https://doi.org/10.53656/ped21-7s.02s101 *Education and Maritime Industry*

S-101 CHARTS, DATABASE TABLES FOR S-101 CHARTS, AUTONOMOUS VESSEL

Vladimir Brozović, Danko Kezić, Rino Bošnjak, Filip Bojić
University of Split (Croatia)

Abstract. This article shows a way to store the data of many S-101 charts into a single Postgres database. The data model of the database with all tables is shown and explained. The concatenation of the indices from the different database tables is explained. This concatenation allows for a faster search of points/curves with certain properties. This fulfills one of the basic requirements for the purpose of navigating an autonomous vessel – that several charts can be interpreted simultaneously by a machine. Mechanisms for up-dating the database with new charts not yet present in the database are shown. Also the mechanisms for updating the charts already present in the database are explained. System limitations are briefly presented to show that in practical use there are in fact none. Memory requirements for such a type of chart storage in the database is compared with memory requirements for ISO8211 files normally used for storage of S-101 charts. With small examples it is finally shown how the stored chart information can be searched specifically.

Keywords: S-101 charts; charts database; autonomous vessel

Introduction

For the coming era in nautical science, new strategies for the exchange of information between different sources and users are defined. These new strategies are housed in a common set of standards, S-100. The first member of this group, S-101 nautical charts, is the focus of this article.

The need for a way of storing chart data above this standard, which offers multiple possibilities of searching within this data according to different criteria arose with the progress of work on a Collision Avoidance System (CAS), which requires map data among other parameters in predicting the most likely future ship positions. In addition to the ship position predictor, within the same system, the function block for determining new ship commands necessary for collision avoidance also requires the possibility of a selective search in the charts.

In this article, the authors' work is presented on how the chart data can be organized into a database so that this data can be used more efficiently by various algorithms used in autonomous vehicles.

Section 2 reviews the S-100 standard group and its members. Section 3 briefly describes the ISO8211 data format used for S-101 files (charts). S-101 records are also presented in this section. Section 4 describes the chart metadata file, assigned to a specific chart. Section 5 is a very short presentation of the chosen database software (postgres) for collecting and storing information and the reasons for its use. The main features of the GIS extension functions of the database software are also mentioned here. Section 6 briefly describes the structure of various database tables used in the database. The links between these tables are explained here. The table indices, their creation and the linking of several tables via these indices are shown. Section 7 presents the software for entering information from S-101 charts into the previously described database. Section 8 discusses the memory requirements for chart information stored in this way compared to classic ISO8211 files. Section 9 shows with an example how curves with specific attributes (e.g. Coastline) can be searched simultaneously in several charts containing a given point (e.g. current ship position).

S-100 group of standards

The whole S-100 standard group is well described in figure 1 on page 3.

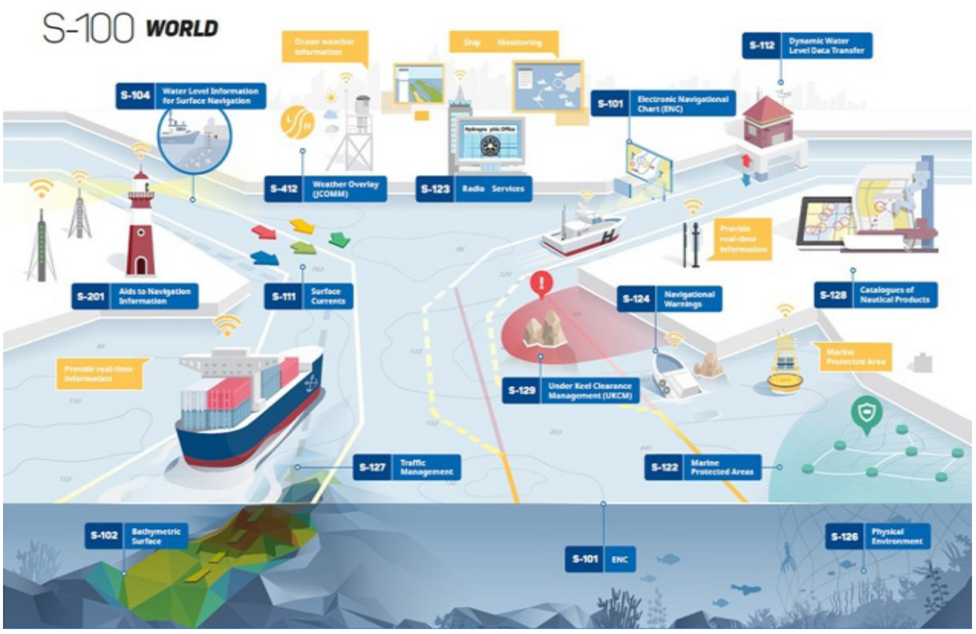

S-100 WORLD
Ocean weather information
Ship Monitoring
S-112 Dynamic Water Level Data Transfer
S-104 Water Level Information for Surface Navigation
S-101 Electronic Navigational Chart (ENC)
S-412 Weather Overlay (JCOMM)
S-123 Radio Services
Provide real-time information
S-201 Aids to Navigation Information
S-111 Surface Currents
S-124 Navigational Warnings
S-128 Catalogues of Nautical Products
Provide real-time information
Marine Protected Area
S-129 Under Keel Clearance Management (UKCM)
S-127 Traffic Management
S-122 Marine Protected Areas
S-102 Bathymetric Surface
S-101 ENC
S-126 Physical Environment

