]	BOOLEAN	bool	boolean	Types.BIT	True or False (1,0)	YES
NUMBER(3,0)	none->SMALLINT	char	byte or Integer	Types.TINYINT	1 byte int -128 to 127	YES
NUMBER(4-5,0)	SMALLINT	short	short or Integer	Types.SMALLINT	2 byte int -32768 to 32677	YES
NUMBER(5-10,0)	INTEGER	int	int or Integer	Types.INTEGER	4 byte int -2147483648 to +2147483647	YES
NUMBER(11-19,0)	BIGINT	long	long or Long	Types.BIGINT	8 byte int -9223372036854775808 to +9223372036854775807	YES

[1] VARCHAR can be used as synonym
[2] Synonym for CHARACTER VARYING
[3] NUMBER(precision,scale)--> precision is number of significant digits, scale is mantissa, scale=0 is an integer. 123.56 = 1.2356 * 10^2 precision=5, scale=2, when scale is omitted it is 0 (i.e. integer).

	SMALLSERIAL, SERIAL ,BIGSERIAL	short,int,long			Autoincrementing, > 0	NO

PostgreSQL’s data type NUMERIC sounds as if it works the same as Oracle’s data type NUMBER, so why not just translate NUMBER(15,0) to NUMERIC(15,0), for example? Because according to the docs: *However, calculations on numeric values are very slow compared to the integer types, or to the floating-point types described in the next section.*

Most primary keys are NUMBER(15,0) as the IDs of table rows. Those aren’t used for math so perhaps we could store them as NUMERIC(15,0) rather than a BIGINT (which is equivalent to NUMERIC(19,0)). A BIGINT takes 8 bytes of space, a NUMERIC(15,0) is ~2^53 significant digits, i.e. almost 7 bytes, + the one extra for the scale = also at least 8 bytes. According to the Oracle documentation, storing a NUMBER(15,0) also takes 8 bytes (9 bytes for a negative number).

