

WsprDaemon Timescale Databases Notes

Commands in this guide are shown in red in Courier font, and responses in blue. The author welcomes comments and corrections.

During winter 2021/22 Rob Robinett AI6VN made significant personal investments in hardware to run the WsprDaemon Timescale databases. This latest version of the Guide is consistent with the new server names etc.

Check the WsprDaemon groups.io page at <https://groups.io/g/WsprDaemon> for news.

1. Introduction and Overview	2
1.1 WsprDaemon Data Architecture	2
1.2 Paths and methods for accessing data from WsprDaemon servers.....	5
1.3 Installing PostgreSQL on your local computer.....	6
1.4 Gaining access	7
2. WsprDaemon database navigation.....	7
3. Querying spots from the WsprDaemon wsprnet database.....	8
3.1 Export query output to a file	9
3.2 Wildcards	9
3.3 Mathematical operations.....	10
3.4 Simple statistics and how to specify a time interval	10
3.5 Query on the azimuth angle at the receiver.	11
3.6 Query on the vertex latitude.....	11
3.7 Select only spot lines with distinct field entries.....	11
3.8 Using subqueries: Order by a different column.....	11
3.9 Use of Joins.....	11
4. Queries from tutorial database: wsprdaemon_spots_s, wspraemon_noise and kp	12
4.1 Table wsprdaemon_spots_s	13
4.2 Table wsprdaemon_noise_s	13
Annex A. Description of data within columns of tables in wsprnet and tutorial.....	15
Annex B. Accessing the WsprDaemon database using node.js.....	21
Annex C. Bash script to read basic spot data from database wsprnet table spots.....	22
Annex D. Skeleton of a Python script to read the WsprDaemon wsprnet database	23
Annex E. Redacted.....	24
Annex F. Octave route	24
Annex G. The briefest of introductions to using WsprDaemon with R.....	28
Annex H. A 'Big Data' approach to using data from the WsprDeamon databases	33
Annex I. A column-oriented database approach - Clickhouse	35
Annex J. Links to postgresQL APIs or notes for other languages/systems	38

1. Introduction and Overview

1.1 WsprDaemon Data Architecture

An overview of the WsprDaemon Timescale¹ databases as of May 2022 is shown in Figure 1.1. Despite this complexity there are straightforward routes for all types of users to gain access to the data using the methods outlined in this Guide.

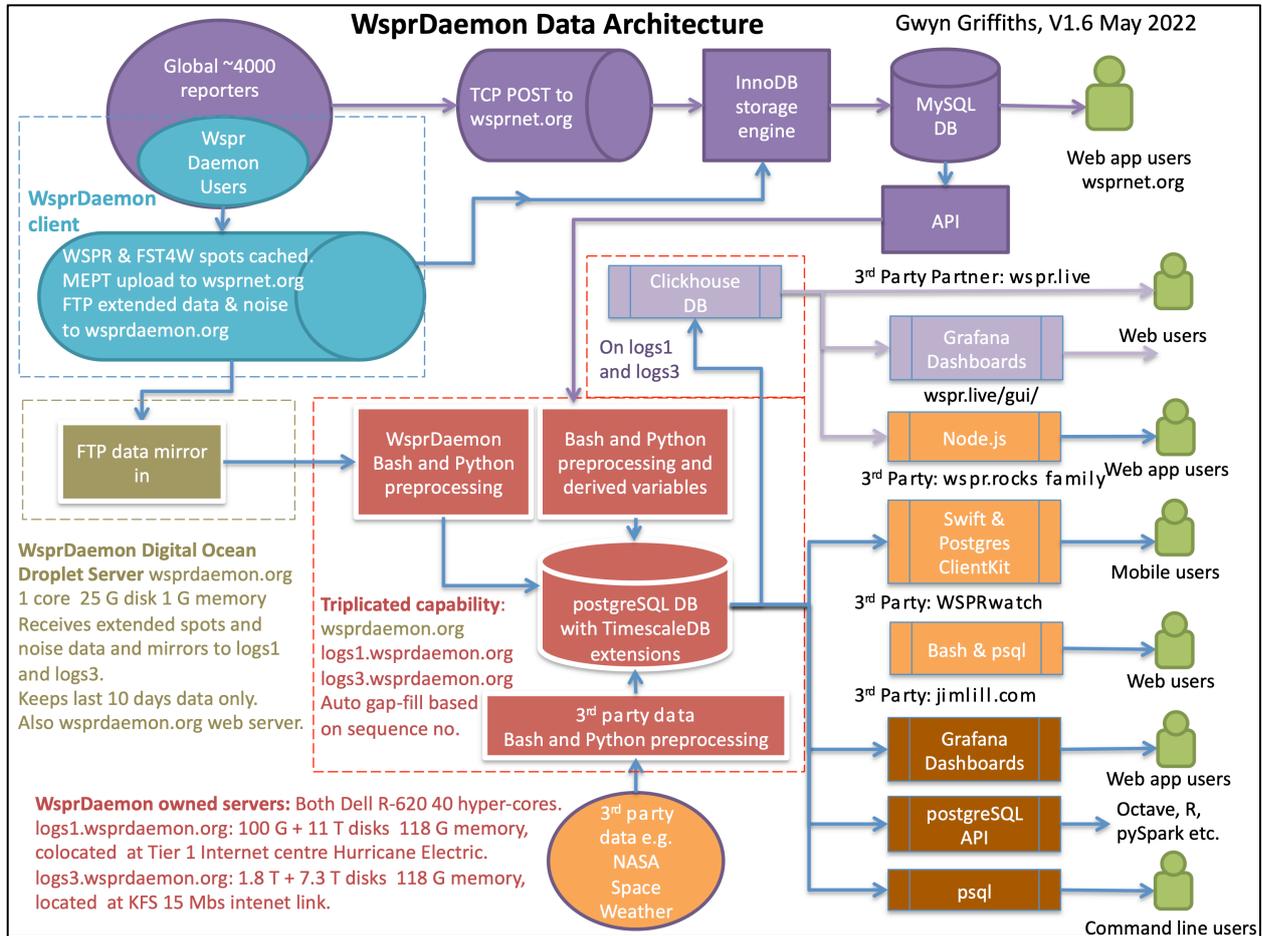


Figure 1.1 A simplified overview block diagram of the WsprDaemon data architecture.

The key points to note are:

- The WsprDaemon data architecture currently comprises two Timescale/postgreSQL databases, namely:
 - wsprnet** - with a data acquisition route represented in the diagram by the purple blocks - where WSPR spots forwarded by the global community of reporters to wsprnet.org are copied to our Timescale installation via an API every two minutes.
 - tutorial** - with a data acquisition route represented in the diagram by the cyan blocks - that accepts data from users running WsprDaemon² software to acquire their WSPR spots and report local noise.
- In this guide we use postgresQL terminology, and so, each **database** contains one or more **tables**, a full explanation of the data in each column is in Annex A, in summary:
 - Database wsprnet** contains a single **table** - **spots** - whose columns are in Table 1.1.

¹ Timescale (<https://www.timescale.com/>) provides extensions to the well-established postgresQL open source database (<https://www.postgresql.org/>) to handle time series data efficiently.

² See <https://github.com/rrobinett/wsprdaemon> on how to obtain the software

Column	Type
wd_time	timestamp without time zone
Spotnum	bigint
Date	integer
Reporter	text
ReporterGrid	character(6)
dB	smallint
MHz	double precision
CallSign	text
Grid	character(6) N
Power	smallint
Drift	smallint
distance	smallint
azimuth	smallint
Band	smallint
version	character(10)
code	smallint
wd_band	text
wd_c2_noise	real
wd_rms_noise	real
wd_rx_az	real
wd_rx_lat	real
wd_rx_lon	real
wd_tx_az	real
wd_tx_lat	real
wd_tx_lon	real
wd_v_lat	real
wd_v_lon	real

Table 1.1 Column names and types for the table spots in database wsprnet

- b. Database tutorial currently contains three tables: **kp**, **wsprdaemon_noise_s** and **wsprdaemon_spots_s**.

kp, a geomagnetic disturbance index at three-hourly intervals, is scraped each day from NOAA's [Space Weather Prediction Center](#). The column names and types are listed in Table 1.2. The mid-latitude index, **kp_mid**, is from Fredericksburg, Virginia, and **kp_high** from College, Alaska.

Column	Type
time	timestamp without time zone
kp_mid	integer
kp_high	integer
kp_planetary	integer

Table 1.2 Column names and types for the table kp in database tutorial

wsprdaemon_noise_s: Users of WsprDaemon software, particularly those using KiwiSDRs, have the option of uploading estimates of local noise obtained at the same time and in the same frequency band as the decoded WSPR transmissions³. The column names and types are listed in Table 1.3, with full details in Annex A. Filed seqnum is generated automatically on data insertion as the unique data record sequence number.

³ Details of the noise estimation algorithms and examples in use are available in: Griffiths, G., Robinett, R. and Elmore, G., 2020. Estimating LF-HF band noise while acquiring WSPR spots. QEX, September-October 2020 and also in a detailed [report](#) available on ResearchGate.net.

Column	Type
time	timestamp without time zone
site	text
receiver	text
rx_grid	text
band	text
rms_level	double precision
c2_level	double precision
ov	integer
seqnum	bigint

Table 1.3 Column names and types for the table `wsprdaemon_noise_s`

wsprdaemon_spots_s:

WsprDaemon access additional data fields within WSJT-X's wsprd decoder. These fields are not uploaded to wsprnet.org, but are included in this table in case they may be of interest to some in the WSPR community, see Annex A for a full description.

Column	Type
time	timestamp without time zone
band	text
rx_grid	text
rx_id	text
tx_call	text
tx_grid	text
SNR	double precision
c2_noise	double precision
drift	double precision
freq	double precision
km	double precision
rx_az	double precision
rx_lat	double precision
rx_lon	double precision
tx_az	double precision
tx_dBm	double precision
tx_lat	double precision
tx_lon	double precision
v_lat	double precision
v_lon	double precision
sync_quality	integer
dt	double precision
decode_cycles	integer
jitter	integer
rms_noise	double precision
blocksize	integer
metric	integer
osd_decode	integer
receiver	character varying
nhardmin	integer
ipass	integer
seqnum	bigint
mode	smallint
ov_count	integer

Table 1.4 Column names and types for the table `wsprdaemon_spots_s`

1.2 Clickhouse database

WsprDaemon servers logs1 and logs3 host the Clickhouse database and tables of Arne, accessed as described at wsprr.live and briefly in Annex I. Clickhouse is a very fast column-oriented database, well suited to time series queries and obtains its data from the WsprDaemon scrape of wsprrnet.org and the extended spots and noise tables. Note that Clickhouse has its own syntax.

1.3. Paths and methods for accessing data from WsprDaemon servers

Read access to the WsprDaemon [wsprrnet](https://wsprrnet.org) database table **spots**, takes four main forms:

1. Using an **App** from a third-party developer, where most of the intricacies are hidden from view. These Apps provide the basic search functionality found on wsprrnet.org and added graphs and tables. Currently two Apps access our [wsprrnet](https://wsprrnet.org) **spots** table:
 - a. wsprr.rocks. As well as simple queries, Phil Barnard, VK7JJ, provides a form-filling advanced query option to use our postgresSQL interface, Figure 1.2. The time taken for each query is shown, and a running average; on 10 November 2020 the average response time over 1654 queries was 0.7 seconds.