Figure 1. S-100 World

The list of all standards in the S-100 group is given as follows:

S-101 Electronic Navigational Chart ENC, S-102 Bathymetric Surface, S-103 Sub-surface Navigation, S-104 Water Level Information for Surface Navigation, S-111 Surface Currents, S-121 Maritime Limits and Boundaries, S-122 Marine Protected Areas, S-123 Radio Services, S-124 Navigational Warnings, S-125 Marine Navigational Services, S-126 Physical Environment, S-127 Traffic management, S-128 Catalogues of Nautical Products, S-129 Under Keel Clearance Management (UKCM), S-1xx Marine Services, S-1xx Digital Mariner Routing Guide, S-1xx Harbour Infrastructure, S-1xx (Social/Political), S-201 Aids to Navigation Information, S-210 Inter-VTS Exchange Format, S-211 Port Call Message Format, S-230 Application Specific Messages, S-240 DGNSS Station Almanac, S-245 eLoran ASF Data, S-246 eLoran Station Almanac, S-247 Differential eLoran Reference Station Almanac, S-401 IEHG Inland ENC, S-402 IEHG Bathymetric Inland ENC, S-411 JCOMM Ice Information, S-412 Weather Overlay (JCOMM), S-413 Weather and Wave Conditions, S-414 Weather and Wave Observations. The numbers of the standards in the S-100 group can be found in[1].

ISO 8211 files used for S-101 charts

The S-101 standard, used for electronic charts, needs a common data format for maximum interoperability. The ISO8211 file format is used for data storage. This format is described in[2]. Since this standard is subject to a charge, the details from this standard may not be reproduced here.

The main ideas incorporated in the chosen data format are:

1. Independence of the computer architecture of the target system (big/little endian, variable type sizes, etc.)
2. Self-describing
3. Compact Target file size for exchange over communication networks <= 10 Mbyte
4. Target file size for direct exchange (for example USB stick in harbor) <= 256 Mbyte
5. Expandable at any time with new features
6. Data update procedures provided
7. Data origin (organization, company, etc.) included in the file.

The main idea of the ISO8211 format is that the file consists of records that can have variable length. In the header of each record there is information about the record type and what type of information fields this record contains.

Each field type has its own subfields defined.

Each application, like in the presented case S-101 standard for electronic charts, defines its own record types, set of possible information fields for each record type and subfields in each information field.

In this definition, the S-101 specifies for each subfield how the bytes are parsed (as 8-bit integer, 16-bit unsigned integer, 16-bit signed integer, 32-bit integer, ASCII string, etc.). In this way an independence from the machine architecture was achieved.

The two special characters were defined as field and subfield delimiter.

As an example, the curve record is described here. A curve record starts with the Curve Record Identifier (CRID) field. Additional information fields are:

1. INAS-Information Association Field, is not a required field and can be repeated as needed.
2. PTAS-Point Association Field is a required field and occurs only once in this record type.
3. SEGH-Segment Header Field is a mandatory field.
4. C2IL 2-D-Integer Coordinate List Field, this list contains any number of YX pairs in integer format. The conversion to float is done by division with CMFY (Coordinate Multiplication Factor for y-coordinate) and CMFX (Coordinate Multiplication Factor for x-coordinate), both specified in the DSSI (Dataset Structure Information) field in the Dataset General Information Record.

Curve Record Identifier Field has the following subfields:

1. RCNM Record Name, 1 unsigned byte, always has the value 120 for this record type (CRID).
2. RCID Record Identification Number, unsigned 32 bit, can take all values.
3. RVER Record Version, unsigned 16 bit, contains the serial number of the Record Edition
4. RUIN Record update instruction, unsigned 8bit, always has the value 1 (Insert)

The Point Association Field PTAS has the following subfields:

1. RRNM Referenced Record, 1 byte unsigned, e.g. 110 for PRID (Point Record Identifier).
2. RRID Referenced Record Identifier, 32-bit value without sign, all values allowed
3. TOPI Topology Indicator, 1 byte unsigned, the following values are allowed:

a) For startpoint

b) for endpoint

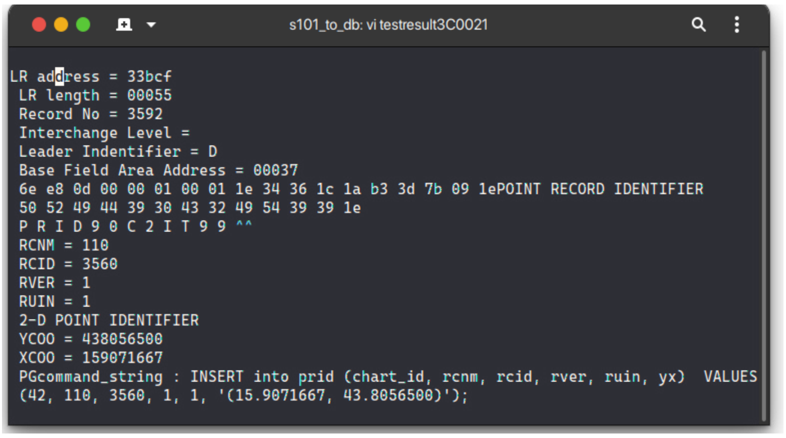
c) for start and endpoint

In figure 2, as an example, a Point Record is shown along with its INSERT command for the database.