PostgreSQL Data Types

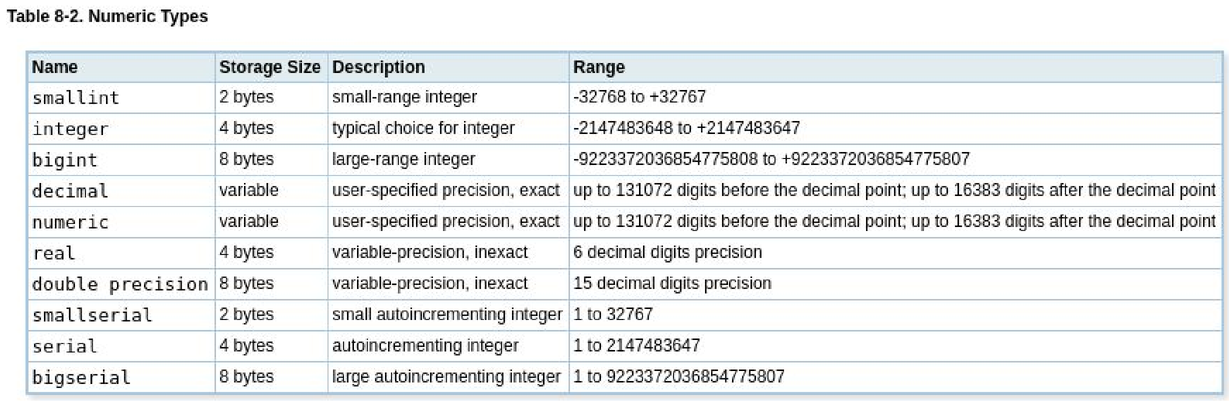


Table 8-2. Numeric Types

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

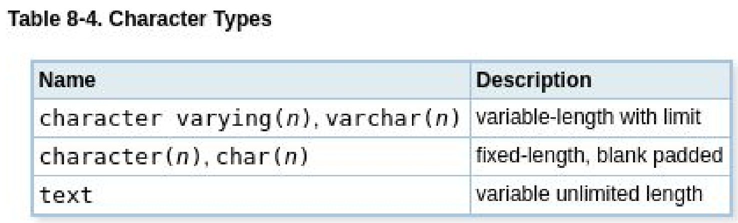
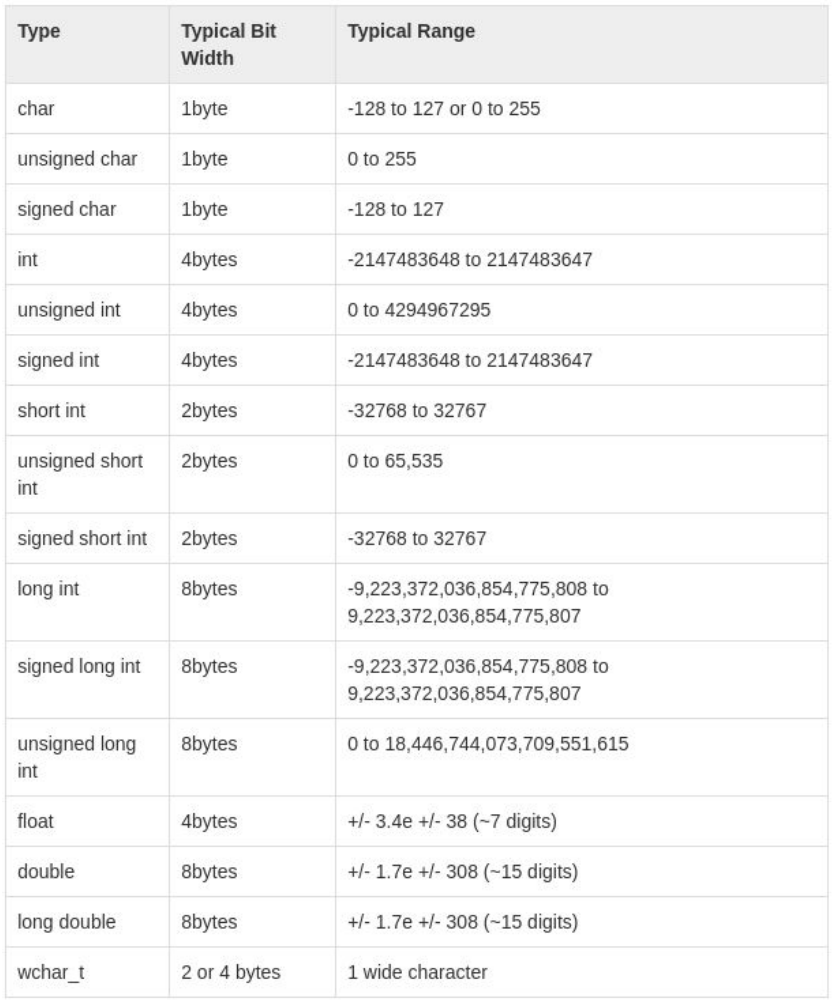


Table 8-4. Character Types

Name	Description
`character varying(n)`, `varchar(n)`	variable-length with limit
`character(n)`, `char(n)`	fixed-length, blank padded
`text`	variable unlimited length

C++ primitive types.