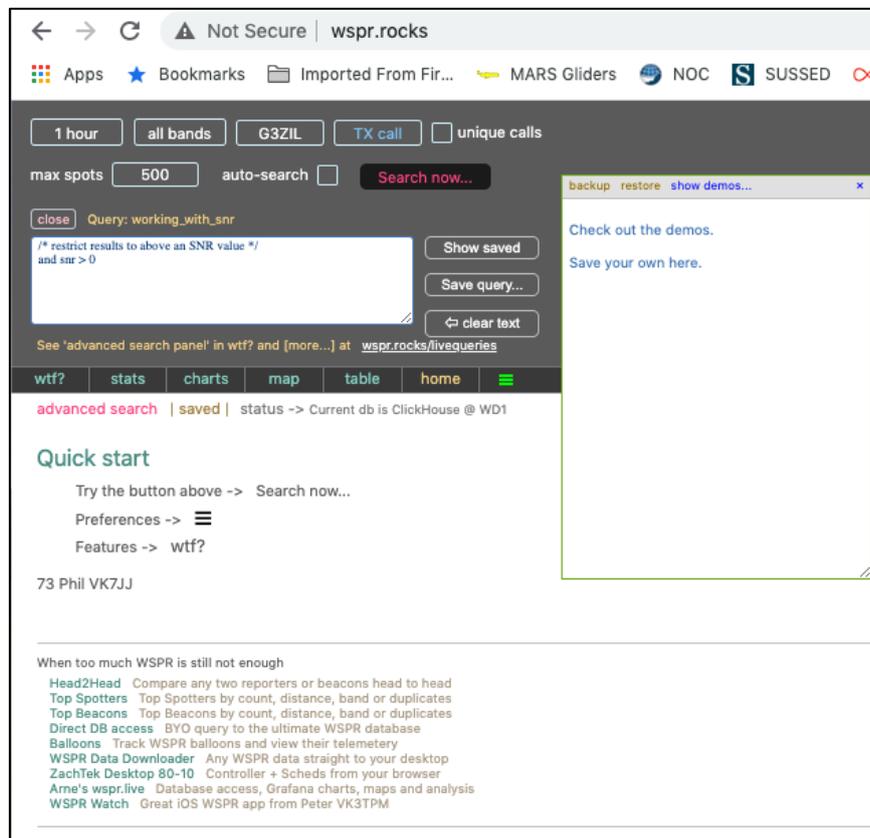


Figure 1.2 Screen shot of the advanced search panel at wsprr.rocks, which also shows several other search tools. In normal use, the data is obtained via Clickhouse.

- b. [wsprrwatch](https://wsprrwatch.com), Figure 1.3, is an iOS App by Peter Marks, VK2TPM, available at <https://apps.apple.com/us/app/wspr-watch/id532487317>

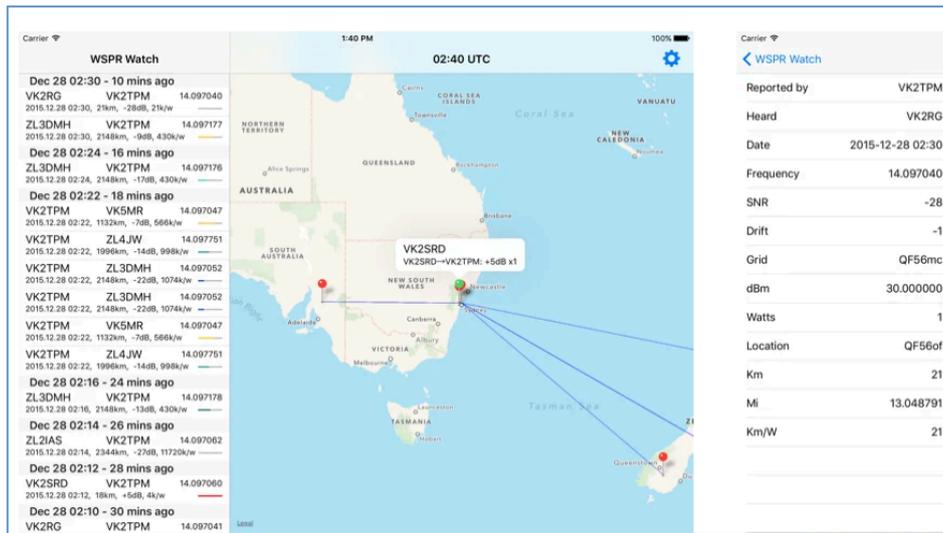


Figure 1.3 Screen shot of WSPRwatch, an iOS app from Peter Marks, VK2TPM

2. Using **Grafana**⁴, a powerful graphing package with built-in connections to PostgreSQL databases. We have a selection of Grafana Dashboards on the WsprDaemon server at logs1.wsprdaemon.org:3000 where you are welcome to use the guest userid Open and password Open. A full Guide to using Grafana with WsprDaemon data is available via the WsprDaemon website⁵.
3. Using **node.js** - this is the method used by Phil Barnard, VK7JJ, to access the data, and he has kindly provided details in **Annex B** as it is clear that others are interested in this route. Questions should be directed to Phil at phil 'at' perite.com.
4. Using **psql** - a command line tool to work with a PostgreSQL database. Two modes are available:
 - a. **Batch command line:** This mode is useful in, among other applications, bash scripts. Care is needed to get the quotes syntax right when calling for column names with upper case letters and especially if mixed with single quotes needed with character or text data values. A simple example of a bash script used by Jim Lill, WA2ZKD is shown in **Annex C**.
 - b. **Interactive mode:** The rest of this guide concentrates on **psql** in interactive mode, but in doing so, provides a rich set of PostgreSQL examples that can be used with node.js, Grafana, or any other method for accessing the data where standard PostgreSQL is used.

Other routes for access that enable data analysis packages to use our databases include:

1. Python and subsequently matplotlib, **Annex D** shows a skeleton Python script for accessing our databases and obtaining data.
2. Octave is a well-established and widely used scientific and engineering data analysis package. PostgreSQL⁶ is the only database system it currently connects to directly. The route for Octave V6 is outlined in **Annex F**.

1.4 Installing PostgreSQL on your local computer

⁴ See <https://grafana.com/>

⁵ See <http://wsprdaemon.org/grafana.html>

⁶ See <https://octave.sourceforge.io/database/index.html>

Note: If you already have an SQL program installed, e.g. SQLite or MySQL there may be an issue when it comes to installing PostgreSQL - unfortunately these problems may only come to light as an installation is attempted.

See <https://www.postgresql.org/download/> for packages and installers for a range of computers.

(A reminder, command line inputs in this document are in red).

If you intend to use Python to read from the databases on the WsprDaemon server you will need an adapter program `psycopyg2` for Python3, see Annex D for an example. You will of course need Python 3 installed, and you may need `pip3` if not already installed:

```
sudo apt install python3-pip
sudo pip3 install psycopyg2
```

Note that you do not need an installation on your own machine of the Timescale DB extensions that are used on the WsprDaemon server alongside PostgreSQL. That is, unless you intend to create your own Timescale databases.

This guide can only provide the bare essentials on using PostgreSQL to query the WsprDaemon databases. Full details on PostgreSQL are available online⁷ and at a very useful simplified tutorial website⁸.

1.5 Gaining access

We have set up a universal read-only user id: `wdread` with password `JTWSR2008`

With PostgreSQL installed on your computer you can access the database **wsprnet** on the WsprDaemon server using:

```
psql -U wdread -d wsprnet -h logs1.wsprdaemon.org
Password for user wdread: JTWSR2008
wsprnet=>
```

In this command line, the `-d` option connects us to Database **wsprnet**, which, as a reminder, contains the single table **spots** with data acquired via an API from `wsprnet.org`. The `-h` option declares the address of the WsprDaemon server, here it is `logs1.wsprnet.org`.

The PostgreSQL prompt shows the database name.

To connect directly to the **tutorial** database, with its tables **kp**, **wsprdaemon_noise_s** and **wsprdaemon_spots_s** simply change `-d wsprnet` to `-d tutorial` in the command line above.

Sections 2–4 of this guide cover navigation and use of the database using `psql` through a wide range of example queries.

2. WsprDaemon database navigation

Once connected to a database as in section 1.4, to list the **tables** in the **database** use `\d`:

```
wsprnet=> \d
          List of relations
Schema | Name | Type | Owner
-----+-----+-----+-----
public | spots | table | postgres
public | spots_Spotnum_seq | sequence | postgres
public | spots_spotnum_seq | sequence | postgres
```

To list the **columns** and **data types** within a **table**, use `\d` with the table name:

⁷ See <https://www.postgresql.org/>

⁸ See <https://www.postgresqltutorial.com/>

\d spots

This will output the list shown in Table 1.1. We will use these column names in our queries to table spots, most are self explanatory to WSPR users, see Annex A for a full annotated list.

There is no need to log out and then back in to work with the other database, use \c and the database name, for example, if connected to database wsprnet, to connect to database tutorial:

```
wsprnet=> \c tutorial
psql (12.2, server 12.4 (Ubuntu 12.4-1.pgdg18.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
compression: off)
You are now connected to database "tutorial" as user "wread".
tutorial=>
```

There is an extensive help system, accessible via \h for a list of commands, and then \h command for specific details.

To exit use \q

3. Querying spots from the WsprDaemon wsprnet database

This section provides examples of interactive queries on table **spots** in database **wsprnet** following on from the access and database selection details in Sections 1 and 2. Virtually all examples use very simple postgresSQL expressions of the form:

```
SELECT something FROM table_name WHERE one_or_more_conditions
one_or_more_options;
```

The expression, which can span several lines, must end with a semicolon. If nothing seems to be happening, you may have forgotten the semicolon. If you forget, just type ; at the next prompt. The WHERE clause is not always needed.

To see a short example, for the last 10 records with the newest first from the table **spots** **order by time desc** is included as an option, and **limit 3** sets how many records to output to the screen. The * signifies all fields.

First, connect to the database:

```
psql -U wread -d wsprnet -h logs1.wsprdaemon.org
Password for user wread: JTWSR2008
wsprnet=> select * from spots order by wd_time desc limit 3;
```

wd_time	Spotnum	Date	Reporter	ReporterGrid	dB	MHz	CallSign	Grid	Power	Drift	distance	azimuth	Band
version	code	wd_band	wd_c2_noise	wd_rms_noise	wd_rx_az	wd_rx_lat	wd_rx_lon	wd_tx_az	wd_tx_lat	wd_tx_lon	wd_v_lat	wd_v_lon	
2020-11-12 09:32:00	2581566325	1605173520	W8SGT	EN80lc	7	3.570046	KC9NBV	EM69oe	30	0	337	71	3
1.9.1	1	80	-999.9	-999.9	254	40.104	-83.042	71	39.188	-86.792	40.104	-83.042	
2020-11-12 09:32:00	2581566329	1605173520	W8SGT	EN80lc	7	3.570053	N2NOM	FN22bg	33	0	646	251	3
1.9.1	1	80	-999.9	-999.9	66	40.104	-83.042	251	42.271	-75.875	42.271	-75.875	
2020-11-12 09:32:00	2581566335	1605173520	W8SGT	EN80lc	-7	3.570102	N1ZPY	FN64cq	37	0	1349	253	3
1.9.1	1	80	-999.9	-999.9	63	40.104	-83.042	253	44.688	-67.792	44.688	-67.792	

(3 rows)

The output to the screen is paged, press the **space bar** for the next page, or type **q** to end.

A **where** clause lets us specify columns of interest. Note the need for double quotes for "Reporter", as it is a column name with a capital letter. Also, note the use of single quotes around 'G3ZIL' and the wd_band '30'. Reporter and wd_band parameters are of type character, hence the need for single quotes. If omitted there will be an error message.

```
wsprnet=> select * from spots where "Reporter" = 'G3ZIL' and wd_band = '30'
order by wd_time desc limit 10;
```

3.1 Export query output to a file

In this interactive mode where you are connecting to the remote logs1.wsprdaemon.org server you can download the result of a query to the current directory of your local computer using the following:

```
tutorial=> \copy (select * from spots where "Reporter" = 'G3ZIL' and wd_band
= '30' order by wd_time desc limit 3) to 'G3ZIL_spots.csv' with csv;
COPY 3
```

The \ before the `copy` signifies an export to a file on the client computer. Note that the PostgreSQL query must be within parentheses. There are limitations in the ability of the csv format to handle some aspects of PostgreSQL output, such as NULLs; full details are in the PostgreSQL documentation⁹.

The output of the query on the local computer is:

```
pi@raspberrypi:~ $ cat G3ZIL_spots.csv
2020-11-12 11:18:00,2581778396,1605179880,G3ZIL,IO90hw,-
8,10.140184,DJ6OI,JO41tp,23,0,768,268,10,
,1,30,-999.9,-
999.9,80,50.938,-1.375,268,51.646,9.625,51.646,9.625
2020-11-12 11:18:00,2581778405,1605179880,G3ZIL,IO90hw,-
9,10.140157,OZ7IT,JO65df,37,0,1028,248,10,
,1,30,-999.9,-
999.9,57,50.938,-1.375,248,55.229,12.292,55.229,12.292
2020-11-12 11:16:00,2581773016,1605179760,G3ZIL,IO90hw,-
1,10.140275,OE7WRT,JN57qg,37,0,1012,298,10,
,1,30,-999.9,-
999.9,109,50.938,-1.375,298,47.271,11.375,50.938,-1.375
```

The `wd_time` format in the first column is directly usable in Excel or other spreadsheets as date time.