Discovery metadata file assigned to S-101 chart

This is the XML encoded representation of exchange set catalogue features.

The most important information for navigation in this file is the name of the map, date of publication, date of revision and area of the map given by the following quantities:



```
LR address = 33bcf
 LR length = 00055
 Record No = 3592
 Interchange Level =
 Leader Indentifier = D
 Base Field Area Address = 00037
 6e e8 0d 00 00 01 00 01 1e 34 36 1c 1a b3 3d 7b 09 1ePOINT RECORD IDENTIFIER
 50 52 49 44 39 30 43 32 49 54 39 39 1e
 P R I D 9 0 C 2 I T 9 9 ^^
 RCNM = 110
 RCID = 3560
 RVER = 1
 RUIN = 1
 2-D POINT IDENTIFIER
 YCOO = 438056500
 XCOO = 159071667
 PGcommand_string : INSERT into prid (chart_id, rcnm, rcid, rver, ruin, yx)  VALUES
 (42, 110, 3560, 1, 1, '(15.9071667, 43.8056500)');
```

Figure 2. Point record, Write to s101_db database table PRID

1. westbound Longitude
2. eastbound Longitude
3. southbound Latitude
4. northbound Latitude.

Also the information whether this is a new edition or an update of an existing map. In this file there is also information about the responsible publisher, their address, responsible person and their phone number.

Information collecting and storage. The storage of the data in ISO8211 format is primarily intended for displaying this map data. In their project, the authors have identified the need to store this map data in such a way that the data can be searched in a targeted manner. In order to be able to search the data very flexibly according to various criteria, a relational database is used for data storage.

PostgreSQL Database. PostgreSQL was chosen for the project.

The following properties speak in favor of choosing this database software:

1. Free use
2. Virtually unlimited database sizes
3. Wide range of geographic functions in the GIS extension
4. Good support for synchronous (blocking) and asynchronous (non-blocking) access from Cprograms
5. Search process planner that can be controlled via parameters. For example, the search from newer to older entries can be carried out in this order and without sorting.
6. The number of coworkers in the search processes can be easily adapted to the hardware architecture (number of CPUs available)

The design of the database tables and the indexing of the fields in these tables play a very important role for the optimal usability of the data from the database in real time. For the design of real-time applications, which are based on relatively large database tables, good knowledge of the planning strategies when executing a search command in the database software used must also be available. This good knowledge of these strategies as well as of possible influences of these strategies often enables a massive reduction of the search times (often several hundred times). PostgreSQL is well documented in[3,4)].

PostGIS extension. A set of various geometric and geographic functions extending the Postgres functionality. The extension functions distinguish between geometric and geographic arguments.These extensions are described in[5].

The functions of interest are e.g. ST_Distance, ST_Intersection etc. If the arguments are of the type geography, e.g. the function ST_Distance calculates the distance between two points on the sphere. If nothing else is specified in the arguments, WGS-84 model is assumed for data in this case.

Database s101_db

The main idea of storing all nautical charts of interest in a common database is to gain the possibility of simultaneous consideration of information from several charts.

In order to be able to use this information from several charts simultaneously in the software in an optimal way, several tables are defined in the database. The tables correspond to the S-101 fields described in[6].

Consequently, the following tables are defined in the database:

arcs, atcs, ccid, crid, crsh_in_csid, csid, cuco_in_ccid, dsid, facs, fasc_in_frid, foid_in_frid, frid, ftcs, iacs, inas, irid, itcs, natc_in_crid, natc_in_fasc, natc_in_frid, natc_in_irid, prid, rias_in_srid, spas_in_frid, srid.

In addition to these tables, the chart_range table was defined, which contains some of discovery metadata entries from the XML file. These tables are all linked to each other with the chart index chart_id.

If the entries of a table are sub-elements of the entries of another table, then these entries of the sub-elements are constructed in such a way that they contain the ID of the parent entry from its table in addition to chart_id. In this way the tables can be connected arbitrarily deep hierarchically.

The IDs of all entries in all tables are automatically generated by Postgres. To use this automatically generated ID in connected sub-tables, an entry is read from the currently written table sorted by the descending ID. It is the last entry written to the table. From this database entry the searched ID is now read.

The linking of the tables via IDs is illustrated with the example an S-101 file contains several feature type records. The identifier of each such record has the following information subfields: RCNM, RCID, NFTC, RVER, RUIN. The corresponding table FRID in the database for this record type has, besides all these fields, indices frid_id as a unique index of this record and chart_id as an index of the chart.

Each feature type record can further contain several FASC (Feature Association) information fields. Each of these information fields has the following subfields: RRNM, RRID, NFAC, NARC, FAUI.

For FASC information fields in the feature type record, the fasc_in_fried table is provided in the database. In this table, there also exist indices fasc_id (index of this FASC entry in the table), frid_id (index of the feature type record to which this FASC info belongs) and chart_id as the index of the chart. Each FASC information field can further have multiple attributes with associated subfields. The description of these attributes is stored in the natc_in_fasc table. The entries of this table have the entries NATC, ATIX, PAIX, ATIN, ATVL as well as an own indexnatc_id and indices fasc_id, frid_id and chart_id, which describe the connection of these entries.