Type	Typical Bit Width	Typical Range
char	1byte	-128 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-128 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long int	8bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

```
# references:
#
# http://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements001.htm
# http://www.postgresql.org/docs/9.4/static/datatype.html
#
# Explanation of substitutions done below:
#
# ---------------------------------------------------------------
# ORACLE DATATYPE/identifier     => PostgreSQL DATATYPE/identifier
# ---------------------------------------------------------------
# NUMBER(1-4, 0), number(1-4, 0) => SMALLINT -32768 to +32767
# NUMBER(1-4,0), number(1-4,0)   => SMALLINT -32768 to +32767
# NUMBER(5-9, 0), number(5-9, 0) => INTEGER  -2147483648 to +2147483647
# NUMBER(5-9,0), number(5-9,0)   => INTEGER  -2147483648 to +2147483647
# NUMBER(>9, 0), number(>9, 0)   => BIGINT   -9223372036854775808 to +9223372036854775807
# NUMBER                         => ~~NUMERIC~~ DOUBLE PRECISION
# VARCHAR2, varchar2, VarChar2   => VARCHAR (=CHARACTER VARYING)
# DATE                           => TIMESTAMP
# DATE,                          => TIMESTAMP,
# DATE$                          => TIMESTAMP
# date                           => TIMESTAMP
# SYSDATE                        => LOCALTIMESTAMP
# FLOAT()                        => DOUBLE PRECISION
# FLOAT                          => DOUBLE PRECISION
# BLOB                           => BYTEA
# cast (sys_extract_utc(systimestamp) as date) => CURRENT_TIMESTAMP AT TIME ZONE 'UTC'
# cast (sys_extract_utc(systimestamp) as       => CURRENT_TIMESTAMP AT TIME ZONE 'UTC'
# cast (sys_extract_utc(systimestamp)          => CURRENT_TIMESTAMP AT TIME ZONE 'UTC'
# as date)                       => (remove, due to cast() wrapped to next line)
#  date)                         => (remove, due to cast() wrapped to next line)
#date                            => (remove, due to cast() wrapped to next line)
# SYS_EXTRACT_UTC(SYSTIMESTAMP)  => CURRENT_TIMESTAMP AT TIME ZONE 'UTC'
# create or replace public synonym => commented out (the equivalent in PostgreSQL is to
explicitely
#                                   set the search_path for each role (user) to include the
schema
#                                   names that need to be accessible)
# grant or GRANT                 => commented out, don't do this in creation script
# "                              => remove, double quotes are part of the identifier
#                                   when used (i.e. table SYSTEM_STATUS would be
"SYSTEM_STATUS")
# ENABLE                         => remove, unknown by PostgreSQL
# DISABLE                        => remove the whole line, unknown by PostgreSQL (cannot
disable check constraint)
# offset  (=a table column name) => i_offset, OFFSET is a reserved keyword in PostgreSQL
#                                   (and in SQL 2008 standard)
#
http://www.postgresql.org/docs/9.4/static/sql-keywords-appendix.html
# QB_SCHEDULE                    => use _QB_TIME instead
# COORDINATES                    => use _LATLON instead
# catseq.nextval                 => use nextval('catseq') instead
# /                              => change to semi-colon
# ---------------------------------------------------------------

cd $ORA_SRC
file_name=$1
echo "Converting $file_name in $ORA_SRC and putting new file in $PG_DEST"
```

```
grep -v "DISABLE" $file_name | sed -e "s:\NUMBER([1-4],\s*[0]):SMALLINT:"i \
     -e "s:NUMBER([1-4]):SMALLINT:"i \
     -e "s:\NUMBER([5-9],\s*[0]):INTEGER:"i \
     -e "s:NUMBER([5-9]):INTEGER:"i \
     -e "s:NUMBER([^0]\+,\s*[0]):BIGINT:"i \
     -e "s: NUMBER: ~~NUMERIC~~ DOUBLE PRECISION:"i \
     -e "s:VARCHAR2:VARCHAR:"i \
     -e "s: DATE : TIMESTAMP :" \
     -e "s: DATE,: TIMESTAMP,:" \
     -e "s: DATE$: TIMESTAMP:" \
     -e "s: date : TIMESTAMP :" \
     -e "s:SYSDATE:LOCALTIMESTAMP:" \
     -e "s:FLOAT([^0]\+):DOUBLE PRECISION:" \
     -e "s:FLOAT:DOUBLE PRECISION:" \
     -e "s:BLOB:BYTEA:"i \
     -e "s:cast (sys_extract_utc(systimestamp) as date):CURRENT_TIMESTAMP AT TIME ZONE 'UTC':"
\
     -e "s:cast (sys_extract_utc(systimestamp) as:CURRENT_TIMESTAMP AT TIME ZONE 'UTC':" \
     -e "s:cast (sys_extract_utc(systimestamp):CURRENT_TIMESTAMP AT TIME ZONE 'UTC':" \
     -e "s:as date)::" \
     -e "s:\sdate)::" \
     -e "s:^date)::" \
     -e "s:SYS_EXTRACT_UTC(SYSTIMESTAMP):CURRENT_TIMESTAMP AT TIME ZONE 'UTC':" \
     -e "s:create or replace public synonym:-- create or replace public synonym:" \
     -e "s:^grant:-- grant:"i \
     -e "s:\"::"g \
     -e "s:ENABLE::"gi \
     -e "s:offset:i_offset:"i \
     -e "s:QB_SCHEDULE:_qb_time:"i \
     -e "s:COORDINATES:_latlon:"i \
     -e "s:catseq.nextval:nextval('catseq'):" \
     -e "s:"^\/$":;:" \
              > ${PG_DEST}/${file_name}
```