Simple expressions in a query

Here is an example of a search for a particular Reporter and `wd_band` that only lists column `dB` (that is, SNR) above a threshold of -10 dB, where only `wd_time`, `CallSign` and `dB` are requested as output (`wd_time` is there by default). As `dB` is a numeric column we can use the mathematical operator `>`, and no single quotes around its numeric value -10. But single quotes are needed around '40' as `wd_band` is a text column. And a reminder, double quotes are needed around Reporter, `CallSign` and `dB` as these column name contain capitals. We've also set a time window using the self explanatory `now()` and subtracting an interval, here 6 hour (within single quotes), to give the last six hours.

```
wsprnet=# select wd_time, "CallSign", "dB" from spots where "Reporter" =
'G3ZIL' and wd_band = '40' and "dB" > -20 and wd_time > now() - interval '6
hour' limit 10;
```

3.2 Wildcards

Queries can include wildcards, where `%` matches zero or more characters or numbers. This example uses `like` and the wildcard symbol `%` for any Grid with characters JN for Reporter 'G3ZIL' and where `wd_band` = '80', remembering the quotes, and for the last 24 hours:

```
wsprnet=# select wd_time, "CallSign", "Grid", "dB" from spots where "Grid"
like 'JN%' and "Reporter" = 'G3ZIL' and wd_band = '80' and wd_time > now() -
interval '24 hour' order by wd_time desc limit 10;
```

The `_` character represents one character or number. If needed there is `not like` to exclude.

This next example shows the syntax for a query where Grid can be any in EN or FN and receive ReporterGrid any in IO or JO and `wd_band` is 30, note the required use of parentheses around the `or` pairs:

⁹ See <https://www.postgresql.org/docs/9.2/sql-copy.html>

```
wsprnet=# select wd_time, "CallSign", "Grid", "Reporter", "ReporterGrid",
"dB" from spots where ("Grid" like 'EN%' or "Grid" like 'FN%') and
("ReporterGrid" like 'IO%' or "ReporterGrid" like 'JO%') and wd_band = '30'
and wd_time > now() - interval '24 hour' order by wd_time desc limit 10;
```

3.3 Mathematical operations

Mathematical operations on one or more columns are allowed; column names within the mathematical expression (here "dB" - "Power" + 30) must be in double quotes as they include capitals. In this example we calculate and output the SNR normalised¹⁰ to a transmit power of 30 dBm (1 watt) by subtracting Power from dB and adding 30. The `as` gives the resulting column a name:

```
wsprnet=# select wd_time, "CallSign", "Grid", "Power", "dB", ("dB" - "Power"
+ 30) as "Norm_SNR" from spots where "Grid" like 'FN%' and "Reporter" =
'G3ZIL' and wd_band = '80' and wd_time > now() - interval '24 hour' order by
wd_time desc limit 10;
```

3.4 Simple statistics and how to specify a specific time interval

Simple statistics can be obtained as in these examples, having done a `count` first to check the validity of the results. Here, as an example of how to set a time interval for the query we have chosen to span approximately local night.

```
wsprnet=# select count("dB") from spots where wd_band = '60' and "Reporter"
= 'G3ZIL' and "CallSign"= 'OZ7IT' and wd_time > '2022-05-20T00:00:00Z' and
wd_time < '2022-05-21T0:00:00Z';
```

Having established that there are enough spots in the interval for meaningful statistics we form the simple arithmetic average, with `round` to round the `avg` to the nearest integer:

```
wsprnet=# select round(avg("dB")) from spots where wd_band = '60' and
"Reporter" = 'G3ZIL' and "CallSign"= 'OZ7IT' and wd_time > '2022-05-
20T00:00:00Z' and wd_time < '2022-05-21T0:00:00Z';
```

This example is for `max`, we can also use `min`

```
wsprnet=# select max("dB") from spots where wd_band = '60' and "Reporter" =
'G3ZIL' and "CallSign"= 'OZ7IT' and wd_time > '2022-05-20T00:00:00Z' and
wd_time < '2022-05-21T0:00:00Z';
```

We can calculate standard deviation with `stddev`, and here we have used `trunc` with `1` to give the result to `1` decimal places. We `cast` the result of `stddev` as `numeric` for `trunc` to work:

```
wsprnet=# select trunc(cast (stddev("dB") as numeric),1) as stddev from
spots where wd_band = '60' and "Reporter"= 'G3ZIL' and "CallSign"= 'OZ7IT'
and wd_time > '2022-05-20T00:00:00Z' and wd_time < '2022-05-21T0:00:00Z';
```

The statistics above have been evaluated over a specific, user-set time interval. Using a Timescale extension to postgresSQL we can easily obtain statistics over a series of time intervals by using `time_bucket`. In this example, we've set a `time_bucket` of 10 minutes, aliased as `ten_min`, and we're calculating the median distance within each ten minute interval using the `percentile_disc` function with `0.5` (i.e. the median, if we were after lower quartile it would be `0.25`, and `0.75` for upper quartile), aliased as "median distance km" (note the use of triple quotes here):

```
wsprnet=# SELECT time_bucket('10 minutes', wd_time) AS ten_min,
percentile_disc(0.5) within group(order by distance) as ""median distance
(km)"" FROM spots WHERE "Reporter"= 'G3ZIL' AND wd_band = '40' and wd_time
```

¹⁰ We are, of course, aware of the pitfalls of any attempt at normalisation, power may not be correctly reported and the actual radiated power and directional characteristics of the transmit and receive antennas are not often well known.

```
> '2022-05-20T10:00:00Z' and wd_time < '2022-05-20T12:00:00Z' GROUP BY
ten_min order by ten_min desc;
```

3.5 Query on the azimuth angle at the receiver.

wsprnet.org calculates the azimuth at the transmitter, azimuth, as the initial bearing of the path to the receiver. However, wsprnet.org does not calculate the azimuth on arrival at the receiver. It is the arrival azimuth, wd_rx_az, that is needed if one is looking to use WSPR spot information to help evaluate the directionality of receive antennas. Except for special cases, such as receiver and transmitter on the same longitude, or where the path distance is less than a few hundred km, azimuth and wd_rx_az are not simply 180° apart. This example is for receiver KFS where band is '40' and we search for transmitters from azimuths between 60° and 90° where distance is between 1000 and 2000 km.

```
wsprnet=# select wd_time, "CallSign", "Grid", distance, wd_rx_az, "dB" from
spots where "Reporter" = 'KFS' and wd_band = '40' and distance > 1000 and
distance < 2000 and wd_rx_az > 60 and wd_rx_az < 90 and wd_time > '2022-05-
20T10:00:00Z' and wd_time < '2022-05-20T12:00:00Z' limit 10;
```

3.6 Query on the vertex latitude

We have added a calculation of the latitude and longitude of the position of the vertex of the great circle path between transmitter and receiver. The vertex is the most northern or southern position. In this example we select a series of parameters where the band is '40' and the vertex latitude is > 60 (i.e. above 60°N), and where the vertex is at a higher latitude than the latitude of either the transmitter or receiver. Where the path is predominantly north-south the vertex is most likely to be at either the receiver or transmitter. Queries on the vertex lat and lon may be informative when looking at paths that are near to, or cross, the northern and southern Auroral Ovals:

```
wsprnet=# select "CallSign", "Reporter", wd_tx_lat, wd_tx_lon, wd_rx_lat,
wd_rx_lon, wd_v_lat, wd_v_lon, "dB" from spots where wd_band = '40' and
wd_v_lat > 60 and (wd_v_lat > wd_rx_lat and wd_v_lat > wd_tx_lat) and wd_time
> '2022-05-20T10:00:00Z' and wd_time < '2022-05-20T12:00:00Z' limit 10;
```

3.7 Select only spot lines with distinct field entries

In this example we want to list only distinct (unique) CallSigns heard by G3ZIL on 40 m in the last hour, using the distinct on clause. Note the order by must match the distinct on field:

```
wsprnet=# select distinct on ("CallSign") * from spots where
"Reporter"='G3ZIL' and wd_time > now() - interval '1 hour' and wd_band='40'
order by "CallSign" limit 10;
```

3.8 Using subqueries: Order by a different column

In this example VK7JJ needed a list of distinct CallSigns as in 3.7 but required them to be ordered by wd_time - a feature not directly available in postgresSQL, as the first order by column has to be the distinct column. The solution was to use a subquery for the distinct and an outer query for the order by wd_time. Note that the output of the subquery needs an alias; here it is temp:

```
wsprnet=# select * from (select distinct on ("Reporter") * from spots where
wd_band = '40' and wd_time > '2022-05-20T00:00:00Z' and "CallSign" = 'OZ7IT'
order by "Reporter", wd_time desc) temp order by wd_time desc, "Reporter"
limit 10;
```

3.9 Use of Joins

As postgresSQL is a full relational database it provides the ability to join two or more tables.

First, we describe a variant - the 'self-join' - where a table is joined to itself, best explained using an example. Here we output wd_time, CallSign, and dB (SNR) at two Reporters for instances where both Reporters spot the same CallSign in the same two-minute interval. The `from spots s1` gives us an alias `s1` for one instance of our table `spots` and `inner join spots s2` gives us a second alias `s2`. As G4HZX only reports on 40 there is no need to check the bands match. We use these aliases as prefixes to the column names to refer to the two instances of our single spots table:

```
wsprnet=# select s1.wd_time, s1."CallSign", s1."dB" as "G3ZIL.dB", s2."dB"
as "G4HZX.dB" from spots s1 inner join spots s2 on s1.wd_time = s2.wd_time
and s1."CallSign" = s2."CallSign" and s1."Reporter" = 'G3ZIL' and
s2."Reporter" = 'G4HZX' where s1.wd_time > now() - interval '1 hour' order by
s1.wd_time desc limit 10;
```

In this example we calculate the SNR difference for the two Reporters and make sure the bands are the same:

```
wsprnet=# SELECT s1.wd_time, s1."CallSign", s1."MHz", (s1."dB" - s2."dB") as
""SNR_difference G3ZIL-G4HZX"" FROM spots s1 join spots s2 on s1.wd_time =
s2.wd_time and s1."CallSign" = s2."CallSign" and s1."Reporter" = 'G3ZIL' and
s2."Reporter" = 'G4HZX' and s1.wd_band='40' and s2.wd_band='40' where
s1.wd_time > now() - interval '1 hour' order by s1.wd_time desc limit 10;
```

Using a left join lets us output those CallSigns heard by one station and not another within the same two-minute interval on a band, in this case, those heard by G3ZIL but not by G4HZX:

```
wsprnet=# SELECT
s1."wd_time" AS "time",
s1.distance as km,
s1."dB",
s1."Reporter",
s1."CallSign"
FROM spots s1
left join spots s2
on s1.wd_time = s2.wd_time
and s1."Reporter" = 'G3ZIL' and s2."Reporter" = 'G4HZX'
and s1.wd_band='40' and s2.wd_band='40'
and s1."CallSign" = s2."CallSign"
where s1."Reporter" = 'G3ZIL' and s1.wd_band='40' and s1.wd_time > now() -
interval '1 hour' and s2."CallSign" is null order by s1.wd_time desc limit
10;
```

4. Queries from tutorial database: wsprdaemon_spots_s, wspraemon_noise and kp

Whereas our database wsprnet spots table contains data from all those reporting to wsprnet.org our database tutorial only contains data uploaded directly by users of WsprDaemon¹¹. The four main reasons for this separate data route are:

1. Allows for immediate data upload even if wsprnet.org is down.
2. Allows for additional spot-related data columns derived from the wsprd program that are not uploaded to, or handled by, the database at wsprnet.org.
3. Allows for a 'receiver' designator, useful for sites that report spots from multiple receivers but under one reporting identifier, usually a callsign, e.g. to compare the results between different receivers and or antennas.
4. Allows for noise estimates data to be uploaded, stored and queried alongside spots data.

In addition, the tutorial database contains tables of ancillary data, currently only kp, the geomagnetic disturbance index, is available.

¹¹ See wsprdaemon.org

4.1 Table `wsprdaemon_spots_s`

The full list of data columns in table `wsprdaemon_spots_s` is shown in Annex A. While some column names are different to table `spots` in our database `wsprnet` by now the reader will be sufficiently familiar to not need a full set of examples that parallel those in section 3. Here we will concentrate on those queries that cannot be run against the `wsprnet spots` table.