This linking is shown in figure 3.

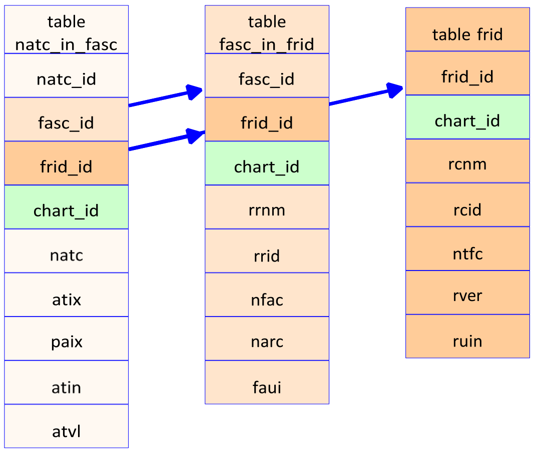


Figure 3. Example of S-101 chart table linking in Postgres

Software for data entry into the database s101_db

The software chart_array was written in C for Linux or MacOS, and generates entries in the s101_db database from S-101 charts.

In the process:

1. Parameters of the software are the database name (in shown cases 101_db) and chart name without any suffixes (neither xml nor 000). For example, the call for the chart 101HR003C0026 is chart_array s101_db 101HR003C0026

Assuming thechart data file101HR003C0026.000 and metadatafile MD_101HR003C0026_000.xml are in the current working directory.

2. the software reads the exchange set catalogue features from the xml file.

3. if the chart is not yet in the database, the chart data from the XML file is entered into the database table chart_range. After this WRITE operation, the chart_

id assigned by the database is read back and is used as the ID for all new entries in all tables. Proceed with step 7.

4. if the chart is in the database, the software checks if the chart read in now is newer than the already stored chart.

5. if so, first the corresponding chart_id will be read from the database tablechart_ range.

6. after which all entries with the selected chart_id is deleted from all database tables.

7. Then the chart data from the xml file are written as a single record into the chart table. After this INSERT operation, the chart_id assigned by the database is read back and this value is written into the chart_id field for all new entries in all tables.

8. The S-101 chart data from the ISO8211 file are parsed and from these data corresponding entries are written into the database tables.

Memory requirements compared with ISO8211 files

Of course, it is clear that data stored in this way has a larger memory footprint compared to original ISO8211 files. For example, the file for 101HR003C0026 chart occupies about 1633 kByte ondisk, and the disk space requirement for the same data stored in the Postgres database is 4816 kByte. Thanks to the development of modern NAND memories, especially in the form of eMMCs, which are mainly used in smart phones, mass storage devices of almost any size are available for the price of much less than 1 EUR per gigabyte in single level cell mode (greater data security, half the capacity with the same number of cells in IC)or 0.5 EUR per gigabyte in multi-level cellmode.

Examples of the queries used in the intelligent memory requirements compared with ISO8211 files

The chart data stored in the database allow, for example, various statements to be made about the calculated (future) position of the ship.

These statements can be used to better design the ship position predictors, which, in turn, are used in equipment designed to prevent collisions at sea.

A great advantage of the chart data stored in the database is the possibility of simultaneous use of information from any number of charts. Here are some possible statements about the calculated future ship position as examples:

1. The calculated point is onland.
2. The calculated point is out of the allowed way for ships with dangerous goods.
3. The calculated point is too close to a dangerousspoint.

The first example shows how to check whether a future ship position is on land. This is done by checking if the line between the current position and the calculated position from the future intersects a curve from the database with the attribute “Coastline”.

For this purpose, GIS extension functions are used. Initially, the search is for all maps that are of interest to the route in the near future. This is done in the following way: Position is (longitude, latitude) and search is in the chart_range table for all chart_ids for which the following condition is met:

(longitude_west_bound<longitude< longitude_east_bound)&& (latitude_south_bound<latitude< latitude_north_bound)

Assuming here that the given position is (16.36241,43.4626), this is done with the following SELECT command:

SELECT chart_id FROM chart_range WHERE longitude_west_bound<16.36241 AND longitude_east_bound>16.36241 AND latitude_south_bound<43.4626 AND latitude_north_bound>43.4626;

The result of this command is a list of all chart_id's in the database of charts that contain the position (16.36241,43.4626). Of course, the search for the relevant charts can be extended in an iterative way by including the calculated positions from the near future in the above check. Next, for each chart_id found in the previous step, the record with the ftcd value equal to 'Coastline' is searched in the ftcs table. From this record, only the value ftnc will be used in next steps. Assuming that the chart_ids 42 and 47 were found in previous step, this is done with the following SELECT commands:

SELECT ftnc FROM ftcs WHERE chart_id=42 and ftcd='Coastline';
SELECT ftnc FROM ftcs WHERE chart_id=47 and ftcd='Coastline';

For both charts (42 and 47), the ftnc value is 64. This is not necessarily always the case. The following command will then find all records in the frid table that have chart_id 42 and ftnc value 64 (all frid records that belong to a 'coast line'):

SELECT * from frid where chart_id=42 and ftnc=64 ;