First, connect to database tutorial:

```
tutorial=# \c tutorial
```

In this example we calculate SNR difference between two receivers and two different callsigns for a set span of spot distances reported by the first callsign, together with the `tx_call`, distance and azimuth at the first callsign. We are using KFS and KPH as the reporters, with specific receiver KIWI_NW at KFS and KPH_HF_76 at KPH, on 40 m and for spots between 0 and 5000 km distant.

```
tutorial=# SELECT
    s1.time AS time, s1.tx_call, s1.km, s1.rx_az,
    (s1."SNR" - s2."SNR" ) as "dB_difference"
from wsprdaemon_spots_s s1
join wsprdaemon_spots_s s2 on s1.time = s2.time
and s1."tx_call"= s2."tx_call"
and s1."rx_id"= 'KFS' and s2."rx_id" = 'KPH'
and s1."receiver"='KIWI_NW' and s2."receiver"='KPH_HF_76'
and s1.band='40' and s2.band='40' and s1.km> 0 and s1.km<5000
and s1.time > now() - interval '1 hour'
order by time desc limit 10;
```

In this next example drawing on a parameter from the `wsprd` decoder not sent to `wsprnet.org` we find the median SNR at G3ZIL on 40 m for the past 24 hours for the spots decoded by the Fano decoder (`osd_decode=0`) and, when the Fano cannot produce a result, by the Ordered Statistics Decoder (`osd_decode=1`), with a count of spots decoded by each. There is no median function as such in PostgreSQL and so we use the `percentile_disc` function with 0.5 (i.e. 50%). This snippet also shows the use of the `filter` clause, note that table `wsprdaemon_spots_s` has been aliased to `a` for brevity, and we've given headings for the derived variables:

```
tutorial=# SELECT osd_decode as decode_type,
    COUNT(a.osd_decode),
    percentile_disc(0.5) within group (order by a."SNR")
    filter (where osd_decode = 0) as Median_SNR_Fano,
    percentile_disc(0.5) within group (order by a."SNR")
    filter (where osd_decode = 1) as Median_SNR_Ordered_Statistics
FROM wsprdaemon_spots_s a
WHERE rx_id='G3ZIL' and band='40' and time > now()-interval'1 day'
GROUP BY a.osd_decode;
```

resulting in:

decode_type	count	median_snr_fano	median_snr_ordered_statistics
0	4857	-12	
1	1173		-25

4.2 Table `wsprdaemon_noise_s`

The full list of data columns in table `wsprdaemon_noise_s` is shown in Annex A. Details of the rms and c2 (FFT) algorithms used to estimate noise have been published in the Sept/Oct 2020 issue of QEX¹².

¹² Gwyn Griffiths, Rob Robinett and Glenn Elmore "Estimating LF-HF band noise while acquiring WSPR spots". QEX, ARRL, Sept-Oct 2020.

The noise time series are far more amenable to graphical presentations, e.g. using Grafana. Nevertheless, the following example is of some interest. At G3ZIL the KiwiSDR receiver G3ZIL_1 has an antenna switch board from Glenn Elmore N6GN at its input; as part of a separate noise-measurement script the antenna input is switched to a 50 ohm terminator during the interval between WSPR transmissions. In the example query below we see that the c2_level FFT noise level estimate is reading noise from the antenna, as it is an estimate of the 30% of the lowest value Fourier coefficients in, and adjacent to, the WSPR band throughout the transmission period. However, the rms estimator looks for the quietest 50 milliseconds during the gap between WSPR transmissions, in this case reading the noise level at the terminated input to the KiwiSDR - a useful systems check.

```
tutorial=# select time, c2_level, rms_level from wsprdaemon_noise_s where
site='G3ZIL' and receiver='G3ZIL_1' and band='40' and time > now() - interval
'10 minutes' order by time desc;
```

time	c2_level	rms_level
2020-11-19 14:10:00	-131.2	-155.81
2020-11-19 14:08:00	-131.34	-155.62
2020-11-19 14:06:00	-131.63	-155.52

The next example is a simple count of the number of times the KiwiSDR overload counter `ov` is greater than 0:

```
tutorial=# select count (ov) from wsprdaemon_noise_s where site='G3ZIL' and
ov >0 and time > now()-interval'1 day';
```

```
count
```

```
1
```

and in contrast, for KFS:

```
tutorial=# select count (ov) from wsprdaemon_noise_s where site='KPH' and ov
>0 and time > now()-interval'1 day';
```

```
count
```

```
427
```

Annex A. Description of data within columns of tables in wsprnet and tutorial

For ease of reference the column names and types are repeated here. As a reminder:

- If the Column name contains a capital letter, e.g. Reporter, PostgreSQL requires it to be within double quotes, e.g. WHERE "dB" > 10
- If the Type is text or character PostgreSQL requires it to be within single quotes, e.g. WHERE "Reporter"='G3ZIL'. Note that column wd_band is of type text, as our initial thought was to have entries for 60 and 60eu and 80 and 80eu but this has not been implemented, thus use WHERE wd_band='80'.

Database wsprnet - table spots:

Column	Type
wd_time	timestamp without time zone
Spotnum	bigint
Date	integer
Reporter	text
ReporterGrid	character(6)
dB	smallint
MHz	double precision
CallSign	text
Grid	character(6) N
Power	smallint
Drift	smallint
distance	smallint
azimuth	smallint
Band	smallint
version	character(10)
code	smallint
wd_band	text
wd_c2_noise	real
wd_rms_noise	real
wd_rx_az	real
wd_rx_lat	real
wd_rx_lon	real
wd_tx_az	real
wd_tx_lat	real
wd_tx_lon	real
wd_v_lat	real
wd_v_lon	real

Columns with data from wsprnet.org

wd_time	UTC time of the start of the two minute interval for a WSPR cycle and is in the format 2018-11-19 18:30:00
Spotnum	An unique identifier assigned by wsprnet to an incoming spot as they are received at wsprnet.org. ¹³
Date	Unix epoch format in seconds, 1604915400 use converter, e.g. at https://www.epochconverter.com/ to convert to human readable date.
Reporter	Identifier as provided by the uploader of the WSPR data, e.g. KD0J, and may include a suffix such as /A, /P etc.

¹³ Do not assume Spotnum to increment uniformly with time, a spot that arrives late, e.g. an Internet outage at the reporter, will have a Spotnum issued at the time it is received at wsprnet.org, and not related to the time the spot was decoded.

ReporterGrid	Maidenhead grid locator of the Reporter. It will be 6 characters if wsprnet.org has that information.
dB	Signal to noise ratio (SNR) as estimated within the decoder within WSJT-X
MHz	Frequency in MHz as seen at the receiver by adding the measured audio frequency to the 'dial' frequency for the selected band, reported to 6 decimal places, i.e. 1Hz.
CallSign	Identifier for the transmitting station, as decoded from the WSPR transmission.
Grid	Maidenhead grid locator of the Reporter. It will be 6 characters if wsprnet.org has a record for that CallSign.
Power	Power reported by the transmitting station in dBm. 30 dBm = 1 Watt.
Drift	Drift of the transmitted signal in Hz over the duration of the WSPR message seen by the receiver (which may also drift).
distance	Distance in km calculated from the receiver and transmitter grid squares. Accuracy will be best with two 6-character locators.
azimuth	Azimuth in degrees of the receiver as seen at the transmitter assuming a great circle short path. Clockwise from north.
Band	This is the band designator assigned by wsprnet.org as the frequency in MHz as an integer except that 136 kHz is listed as -1. Note that there are spurious entries, e.g. 49 with 7074, 41 with 2 (out of 238 million spots).
version	Where available, the version of the WSJT-X software in use.
code	A mode designator code: 1 is 'standard' WSPR2 and the new mode FST4W-120, 2 is WSPR15 and FST4W-900, 4 is FST4W-300 and 8 is FST4W-1800.
<i>columns of data derived during preprocessing by WsprDaemon server from the above.</i>	
wd_band	Determined from the frequency by WsprDaemon preprocessing software, expressed in metres for 2200 - 2 metres, with 70cm and 23cm as 70 and 23. Where an appropriate band cannot be determined it is listed as 9999 (26,057 in 238 million).
wd_c2_noise & wd_rms_noise	These two noise fields are set to absent data currently, i.e. -999.0.
wd_rx_az	Azimuth in degrees of the incoming signal at the receiver assuming a great circle short path from the transmitter. Clockwise from north.
wd_rx_lat	Latitude in degrees of the receiver calculated from Grid. Negative is south. These numeric latitude and longitude fields allow for numeric SELECT statements in postgresSQL queries.
wd_rx_lon	Longitude in degrees of the receiver calculated from the rx_grid. Negative is west.
wd_tx_lat	Latitude in degrees of the transmitter calculated from the tx_grid. Negative is south.
wd_tx_lon	Longitude in degrees of the transmitter calculated from the tx_grid. Negative is west.
wd_v_lat	Latitude in degrees of the vertex of the great circle path between receiver and transmitter. The vertex is the most northerly, or southerly, point on the path. There are, of course, instances where the vertex is at the receiver or transmitter.

This is calculated by WsprDaemon and can be useful when studying paths that are near to or within the polar Auroral Ovals.

wd_v_lon Longitude in degrees of the vertex of the great circle path between receiver and transmitter.

Database tutorial table wsprdaemon_noise_s

Column	Type
time	timestamp without time zone
site	text
receiver	text
rx_grid	text
band	text
rms_level	double precision
c2_level	double precision
ov	integer
seqnum	bigint

time UTC time of the start of the two minute interval for a WSPR cycle and is in the format 2018-11-19 18:30:00

site Usually the callsign of the reporting station, e.g. N6GN.

receiver It is not uncommon for WsprDaemon users to use more than one receiver. In some cases they may use a separate site name e.g. N6GN/K to distinguish different receivers and or different antennas. In other cases, this receiver column allows the user to use a secondary identifier of their own choice, e.g. N6GN has used GN0, GN1, GN2, GN3. As styles and usage vary, and there is no metadata available, users of this table should consult the reporting station for details.

rx_grid Maidenhead grid locator of the Reporter. It should be 6 characters.

band The band in metres; a text column that includes separate entries for 60, 60eu, 80 and 80eu. At least one site uses the noise estimation capability to estimate the signal level of standard frequency stations, including WWVB, CHU-3, WWV-10. For a full list use:

```
select distinct band from wsprdaemon_noise_s;
```

rms_level Noise estimate from the wsprdaemon RMS algorithm, essentially the RMS value of the quietest 50ms within the gap between WSPR transmissions. Units are dBm in 1Hz, however the absolute value will depend on the offset calibration provided at the receiver (same goes for c2_level).

c2_level Noise estimate from the wsprdaemon FFT algorithm using the c2 decimated samples file produced by wsprd.

ov For the KiwiSDR a count of the number of ADC overload events within the two-minute reception interval.

seqnum An unique auto-incrementing sequence number.

Database tutorial table wsprdaemon_spots_s

This was our first table of WSPR spots and we chose column names that seemed appropriate, but have ended up being different to the spots table in our wsprnet table (which we implemented later with names requested by the wsprnet administrators). Nevertheless, the data fields should be easily understood from the names and reference to the wsprnet spots table described above,

until the `sync_quality` column. `Sync_quality` and subsequent columns and fields available within the `wsprd` decoding program but not sent to `wsprnet.org`. As the `WsprDaemon` program does have access to these fields and there is no impediment to their transmission to, and incorporation in, the `wsprdaemon_spots_s` table they are included in case they may be of interest.

Column	Type
<code>time</code>	timestamp without time zone
<code>band</code>	text
<code>rx_grid</code>	text
<code>rx_id</code>	text
<code>tx_call</code>	text
<code>tx_grid</code>	text
<code>SNR</code>	double precision
<code>c2_noise</code>	double precision
<code>drift</code>	double precision
<code>freq</code>	double precision
<code>km</code>	double precision
<code>rx_az</code>	double precision
<code>rx_lat</code>	double precision
<code>rx_lon</code>	double precision
<code>tx_az</code>	double precision
<code>tx_dBm</code>	double precision
<code>tx_lat</code>	double precision
<code>tx_lon</code>	double precision
<code>v_lat</code>	double precision
<code>v_lon</code>	double precision
<code>sync_quality</code>	integer
<code>dt</code>	double precision
<code>decode_cycles</code>	integer
<code>jitter</code>	integer
<code>rms_noise</code>	double precision
<code>blocksize</code>	integer
<code>metric</code>	integer
<code>osd_decode</code>	integer
<code>receiver</code>	character varying
<code>nhardmin</code>	integer
<code>ipass</code>	integer
<code>seqnum</code>	bigint
<code>mode</code>	smallint
<code>ov_count</code>	integer

`sync_quality` Our conjecture: A measure of how well the incoming sync symbol sequence is synchronised to the sync vector sequence timed by the receiver's clock. The raw variable is on a scale of 0 to 1 (but was recorded as the integer of 10 times the raw value in V2.1), hence care needs to be taken if comparing across versions.

`dt` This is the time difference between the actual start of the audio signal presented to `wsprd` and 2 seconds past an even minute as perceived by the clock at the receiver. `wsprdaemon` records the full 10ms resolution value. There are (possibly at least) four main causes for a non-zero reading:

1. A time offset from UTC in the clock at the transmitter.
2. A time offset from UTC in the clock at the receiver.
3. A delay (latency) at the transmitter between the clock-commanded start of transmission and the actual transmission.

4. A delay (latency) at the receiver between the clock-commanded start of reception and the actual audio file start.

decode_cycles	This is the number of cycles taken for the Fano (default) or Jelinek (if selected) decoder to produce an output. The default maximum number of cycles is 10,000, but can be set with the -C option on calling wsprd. In practice the most common value for decode_cycles (mode) is 1; for example, decode_cycles was 1 for 10212 out of 13285 spots for KD2OM on 40m, which is about 77%. This field may also be referred to as Fano iterations-per-bit.
jitter	Applied as a fine adjustment to the well-known dt time shift value. Centred on zero, values are in steps of 8 between -64 and +64. and are tried in the sequence 0, -8, 8, 16 etc. For 91092 spots at KD2OM 85645 i.e. 94% jitter was 0, 1.5% were at -8 and 1% at +8, and all allowed values to +/-64 were present. While no unit is given, it is certainly time, and quite likely to be the sampling interval, that is 1/375Hz or 0.26667 seconds, making each step of 8 equivalent to 0.021333 seconds.
blocksize	A parameter controlling the detection of individual symbols in the wsprd demodulator. Allowable values are 1, 2, 3, units are symbols. A value of 1 signifies that the first try using non-coherent detection of individual symbols was successful (sufficient), this is equivalent to the original wsprd demodulator. Blocksizes of 2 and above means that that many symbols are decoded at once; from the source code, "Longer block lengths require longer channel coherence time". Most of the time blocksize will be 1, as an example, of 90271 spots at KD2OM 88214 were blocksize 1, 1575 at 2 and 482 at 3.
metric	This is an output from the Fano (default) or Jelinek (if selected in the call to wsprd.c) decoder. In Information Theory, metric is a measure of the "closeness of a path to the received sequence". The distribution of metric for 92513 spots at KD2OM is shown below. There is a broad asymmetric distribution at about 570. The singular peak at 810 is because that value is when the Fano (or Jelinek) algorithm has failed. In that case, the Ordered Statistic Decoder (OSD) is executed. It will come up with its decode, and if accepted, the osd_decode flag (see below) will be set to 1. This happened (osd_decode flag set to 1) in 98.8% of instances when metric was 810 in this test case with KD2OM spots.
osd_decode	Flag, either 0 or 1. If 0 then either the Fano (default) or the stack (Jelinek, only if wsprd is called with option -J) decoding algorithms have been used to decode. These algorithms can end without a decode being produced. If 1 then the Ordered Statistics Decoder (OSD) has been used. This decoder will always produce a decode - but of course it can be wrong. Therefore, wsprd only accepts an OSD decode if the tx_call it produces is already in the hash table having been decoded previously by the Fano or Jelinek algorithms.
receiver	A user-supplied designator, exactly as used in the wsprdaemon_noise_s table.
nhardmin	A count of the hard errors from the Fano (default) or the stack (Jelinek, only if wsprd is called with option -J) decoding algorithm. Not yet clear what it means if the Ordered Statistics Decoder (OSD) has been used.
ipass	A flag determined by user set options and the number of passes required to effect a decode. If wsprd is called with option -s (which it is not in wsprdaemon) this is the single pass (now very old) mode, so ipass can never be greater than 1, but (I think) can still be 1 if only a single pass was needed. If option -B was set (which it is not in wsprdaemon) then block demodulation is disabled, only single-symbol noncoherent demodulation is used, and npass can take the values 1 or 2. Otherwise, npass may take a value of up to 3.

seqnum An auto-incrementing sequence number.

mode An integer that refers to the transmission protocol and duration:

- 2 WSPR-2
- 3 FST4W-120
- 6 FST4W-300
- 16 FST4W-900
- 31 FST4W-1800

Note that wsprnet.org gives mode=2 for WSPR-2 and FST4W-120 and 15 for WSPR-15 and FST4W-900.

ov_count For the KiwiSDR a count of the number of ADC overload events within the two-minute reception interval.

Annex B. Accessing the WsprDaemon database using node.js

This Annex is a note on how to use node.js to access the wsprnet database table spots on the WsprDaemon server courtesy of Phil Barnard, VK7JJ, phil 'at' perite.com to whom we are

The code examples provided assume a working knowledge of node.js and its package manager NPM.

1. Open an ssh connection in the terminal you use to connect to node.js
2. Use NPM to install the pg package <https://www.npmjs.com/package/pg> - documentation of the pg package with examples: <https://node-postgres.com/>
3. The three sample scripts provided with this doc. all accept a simple postgres text query.

The pgclient and pgpool scripts are able to be run directly in a terminal via ssh and return their results to the console. pgweb is for use with the node express web server module.

Database access details as required by db_config are available from WSPR Daemon.

NOTE: the script files are bundled as .txt files but need to be changed to .js before being run.

pgclient.js

A pg client is designed to be used for normal day to day personal queries of the db and is fine for a personal web server.

Each client query connects to the db, authenticates, passes the query to the db and receives either an error message or a JSON results.rows string and then disconnects from the db.

The overhead is inconsequential and queries are fast and efficient, there is no persistent connection. The client_query function is as straightforward as possible with direct access to errors and results.

pgpool.js

For use with a heavy duty server. A pool of of pre-authenticated clients is assigned by the WSPR Daemon db server when the node.js server starts and is used for the life of the node server.

The pool.query method runs a query on the first available idle client and returns its result. Each client is released back into the pool automatically.

Authentication occurs with the pool assigned as the web server starts up. The pool is only released when the server is shut down.

The pool_query function is as straightforward as possible with direct access to errors and results.

pgweb.js

For use with a public facing web server and assumes the use of node express.

The web_query function is effectively copied and pasted from <http://wsprd.vk7jj.com>

Express (code not included) accepts Javascript fetch queries from the browser and calls the web_query function for each query.

The web_query function takes two arguments, a query string and a response object passed to it by express. It returns JSON stringified results.rows and errors to the web client using promises with a catch.

A zip folder with these notes and the three script files can be downloaded from

http://wsprdaemon.org/VK7JJ_node_pg_how_to.zip

Annex C. Bash script to read basic spot data from database wsprnet table spots

This bash script was written in conjunction with Jim Lill WA2ZKD to access spots data for his real-time WSPR spots analysis website <http://www.jimlill.com:8088/>

Invoked as the script name with two arguments, the unique, wsprnet.org assigned, Spotnum at which to start listing (SPOTNUM_START) and the number of rows required (N_SPOTS) it returns the latest N_SPOTS rows to a file which is then sent to the standard output. Bash variable `query_1` is formed within a string with single quotes allowing double quotes internally for the column names with upper case letters. Bash variable `query` if formed from concatenating `${query_1}`, the required start spotnum `${SPOTNUM}` and number of rows in variable `${N_SPOTS}` in an expression with double quotes that does not upset the quoting within `${query_1}` itself, and uses a `\` to escape the double quotes around the column name `Spotnum`.

Jim Lill added the awk lines to decode the type of WSPR transmission from the `code` data. This decode to transmission type is only valid for spots after around 19 October 2020 when the code column was repurposed by the wsprnet.org team to indicate the new modes in WSJT-X V2.3.0-rc1.

```
#!/bin/bash
# revised 22 May 2022
WSPRDAEMON_TIMESCALE_HOST="logs1.wsprdaemon.org"
SPOTNUM_START=$1
N_SPOTS=$2

query_1='select wd_time, "CallSign", "MHz", "dB", "Grid", "Power", "Reporter",
"ReporterGrid", code, "Spotnum" from spots $
query="${query_1} ${SPOTNUM_START} order by "\"Spotnum\"\"" asc limit
${N_SPOTS};'

PGPASSWORD=WA2ZKD psql -U wdjim -d wsprnet -h ${WSPRDAEMON_TIMESCALE_HOST} -A
-F, -t -c "$query" 2> wdquery.log | \
awk -F, 'BEGIN { OFS="\t" }
$9 == 0 { $9 = "UNK" }
$9 == 1 { $9 = "WSPR-2" }
$9 == 2 { $9 = "WSPR-15" }
$9 == 4 { $9 = "FST4W-300" }
$9 == 8 { $9 = "FST4W-1800" }
{ printf ("%s %-12s\t %.6f\t %s\t %s\t %s\t %-
12s\t %s\t %s\t %s\t %s\n", $1, tolower($2), $3, $4, $5, $6, $7, $$
cat smscrape.new
```

Typical time taken for 10000 rows into a file:

```
Gwyn-2:Desktop gwg$ time ./lill.sh 4297254651 10000 >/dev/null
real 0m3.164s
user 0m0.150s
sys 0m0.037s
```

Annex D. Skeleton of a Python script to read the WsprDaemon wsprnet database

This skeleton shows how to use the adapter psycopg2 on a local computer to read data from the logs1.wsprdaemon.org wsprnet database table spots and send to the standard output. This script needs psycopg2 and PostgreSQL to be installed (see section 1).

The line `select_sql` in the example `ts_spot_read_1.py` can be changed to whatever query you wish to use. However, as it stands, the line cannot contain a column name with a capital letter as that would need double quotes that would conflict with the required double quotes around the entire `select_sql` line. The way around this is to split the sql declaration into two or more lines, with the line needing an upper case column name in single quotes and then concatenating the lines into a single query line, as in the example `ts_spot_read_2.py`.

Both skeletons are invoked with a single argument, the number of rows to return, as in the two examples below.

```
pi@ZIL-Kiwi:~ $ cat ts_spot_read_1.py
#!/usr/bin/python
# ts_spot_read.py      Gwyn Griffiths March 2020
# See https://wiki.postgresql.org/wiki/Psycopg2_Tutorial for further details

import psycopg2
import sys

N_RECS=sys.argv[1]
select_sql="SELECT * from spots ORDER BY wd_time DESC LIMIT " + str(N_RECS)
print(select_sql)

try:
    # connect to the PostgreSQL database
    print("about to connect")
    conn = psycopg2.connect("dbname='wsprnet' user='wdread'
host='logs1.wsprdaemon.org' password='JTWSPR2008'")
    # create a new cursor
    cur = conn.cursor()
    cur.execute(select_sql)
    rows = cur.fetchall()
    for row in rows:
        print (row)
    # close communication with the database
    cur.close()
except:
    print ("Unable to connect to the database")
finally:
    if conn is not None:
        conn.close()
```

```
Gwyn-2:Desktop gwg$ python3 test.py 2
SELECT * from spots ORDER BY wd_time DESC LIMIT 2
about to connect
(datetime.datetime(2022, 5, 22, 16, 18), 4297441812, 1653236280, 'K9REO',
'EN53gb', -19, 14.097037, 'VE2DJK', 'FN35ig', 37, 0, 1310, 265, 14, '2.5.4
', 1, '20', -999.9, -999.9, 74.0, 43.062, -89.458, 265.0, 45.271, -73.292,
45.271, -73.292)
(datetime.datetime(2022, 5, 22, 16, 18), 4297441816, 1653236280, 'K9REO',
'EN53gb', -23, 14.09712, 'W6TOM/1', 'FN42oa', 13, 0, 1531, 281, 14, '2.5.4
', 1, '20', -999.9, -999.9, 88.0, 43.062, -89.458, 281.0, 42.021, -70.792,
43.102, -86.46)
```

Note that the output lines are tuples.

Annex E. Redacted

Annex F. Octave route

The author would be very pleased to hear from anyone that has tried and failed or succeeded with implementing the PostgreSQL package with Octave on Ubuntu.

The latest package release can be found at: octave.sourceforge.io/database/index.html

Ubuntu

Installation instructions are at: wiki.octave.org/Database_package. However, Glenn Elmore N6GN found this approach to work when trying `pkg install` within Octave did not:

```
sudo apt-get update -y
sudo apt-get install -y octave-struct
```

However, despite having the necessary `pg_config` file in directories:

```
/usr/lib/postgresql/12/bin/pg_config
/usr/bin/pg_config
```

and using `addpath()` to point to `pg_config` the database package would not install within Octave via

```
pkg install -forge database
```

Windows 10

Glenn Elmore N6GN found the following approach, within Octave V4.4, worked

```
pkg install -forge struct
pkg install -forge database
```

MacOS

The author is running Octave V6.0 [this is needed for features that make generating a video from single image frames a one-script process]. The macOS version is available as a dmg file at:

<https://github.com/octave-app/octave-app/releases/tag/v6.0.90-rc1>

Noting the dependencies at octave.sourceforge.io/database/index.html, the step within Octave:

```
pkg install -forge struct
```

runs, although with warnings. However, the step:

```
pkg install -forge database
```

initially failed as it could not find the `pg_config` file (PostgreSQL API).