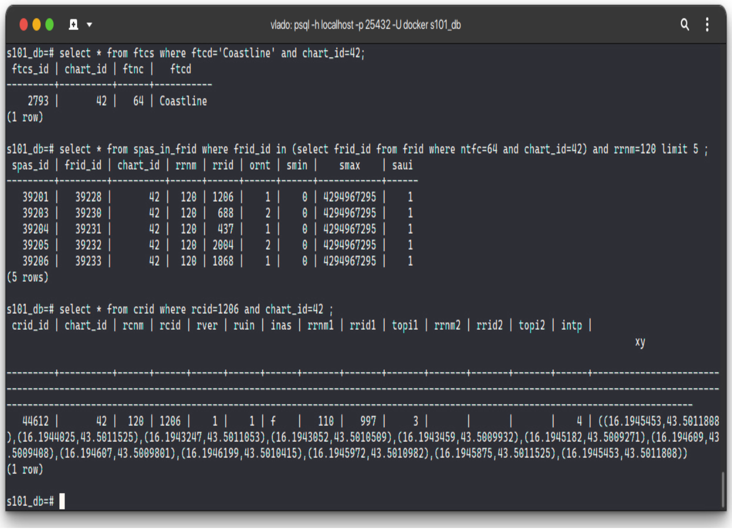
For every value iiii for frid_id in the result from the last select, the following is done:

SELECT*fromspas_in_fridwherechart_id=42ANDfrid_id=iiiiANDrrnm=120;
and from this result rrid is read.

Now for all rrid results jjjj the next SELECT in the table crid is done:

SELECT * from crid where chart_id=42 AND rcid=<rrid value jjjj from last result>;

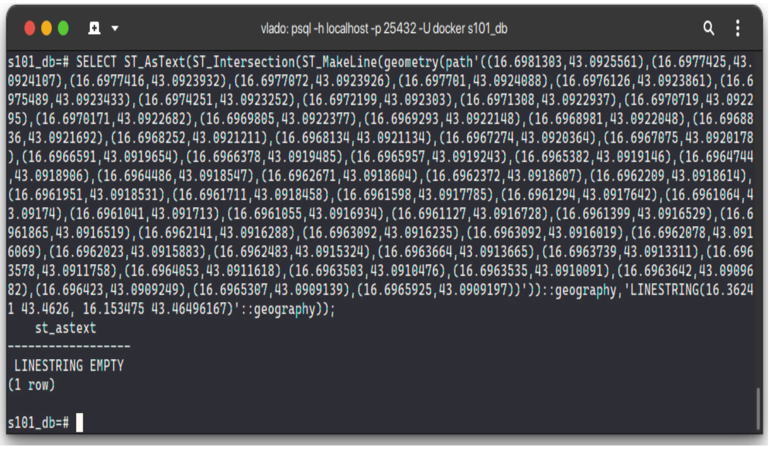
Each answer is one coastline on the map with chart_id equal to 42. An example of such a search is shown on figure 4.



```
s101_db=# select * from ftcs where ftcd='Coastline' and chart_id=42;
 ftcs_id | chart_id | ftnc |   ftcd
---------+----------+------+-----------
    2793 |       42 |   64 | Coastline
(1 row)

s101_db=# select * from spas_in_frid where frid_id in (select frid_id from frid where ntfc=64 and chart_id=42) and rrnm=120 limit 5 ;
 spas_id | frid_id | chart_id | rrnm | rrid | ornt | smin |    smax    | saui
---------+---------+----------+------+------+------+------+------------+------
   39201 |   39228 |       42 |  120 | 1206 |    1 |    0 | 4294967295 |    1
   39203 |   39230 |       42 |  120 |  688 |    2 |    0 | 4294967295 |    1
   39204 |   39231 |       42 |  120 |  437 |    1 |    0 | 4294967295 |    1
   39205 |   39232 |       42 |  120 | 2004 |    2 |    0 | 4294967295 |    1
   39206 |   39233 |       42 |  120 | 1868 |    1 |    0 | 4294967295 |    1
(5 rows)

s101_db=# select * from crid where rcid=1206 and chart_id=42 ;
 crid_id | chart_id | rcnm | rcid | rver | ruin | inas | rrnm1 | rrid1 | topi1 | rrnm2 | rrid2 | topi2 | intp |
                                                                                                        xy

---------+----------+------+------+------+------+------+-------+-------+-------+-------+-------+-------+------+----------------------
------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
   44612 |       42 |  120 | 1206 |    1 |    1 | f    |   110 |   997 |     3 |       |       |       |    4 | ((16.1945453,43.5011808
),(16.1944025,43.5011525),(16.1943247,43.5011053),(16.1943052,43.5010509),(16.1943459,43.5009932),(16.1945182,43.5009271),(16.194609,43
.5009408),(16.194607,43.5009801),(16.1946199,43.5010415),(16.1945972,43.5010982),(16.1945875,43.5011525),(16.1945453,43.5011808))
(1 row)

s101_db=#
```