This was corrected by copying the 'missing' `pg_config` file from a PostgreSQL directory to the directory where the `pkg` script was looking, i.e.

```
cp /usr/local/Cellar/libpq/13.1/bin/pg_config /Applications/Octave-6.0.90.app/Contents/Resources/usr/Cellar/octave-octave-app@6.0.90/6.0.90/bin/pg_config
```

The `pkg install -forge database` script now ran, although with warnings, nevertheless the database package installed and could be loaded using the following command at the Octave prompt:

```
>> pkg load database
```

Octave script to demonstrate database connection

The following Octave script `connect_wd_wsprnet.m` sets the connection parameters as variables for the `pq_connect` line to return the latest row from the `spots` table and then prints it as a data structure `struct` and then as a cell array:

```

% Bare bones Octave script to demonstrate access to the WsprDaemon
% wsprnet database table spots % Gwyn Griffiths G3ZIL November 2020
% Tested on Matlab V4.4 and on V6.0
% Follow notes at wiki.octave.org/Database_package and
octave.sourceforge.io/database/index.html
% for database package installation notes
% load the installed database package
pkg load database;
% set up default connection parameters
dbname="wsprnet";
host="logs1.wsprdaemon.org";
port="5432";
user="wdread";
password="JTWSPR2008";

% connect to the database
conn = pq_connect (setdbopts ("dbname", dbname, "host", host, "port", port,
"user", user, "password", password));

% get the latest single row into datastructure spots

struct = pq_exec_params (conn, "select * from spots order by wd_time desc
limit 1;");

spots=getfield(struct,'data'); %produces cell array

% close the connection
pq_close(conn);
% print out the structure, which has the column names etc and the data
struct

% print out the cell array
spots

```

The octave script below access the WsprDaemon wsprnet database table spots and uses the `m_map` toolkit¹⁴ to plot, on a great circle map, the locations of spots heard in the last hour. It reads in the band, but does nothing with it, but it does show how a character column representing numbers should be handled. [*There looks to be an issue with `m_grid` in Octave V6.0 due to use of `char(176)` for the degree sign ° causing a UTF-8 error; it was fine in Octave 4.4. If you are confident, you can change occurrences of `char(176)` in `m_grid` to `char(111)`, lower case o, which is what I did here*].

```

% Program to illustrate Octave connection to WsprDaemon wsprnet database
table spots
% with a postgresSQL query that shows how to handle double quotes around upper
case
% together with the simple use of the m_map package to draw a great circle
world map and plot
% location of stations heard
% Gwyn Griffiths G3ZIL 17 November 2020 V1.0
% This version using Octave 6.0 on a macbook pro

% m_map package available from https://www.eoas.ubc.ca/~rich/map.html
%
% load the installed database package

```

¹⁴ See <https://www.eoas.ubc.ca/~rich/map.html>

```

pkg load database;

% set up connection parameters
dbname="wsprnet";
host="logs1.wsprdaemon.org";
port="5432";
user="wdread";
password="JTWSPR2008";

% connect to the database
conn = pq_connect (setdbopts ("dbname", dbname, "host", host, "port", port,
"user", user, "password", password));

% get the last hour of time, lat, lon and band of stations heard by G3ZIL
from spots
% into datastructure struct
% Note how we deal with escaping the double quotes, e.g. around "Reporter"
% which is different to Bash, here we do not need to end previous text with "

struct = pq_exec_params (conn, "select wd_time, wd_tx_lat, wd_tx_lon, wd_band
from spots where \"Reporter\"='G3ZIL' and wd_time > now() -interval \"1
hour\" order by wd_time desc;");

% convert the data parts of the structure into a cell array
spots=getfield(struct,'data'); %produces cell array

% close the connection
pq_close(conn);

% for loop to move the data from the cell array into individual variable
vectors
n_spots=rows(spots); % number of rows in the dataset, i.e. in last
hour
for i=1:n_spots
    t_lat(i)=spots{i,2}; % same ordering as in the postgresQL select
    t_lon(i)=spots{i,3};
endfor

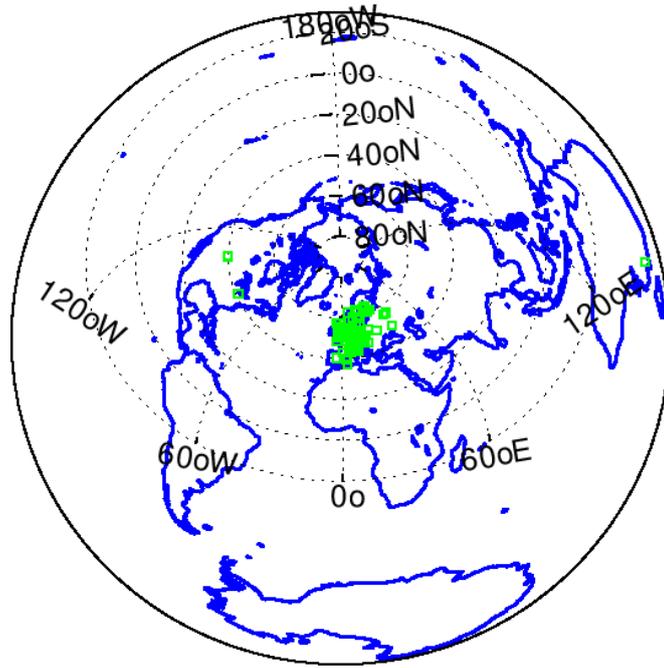
% as the band values are of type character we need to convert to decimal
band=base2dec(spots(:,4),10);

% plot lat and lon of stations heard using the m_map toolbox
% in m_proj use the azimuthal equidistant, i.e. great circle map lat and lon
specify
% the centre of the map 'rad' specifies the radius in degrees, 180 would be
all globe
% m_coast gives us a simple coastline, m_grid gives lat lon gridlines
% the function m_ll2xy changes rectangular coordinates lat lon to thos
suitable for the projection
% the line function plots the data with the style parameters given
% print saves the graphics file

m_proj('azimuthal equidistant', 'lon',[-1], 'lat', [50], 'rad', [160], 'rec',
'circle');
m_coast('linewidth',1,'color','b');
m_grid('ytick', [-20 0 20 40 60 80], 'fontsize', 12);
[M,N]=m_ll2xy(t_lon, t_lat);
line(M,N, 'linestyle', 'none',
'marker','square','markersize',3,'color','green');
print( gcf, '-dpng', fullfile(sprintf('map.png')));

```

See example map output below.



Annex G. The briefest of introductions to using WsprDaemon with R

R is a powerful programming language¹⁵ for "*statistical computing and graphics*". This Guide's author has only the most basic skills with R; he is not alone¹⁶ in considering the language hard to learn. Consequently, I was delighted that Andi Fugard (MOINF) shared a comprehensive set of examples on how to query the WsprDaemon wsprnet database spots table using R on their website at <https://inductivestep.github.io/WSPR-analysis/>

The following notes begin with how to obtain R, installing the packages required to run Andi's examples and add a few further examples. To get an impression of the enormous range of graph types available using R visit <http://r-graph-gallery.com/>.

Installation details for Linux, MacOS X and Windows are available on a number of CRAN Mirrors, including <https://cran.ma.imperial.ac.uk/>

These examples were run using R-4.0.3.

With R installed and running a number of packages need to be installed to provide the functions used in Andi's examples, and so at R's > prompt:

```
install.packages ('DBI')
install.packages ("RPostgres")
install.packages ("tidyverse")
install.packages ("DT")
install.packages ("fuzzyjoin")
install.packages ("tmap")
install.packages ("sf")
install.packages ("ggeffects")
install.packages ("effects")
```

Once installed these need to be loaded for the session,

```
library(RPostgres)
library(DBI)
library(tidyverse)
library(DT)
library(knitr)
library(fuzzyjoin)
library(tmap)
library(sf)
library(ggeffects)
library(effects)
```

more conveniently, these lines can be put in a text file and then read in from the File ... Source File pull down (on MacOS).

Following Andi's webpage, the following provide connection details and then connect to the WsprDaemon wsprnet database table spots:

```
db_name <- "wsprnet"
db_host <- "logs1.wsprdaemon.org"
db_user <- "wdread"
db_password <- "JTWSR2008"
db_con <- dbConnect(RPostgres::Postgres(),
  dbname = db_name,
  host = db_host,
  user = db_user,
  password = db_password)
```

¹⁵ See <https://www.r-project.org/>

¹⁶ See <http://r4stats.com/articles/why-r-is-hard-to-learn/> for example. There is a nice web tutorial at http://rtutorialseries.blogspot.com/2009/10/r-tutorial-series-introduction-to-r_11.html

alternatively, to connect to the WsprDaemon tutorial database for the wsprdaemon_spots_s and wsprdaemon_noise_s tables, simply alter the first line to:

```
db_name <- "tutorial"
```

To list the available tables, in this case where we've specified db_name as "tutorial":

```
dbListTables(db_con)
[1] "kp"                "wsprdaemon_noise_s" "wsprdaemon_spots_s"
```

The following example is a variant on a query in Andi's notes, where we've specified db_name as "wsprnet". It reads in one day of spots from G3ZIL on 20 m. Note that the double quotes needed for the "Reporter" field name with an upper case letter has been escaped as "\"Reporter\"":

```
one_day <- dbSendQuery(db_con,
"SELECT * FROM spots
WHERE \"Reporter\" = 'G3ZIL'
AND wd_band='20'
AND wd_time >= '2022-05-22T00:00:00Z'
AND wd_time < '2022-05-23T00:00:00Z';")
```

Note that dbSendQuery only works for SELECT queries. This only sends the query; it does not extract the records requested. Extracting records is done using dbFetch, followed by a dbClearResult to finish:

```
the_dat <- dbFetch(one_day)
dbClearResult(one_day)
```

The dbSendQuery documentation¹⁷ notes that, "*The query is submitted to the database server and the DBMS executes it, possibly generating vast amounts of data. Where these data live is driver-specific: some drivers may choose to leave the output on the server and transfer them piecemeal to R, others may transfer all the data to the client -- but not necessarily to the memory that R manages. See individual drivers' dbSendQuery documentation for details.*"

The data resulting from the dbFetch can be examined using:

```
the_dat %>%
datatable(options = dt_options)
```

The data appears as a web page in a browser. For the query above, which returned 5,702 rows, it appears that the data was transferred from the server to the local machine running R. However, if the same query was run without specifying a band, i.e. to select spots from all bands, resulting in a total of 16,171 rows in this case, the following message appeared, "*Warning message: In instance\$preRenderHook(instance) : It seems your data is too big for client-side DataTables. You may consider server-side processing: <https://rstudio.github.io/DT/server.html>*"

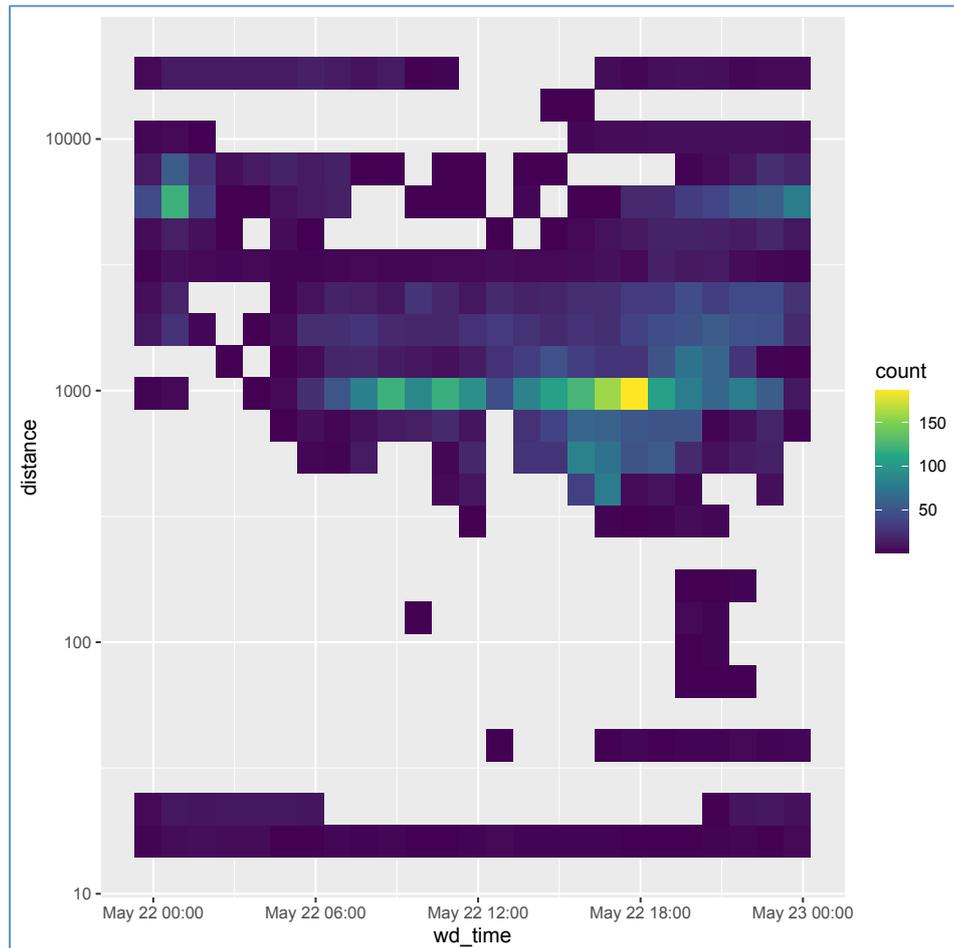
Andi's webpage has numerous examples of different kinds of data plots, here is a variant on those presented, using ggplot with geom_bin2D to give a heatmap (2D histogram) of the number of spots in time/distance bins [Note: If you get an error like the following it may be because you have a second monitor connected and it confuses R - the code below does not, of itself, have an error - *Error in UseMethod("depth") : no applicable method for 'depth' applied to an object of class "NULL"*]:

```
the_dat %>%
ggplot(aes(wd_time, distance)) +
geom_bin2d(bins = 24) +
```

¹⁷ See <https://www.rdocumentation.org/packages/DBI/versions/0.5-1/topics/dbSendQuery>

```
scale_fill_continuous(type = "viridis") +
scale_y_continuous(trans = "log10")
```

where we have specified a log10 y axis, which results in the plot below. Here we have spots via ground wave from local senders, a block at 3000 km and less (from Europe) that slopes from distant to near through daytime hours, a horizontal band representing North American senders from about 1400 to 0600 UTC, and Australia *except* from about 1100 to 1500 UTC:



The following is an example of a different type of plot, drawing on the WsprDaemon tutorial database table `wsprdaemon_spots_s` for columns `metric`¹⁸ and `SNR`. Here we extract two days of spots from 40 m from G3ZIL, having connected to database tutorial:

```
two_days <- dbSendQuery(db_con,
"SELECT * FROM wsprdaemon_spots_s
WHERE rx_id = 'G3ZIL'
AND band='40'
AND time >= '2022-05-22T00:00:00Z'
AND time < '2022-05-24T00:00:00Z';")
```

```
the_data <- dbFetch(two_days)
dbClearResult(two_days)
```

List the data to a web page to check:

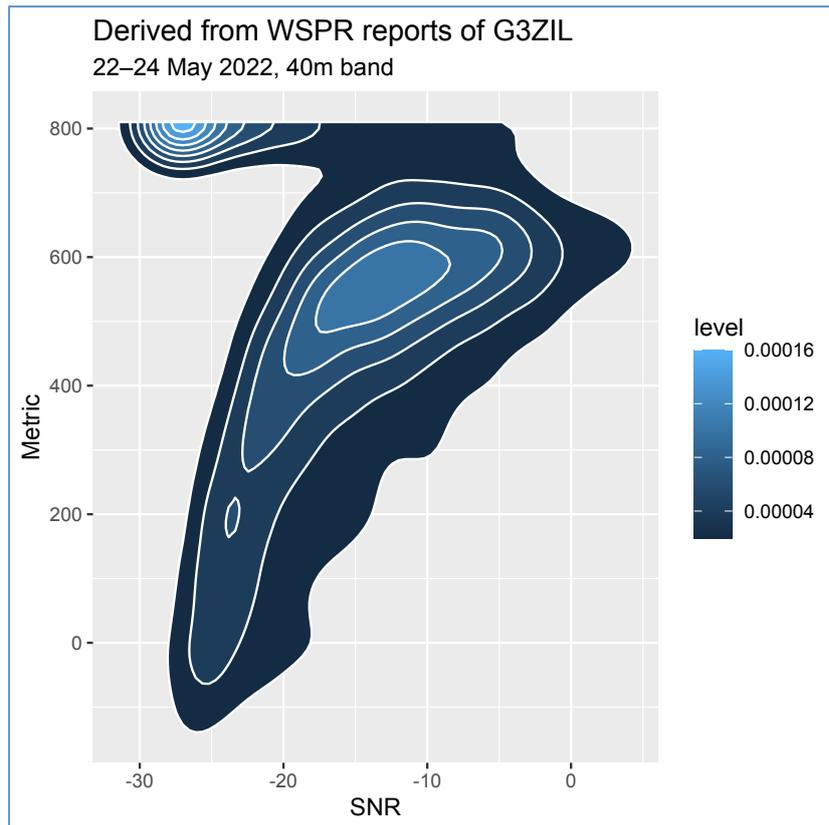
```
the_data %>%
datatable(options = dt_options)
```

¹⁸ See page 19 for a working definition of `metric`

Use ggplot with SNR on the x axis and metric on the y axis, with stat_density_2d being called to produce filled contours:

```
the_data %>%
  ggplot(aes(SNR, metric)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon", colour="white") +
  labs(x = "SNR", y = "Metric",
  title = "Derived from WSPR reports of G3ZIL",
  subtitle = "22-24 May 2022, 40m band")
```

The peak at a metric of 810 represents those spots decoded sent to the Ordered Statistics Decoder rather than the Fano decoder.



The following is a simple example comparing the c2_FFT and rms noise level estimates within database tutorial table wsprdaemon_noise_s. The data is from the Northern Utah SDR site KA7OEI-1 on 80 m between 1 and 7 January 2021.

```
five_days <- dbSendQuery(db_con,
  "SELECT * FROM wsprdaemon_noise_s
  WHERE site = 'KA7OEI-1'
  AND receiver='NUT_KIWI5'
  AND band='80'
  AND c2_level>-174
  AND time >= '2021-01-01T00:00:00Z'
  AND time < '2021-01-06T00:00:00Z';")
the_data <- dbFetch(five_days)
dbClearResult(five_days)
```

Calculate the summary statistics, note that the character between the two variables is a tilde ~ and not a minus sign:

```
summary(lm(the_data$c2_level ~ the_data$rms_level))
```

```

Call:
lm(formula = the_data$c2_level ~ the_data$rms_level)
Residuals:
    Min       1Q   Median       3Q      Max
-7.0665 -0.5004 -0.0531  0.4162  6.6198
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.440125   0.393603   3.659 0.000257 ***
the_data$rms_level 1.003274   0.003088 324.929 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.8682 on 3488 degrees of freedom
Multiple R-squared:  0.968,    Adjusted R-squared:  0.968
F-statistic: 1.056e+05 on 1 and 3488 DF,  p-value: < 2.2e-16

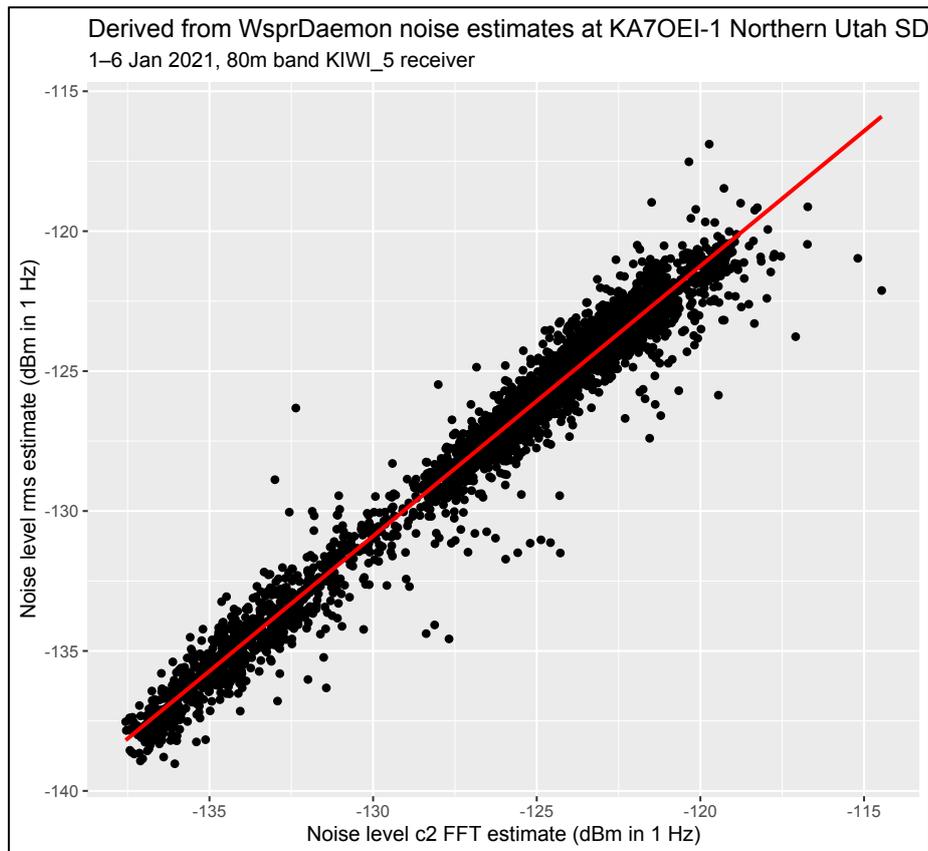
```

and plot the scatter plot with a linear fit:

```

the_data %>%
  ggplot(aes(c2_level,rms_level)) +
  geom_point() +
  geom_smooth(method=glm , color="red",fill="#69b3a2", se=TRUE,
  show.legend=TRUE) +
  labs(x = "Noise level c2 FFT estimate (dBm in 1 Hz)", y = "Noise level rms
  estimate (dBm in 1 Hz)",
  title = "Derived from WsprDaemon noise estimates at KA7OEI-1 Northern Utah
  SDR",
  subtitle = "1-6 Jan 2021, 80m band KIWI_5 receiver")

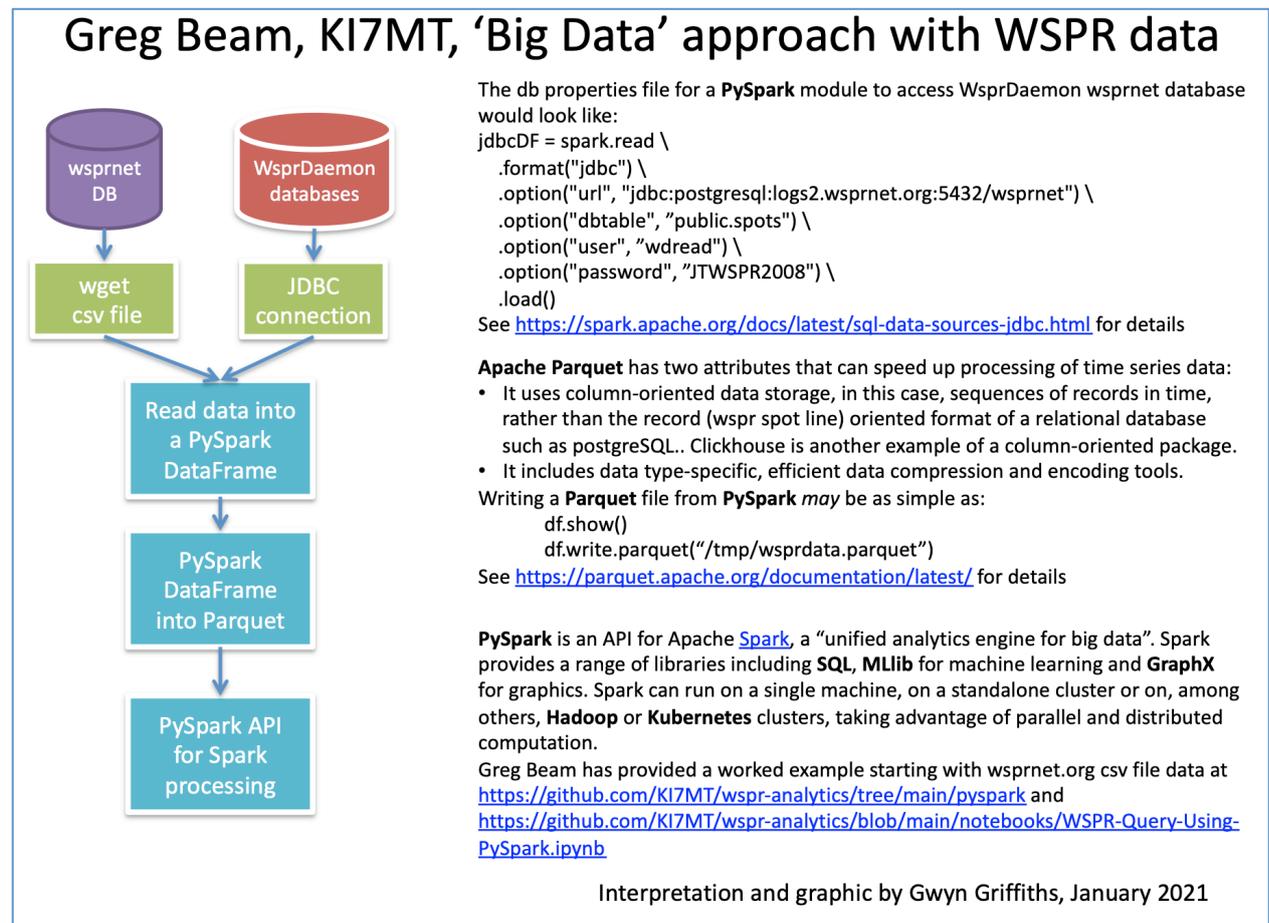
```