Figure 4. Search for coastline curves in the database

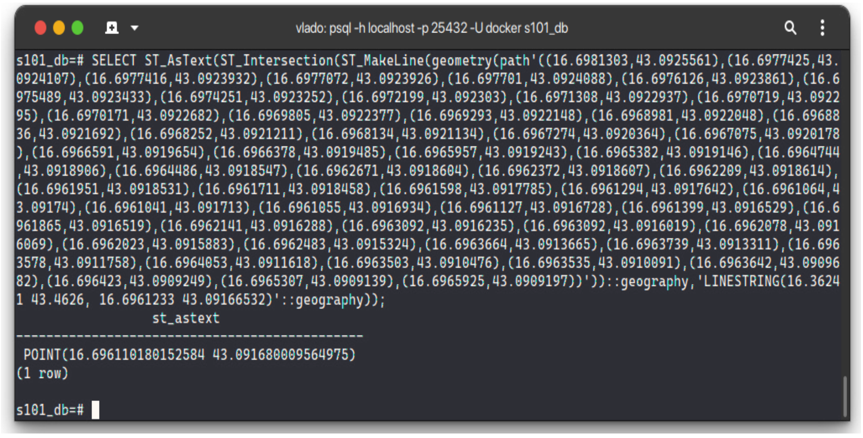
For each curve found in this way, a check can be made to see if the line between the current position point and the future position point intersects this line. This check is done by using the GIS function ST_Intersection. An example for a check of intersection between coastline curve and course line without an intersection as a result is shown on the figure 5.



```
s101_db=# SELECT ST_AsText(ST_Intersection(ST_MakeLine(geometry(path'((16.6981303,43.0925561),(16.6977425,43.
0924107),(16.6977416,43.0923932),(16.6977072,43.0923926),(16.697701,43.0924088),(16.6976126,43.0923861),(16.6
975489,43.0923433),(16.6974251,43.0923252),(16.6972199,43.092303),(16.6971308,43.0922937),(16.6970719,43.0922
95),(16.6970171,43.0922682),(16.6969805,43.0922377),(16.6969293,43.0922148),(16.6968981,43.0922048),(16.69688
36,43.0921692),(16.6968252,43.0921211),(16.6968134,43.0921134),(16.6967274,43.0920364),(16.6967075,43.0920178
),(16.6966591,43.0919654),(16.6966378,43.0919485),(16.6965957,43.0919243),(16.6965382,43.0919146),(16.6964744
,43.0918906),(16.6964486,43.0918547),(16.6962671,43.0918604),(16.6962372,43.0918607),(16.6962209,43.0918614),
(16.6961951,43.0918531),(16.6961711,43.0918458),(16.6961598,43.0917785),(16.6961294,43.0917642),(16.6961064,4
3.09174),(16.6961041,43.091713),(16.6961055,43.0916934),(16.6961127,43.0916728),(16.6961399,43.0916529),(16.6
961865,43.0916519),(16.6962141,43.0916288),(16.6963092,43.0916235),(16.6963092,43.0916019),(16.6962078,43.091
6069),(16.6962023,43.0915883),(16.6962483,43.0915324),(16.6963664,43.0913665),(16.6963739,43.0913311),(16.696
3578,43.0911758),(16.6964053,43.0911618),(16.6963503,43.0910476),(16.6963535,43.0910091),(16.6963642,43.09096
82),(16.696423,43.0909249),(16.6965307,43.0909139),(16.6965925,43.0909197))'))::geography,'LINESTRING(16.3624
1 43.4626, 16.153475 43.46496167)'::geography));
    st_astext
------------------
 LINESTRING EMPTY
(1 row)

s101_db=#
```

Figure 5. Check of intersection between curve and line with no intersection as result

An example for a check of intersection between coastline curve and courseline with an intersection as result is shown on figure 6.



```
s101_db=# SELECT ST_AsText(ST_Intersection(ST_MakeLine(geometry(path'((16.6981303,43.0925561),(16.6977425,43.
0924107),(16.6977416,43.0923932),(16.6977072,43.0923926),(16.697701,43.0924088),(16.6976126,43.0923861),(16.6
975489,43.0923433),(16.6974251,43.0923252),(16.6972199,43.092303),(16.6971308,43.0922937),(16.6970719,43.0922
95),(16.6970171,43.0922682),(16.6969805,43.0922377),(16.6969293,43.0922148),(16.6968981,43.0922048),(16.69688
36,43.0921692),(16.6968252,43.0921211),(16.6968134,43.0921134),(16.6967274,43.0920364),(16.6967075,43.0920178
),(16.6966591,43.0919654),(16.6966378,43.0919485),(16.6965957,43.0919243),(16.6965382,43.0919146),(16.6964744
,43.0918906),(16.6964486,43.0918547),(16.6962671,43.0918604),(16.6962372,43.0918607),(16.6962209,43.0918614),
(16.6961951,43.0918531),(16.6961711,43.0918458),(16.6961598,43.0917785),(16.6961294,43.0917642),(16.6961064,4
3.09174),(16.6961041,43.091713),(16.6961055,43.0916934),(16.6961127,43.0916728),(16.6961399,43.0916529),(16.6
961865,43.0916519),(16.6962141,43.0916288),(16.6963092,43.0916235),(16.6963092,43.0916019),(16.6962078,43.091
6069),(16.6962023,43.0915883),(16.6962483,43.0915324),(16.6963664,43.0913665),(16.6963739,43.0913311),(16.696
3578,43.0911758),(16.6964053,43.0911618),(16.6963503,43.0910476),(16.6963535,43.0910091),(16.6963642,43.09096
82),(16.696423,43.0909249),(16.6965307,43.0909139),(16.6965925,43.0909197))'))::geography,'LINESTRING(16.3624
1 43.4626, 16.6961233 43.09166532)'::geography));
                  st_astext
----------------------------------------------
 POINT(16.696110180152584 43.091680009564975)
(1 row)

s101_db=# 
```