Annex H. A 'Big Data' approach to using data from the WsprDaemon databases

In a 5 January 2021 post (<https://groups.io/g/wsprdaemon/message/125>) to the WsprDaemon groups.io site Greg Beam, KI7MT, outlined a 'Big Data' approach he is pursuing to working with WSPR data. His starting point has been reading spot data from the csv format wsprnet.org archive files. Having discovered WsprDaemon he intends to make available tools to read spot data from the WsprDaemon postgresSQL databases.

More details on Greg's approach are in the exchanges following his initial post to groups.io/wsprdaemon, with more detail and resources on his github pages at <https://github.com/KI7MT/wspr-analytic>. Specific resources for working with WsprDaemon will be in the wsprdaemon directory. The annotated diagram below is my interpretation of the main steps of Greg's approach. s [Note: There have been no updates to his site since March 2021, interested readers are advised to contact him directly.]



On the following page we reproduce Greg's two examples of query times using the path outlined above for February 2020 spots data initially read from the wsprnet.org csv archive file.

In summary, on an undeclared machine/cluster:

The record count time (47,310,649 records) was 0.90 seconds.

A group by reporter count, ordered by record count for the top 10 took 4.94 seconds.

The next page also includes the code and times taken for the equivalent postgresSQL queries on the logs1.wsprdaemon.org server for data from September 2020 (we do not have February 2020).

In summary, on logs1.wsprdaemon.org using postgresSQL directly on the database:

The record count time (50,262,680 records) was 2.20 seconds.

A group by reporter count, ordered by record count for the top 10 took 3.32 seconds.

Greg Beam's metrics using the 'Big Data' approach:

```

* Reading file ...: wsprspots-2020-02.parquet
* File Size .....: 490,259,730 bytes compressed
* Read Time .....: 1.51834 sec

* Counting Records
* Record Count ...: 47,310,649
  • Count Time .....: 0.90268 sec

From https://github.com/KI7MT/
wspr-analytics/tree/main/pyspark

For February 2020 data from csv file from
wsprnet.org

* Running Group by Count Query and return the dataframe
+-----+-----+
|Reporter| count|
+-----+-----+
| DK6UG |838081|
| OE9GHV|690104|
| EA8BFK|648670|
| KD2OM |589003|
|KA7OEI-1|576788|
| K4RCG |571445|
| KPH  |551690|
| K9AN |480759|
| DF5FH|480352|
| DJ9PC|474211|
+-----+-----+
only showing top 10 rows

* Query Time .....: 4.94034 sec

```

Using postgresQL directly on September 2020 data on the then logs1.wsprdaemon.org server:

```

wsprnet=# SELECT count(*) FROM spots
WHERE wd_time >= '2020-09-01T00:00:00Z'
AND wd_time < '2020-10-01T00:00:00Z';

count
-----
50262680

From Explain Analyze:
Timing: Generation 8.959 ms, Inlining 295.593 ms,
Optimization 182.575 ms, Emission 131.328 ms,
Total 618.455 ms
Execution Time: 2206.806 ms

wsprnet=# SELECT "Reporter", count(*) FROM spots
WHERE wd_time >= '2020-09-01T00:00:00Z'
AND wd_time < '2020-10-01T00:00:00Z'
GROUP BY "Reporter"
ORDER BY count(*) desc limit 10;

Reporter | count
-----+-----
EA8BFK   | 900024
OE9GHV   | 810770
IW2NKE   | 617091
DK6UG    | 597008
KD2OM    | 585943
DF4UE/P  | 575539
KA7OEI-1 | 573672
LX1DQ    | 538225
WA2TP    | 529580
ON5KQ    | 509224
(10 rows)

From Explain Analyze:
Timing: Generation 16.510 ms, Inlining 342.045 ms,
Optimization 448.621 ms, Emission 261.886 ms, Total
1069.063 ms
Execution Time: 3322.240 ms

```

Annex I. A column-oriented database approach - Clickhouse

Over the last twenty years column-oriented databases have become popular¹⁹ for data-intensive applications. Values from individual columns are stored contiguously in memory, with compression best suited for the column data type (e.g. numeric or character). As a result, column-oriented queries, especially aggregates such as count and average, are very fast. This approach is well suited to time series data where inserts are overwhelmingly added to the end of the existing data.

Of the many column-oriented databases available²⁰, Clickhouse²¹ has grown an impressive list of adopters²², including Alibaba Cloud, Bloomberg, CERN and Cisco. It was developed for Yandex.Metrica ("*the second largest web analytics platform in the world*") by Yandex, Russia's largest technology company. From 2008–2015 a subsidiary, Yandex Labs, operated from Palo Alto, USA.

Arne, who has created the <https://wspr.live/> website uses Clickhouse as the database, and, in January 2021, switched to obtaining data from logs1.wsprdaemon.org (initially only those columns available on wsprnet.org, not the WsprDaemon-added columns but the current version can access the extended WsprDaemon spots and noise tables).

However, the wspr.live Clickhouse database is also populated with all of the spots ever reported to wsprnet.org. Arne has a simple data exporter for user specified times and calls at https://wspr.live/wspr_downloader.php, and numerous Grafana visualisation dashboards at <https://wspr.live/gui/>. The database itself is at db1.wspr.live and simple queries can be entered via Phil Barnard's page at: <http://wspr.rocks/livequeries/>

Comparative query times TimescaleDB and Clickhouse

a. Simple count of records for the month of September 2020

In Clickhouse syntax:

```
SELECT count(*) FROM rx
WHERE time >= toDateTime('2020-09-01 00:00:00')
AND time < toDateTime('2020-10-01 00:00:00');
count()
52204551
Round trip time 0.3 secs
```

b. Count spots from the top 10 reporters during September 2020

In Clickhouse syntax:

```
SELECT "rx_sign", count(*) FROM rx
WHERE time >= toDateTime('2020-09-01 00:00:00')
AND time < toDateTime('2020-10-01 00:00:00')
GROUP BY "rx_sign" ORDER BY count(*) DESC LIMIT 10;
```

¹⁹ See Abadi, D.J., Boncz, P.A. and Harizopoulos, S., 2009. Column-oriented database systems.

Proceedings of the VLDB Endowment, 2(2), pp.1664-1665. Available at ir.cwi.nl/pub/14834/14834A.pdf

²⁰ See https://en.wikipedia.org/wiki/List_of_column-oriented_DBMSes

²¹ See <https://clickhouse.tech/docs/en/>

²² See <https://clickhouse.tech/docs/en/introduction/adopters/>

```

row rx_sign  count()
1  EA8BFK  924654
2  OE9GHV  840895
3  IW2NKE  643998
4  DK6UG   621769
5  KD2OM   604761
6  DF4UE/P 598421
7  KA7OEI-1 591893
8  LX1DQ   564170
9  WA2TP   553122
10 ON5KQ   531990

```

Round trip time 0.3 secs

c. Inner Join on same table to compute average SNR difference between two reporters

There are reports²³ that the speed-up enabled by Clickhouse is not as great where JOINS are needed. The following TimescaleDB query syntax as used in several current Grafana dashboards was sent to Arne:

```

SELECT
  avg(s1."dB"-s2."dB") as ""dB difference""
FROM spots s1
JOIN spots s2 on s1.wd_time = s2.wd_time
AND s1."CallSign" = s2."CallSign"
AND s1."Reporter" = 'G3ZIL' and s2."Reporter" = 'G4HZX'
AND s1.wd_band='40' and s2.wd_band='40'
WHERE s1.wd_time >= '2022-05-01T00:00:00Z' AND s1.wd_time < '2022-05-14T00:00:00Z';

```

As written, executing on logs1.wsprdaemon.org, the execution time was 84.38 seconds. (The number of coincident spots was 69,356)

Arne reported that, "*Joining into the whole table takes forever so I tried to rewrite the query*", here, the JOIN is done on data extracted by two subqueries. In Clickhouse syntax and using Arne's column names:

```

SELECT avg(snr1 - snr2) AS dbdiff
FROM
  (SELECT time, tx_sign, snr AS snr2
   FROM rx
   WHERE (time >= toDateTime('2022-05-01 00:00:00'))
   AND (time < toDateTime('2022-05-14 00:00:00'))
   AND (band = 7)
   AND (rx_sign = 'G4HZX'))
) AS a
INNER JOIN
  (SELECT time, tx_sign, snr AS snr1
   FROM rx
   WHERE (time >= toDateTime('2022-05-01 00:00:00'))
   AND (time < toDateTime('2022-05-14 00:00:00'))
   AND (band = 7)
   AND (rx_sign = 'G3ZIL'))
) AS b USING (time, tx_sign)

```

²³ e.g. see <https://www.percona.com/blog/2017/06/22/clickhouse-general-analytical-workload-based-star-schema-benchmark/>

The elapsed time varied between 0.3 and 2.0 seconds:

Using this approach, but with TimescaleDB syntax:

```
SELECT avg(b."dB" - a."dB") AS dbdiff
FROM
  (SELECT wd_time, "CallSign", "dB" FROM spots
   WHERE wd_time >= '2022-05-01T00:00:00Z'
   AND wd_time < '2022-05-14T00:00:00Z'
   AND wd_band = '40'
   AND "Reporter" = 'G4HZX')
AS a
INNER JOIN
  (SELECT wd_time, "CallSign", "dB" FROM spots
   WHERE wd_time >= '2022-05-01T00:00:00Z'
   AND wd_time < '2022-05-14T00:00:00Z'
   AND wd_band = '40'
   AND "Reporter" = 'G3ZIL')
AS b
ON (a.wd_time = b.wd_time AND a."CallSign" = b."CallSign");
```

This resulted in a faster execution time of 4.97 seconds

Adding GROUP BY to this approach:

```
SELECT avg(b."dB" - a."dB") AS dbdiff
FROM
  (SELECT wd_time, "CallSign", "dB" FROM spots
   WHERE wd_time >= '2022-05-01T00:00:00Z'
   AND wd_time < '2022-05-14T00:00:00Z'
   AND wd_band = '40'
   AND "Reporter" = 'G4HZX' group by wd_time, "CallSign", "dB")
AS a
INNER JOIN
  (SELECT wd_time, "CallSign", "dB" FROM spots
   WHERE wd_time >= '2022-05-01T00:00:00Z'
   AND wd_time < '2022-05-14T00:00:00Z'
   AND wd_band = '40'
   AND "Reporter" = 'G3ZIL' group by wd_time, "CallSign", "dB")
AS b
ON (a.wd_time = b.wd_time AND a."CallSign" = b."CallSign");
```

extended the execution time to 5.36 seconds. The lesson here is that manual optimisation of the code can result in very useful performance improvements over and above the optimisation that postgresql does itself.

Annex J. Links to PostgreSQL APIs or notes for other languages/systems

C#

<http://zetcode.com/csharp/postgresql/>

Haskell

<https://hackage.haskell.org/package/postgrest>

Kotlin

<https://github.com/JetBrains/Exposed>

Lua

<https://keplerproject.github.io/luasql/manual.html>

Spark

<https://www.cdata.com/kb/tech/postgresql-jdbc-apache-spark.rst>

Swift 5 or later

<https://github.com/codewinsdotcom/PostgresClientKit>