Figure 6. Check of intersection between curve and line with an intersection as result

The search for curves stored as composite curves in charts and in the database has further steps in which, among other things, the CUCO table is also searched in the manner presented. The second example below shows how a sequence of Postgres and Postgres GIS queries can be used to check if a specified route is getting close to a dangerous point in the near future. Dangerous points are searched in all charts that contain the specified route. To find the charts of interest the procedure from the previous example is used. To keep the examples simple here, only the dangerous points with the attribute BuoyIsolatedDanger are searched. Assuming that the chart_id's 42 and 47 were found in previous step, this is done with the following SELECT commands:

SELECT ftnc FROM ftcs WHERE chart_id=42and ftcd='BuoyIsolatedDanger';

SELECT ftnc FROM ftcs WHERE chart_id=47and ftcd='BuoyIsolatedDanger';

For both charts (42 and 47), the ftnc value is 51. This is not necessarily always the case. The following command will then find all records in the frid table that have chart_id 42 and ftnc value 51 (all frid records that belong to a 'BuoyIsolatedDanger'):

SELECT * from frid where chart_id=42 AND ftnc=51;

For every value iiii for frid_id in the result from the last select, the following is done:

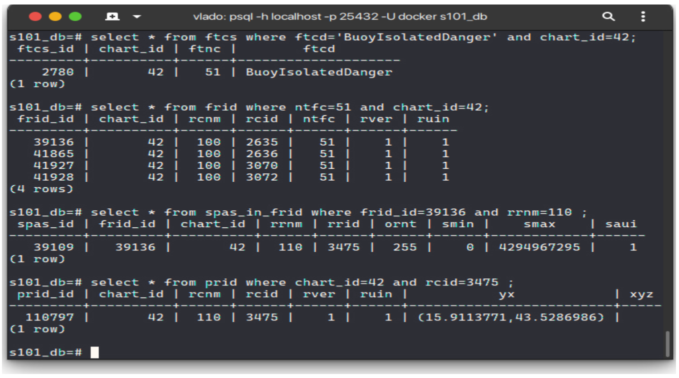
SELECT * from spas_in_frid where chart_id=42 AND frid_id=iiii AND rrnm=110;

and from this result rrid is read.

Now for all rrid results jjjj the next SELECT in the table prid is done:

SELECT * from prid where chart_id=42 AND rcid=<rrid value jjjj from last result>;

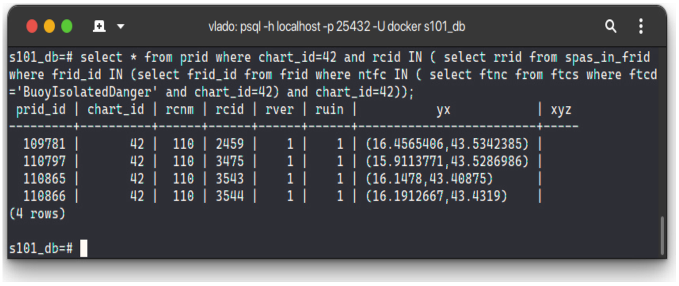
Each result is one dangerous point on the map with chart_id equal to 42.The example how to find a dangerous point in the database is shown in figure 7.



```
s101_db=# select * from ftcs where ftcd='BuoyIsolatedDanger' and chart_id=42;
 ftcs_id | chart_id | ftnc |        ftcd
---------+----------+------+--------------------
    2780 |       42 |   51 | BuoyIsolatedDanger
(1 row)

s101_db=# select * from frid where ntfc=51 and chart_id=42;
 frid_id | chart_id | rcnm | rcid | ntfc | rver | ruin
---------+----------+------+------+------+------+------
   39136 |       42 |  100 | 2635 |   51 |    1 |    1
   41865 |       42 |  100 | 2636 |   51 |    1 |    1
   41927 |       42 |  100 | 3070 |   51 |    1 |    1
   41928 |       42 |  100 | 3072 |   51 |    1 |    1
(4 rows)

s101_db=# select * from spas_in_frid where frid_id=39136 and rrnm=110 ;
 spas_id | frid_id | chart_id | rrnm | rrid | ornt | smin |    smax    | saui
---------+---------+----------+------+------+------+------+------------+------
   39109 |   39136 |       42 |  110 | 3475 |  255 |    0 | 4294967295 |    1
(1 row)

s101_db=# select * from prid where chart_id=42 and rcid=3475 ;
 prid_id | chart_id | rcnm | rcid | rver | ruin |            yx             | xyz
---------+----------+------+------+------+------+---------------------------+-----
  110797 |       42 |  110 | 3475 |    1 |    1 | (15.9113771,43.5286986) |
(1 row)

s101_db=# 
```

Figure 7. Example how to find dangerous point in the chart database

Postgres also gives the possibility to nest several commands with the construction IN in a single SELECT command. Example of such a nesting, which finds all dangerous points with type BuoyIsolatedDanger from the map with chart_id=42, is shown in figure 8.

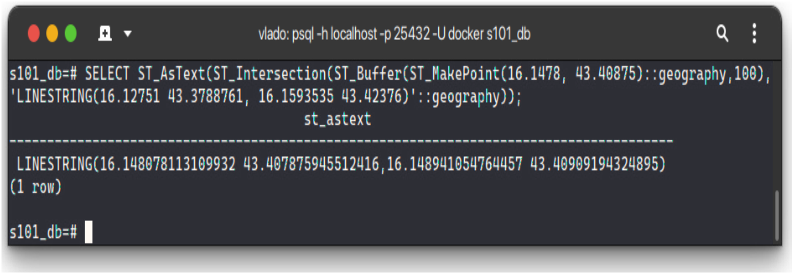


```
s101_db=# select * from prid where chart_id=42 and rcid IN ( select rrid from spas_in_frid
where frid_id IN (select frid_id from frid where ntfc IN ( select ftnc from ftcs where ftcd
='BuoyIsolatedDanger' and chart_id=42) and chart_id=42));
 prid_id | chart_id | rcnm | rcid | rver | ruin |            yx             | xyz
---------+----------+------+------+------+------+---------------------------+-----
  109781 |       42 |  110 | 2459 |    1 |    1 | (16.4565406,43.5342385) |
  110797 |       42 |  110 | 3475 |    1 |    1 | (15.9113771,43.5286986) |
  110865 |       42 |  110 | 3543 |    1 |    1 | (16.1478,43.40875)      |
  110866 |       42 |  110 | 3544 |    1 |    1 | (16.1912667,43.4319)    |
(4 rows)

s101_db=# 
```

Figure 8. Example how to find all dangerous points of specified type with single command

For each dangerous point found in this way, a check can be made to see if the line between the current position point and the future position point comes to close this point. This check is done by using the GIS function ST_Intersection, which has already been used here. First, a safety radius around the dangerous points is defined. This can be, for example, five times the width of the ship. For a ship width of 20m in this case the assumed safety radius is 100m.

In figure 9, an example is shown where the course from the point (16.12751, 43.3788761) to the point (16.1593535,43.42376) comes too close to a dangerous point (16.1475, 43.40875).



```
s101_db=# SELECT ST_AsText(ST_Intersection(ST_Buffer(ST_MakePoint(16.1478, 43.40875)::geography,100),
'LINESTRING(16.12751 43.3788761, 16.1593535 43.42376)'::geography));
                                       st_astext
------------------------------------------------------------------------------------------
 LINESTRING(16.148078113109932 43.407875945512416,16.148941054764457 43.40909194324895)
(1 row)

s101_db=# 
```

Figure 9. Example when the course to second point comes too close to a dangerous point

Between the points (16.14807811...,43.40787...) and (16.148941...,43.409091...) the course comes closer than 100m to the dangerous point.

Conclusions

The new standard group S-100 describes powerful possibilities for coding of several types of data used in navigation. Storage of this data in a relational database allows searching the data according to many criteria. The authors have presented here a way in which they built the database for storing S-101 data in an ongoing project. The software for reading in the S-101 files and storing this data in the presented database was written by authors. The authors were guided in the design of the database and powerful search capabilities resulting from this design by needs arising in the development of various algorithms for autonomous vessels. Furthermore, these search capabilities combined with information about the ship's current position, speed and course could generate information on an additional display that would help the officer in manned navigation to make some important decisions in a shorter time. With the help of such techniques, it would be relatively easy to prevent shipping accidents such as those involving the Marco Polo ferry.

NOTES

1. INTERNATIONAL HYDROGRAPHIC ORGANIZATION. S-100 Specification numbers. 2020. URL: http://s100.iho.int/S100/home/s-100-specification-numbers (visited on 05/01/2020).

2. INTERNATIONAL STANDARD ISO/IEC. ISO/IEC 8211 Information technology - Specification for a data descriptive file for information interchange. Second edition. Reviewed and confirmed in 2000. IEC, Oct. 1994.
3. The PostgreSQL Global Development Group. PostgreSQL 9.1.24 Documentation 2016. URL: https://www.postgresql.org/docs/9.1/ (visited on 05/04/2021).
4. The PostgreSQL Global Development Group. PostgreSQL 13.3 Documentation. URL: https://www.postgresql.org/files/documentation/pdf/13/postgresql-13-A4.pdf/ (visited on 13/04/2021).
5. The PostGIS Development Group. PostGIS 3.1.2 Manual. URL: https://postgis.net/docs/ (visited on 17/04/2021).
6. INTERNATIONAL HYDROGRAPHIC ORGANIZATION. IHO ELECTRONIC NAVIGATIONAL CHART PRODUCT SPECIFICATION. 1.0.0. IHO Publication S-101. 4b quai Antoine 1er, Principauté de Monaco: International Hydrographic Organization, Dec. 2018.

✉ **Vladimir Brozović**
Danko Kezić
https://orcid.org/0000-0003-2055-8039
Rino Bošnjak
https://orcid.org/0000-0002-1795-333X
Filip Bojić
https://orcid.org/0000-0002-9706-200X

Faculty of Maritime Studies
University of Split
Split, Croatia

E-mail: vladimir.brozovic@pfst.hr
E-mail: danko.kezic@pfst.hr
E-mail: rino.bosnjak@pfst.hr
E-mail: filip.bojic@pfst.hr