Add Missing Procedures to an Upgraded PostgreSQL Instance

PostgreSQL Partition Manager

pg_partman is an extension to create and manage both time-based and number-based table partition sets. As of version 5.0.1, only built-in, declarative partitioning is supported and the older trigger-based methods have been deprecated.

The declarative partitioning built into PostgreSQL provides the commands to create a partitioned table and its children. pg_partman uses the built-in declarative features that PostgreSQL provides and builds upon those with additional features and enhancements to make managing partitions easier. One key way that pg_partman extends partitioning in Postgres is by providing a means to automate the child table maintenance over time (Ex. adding new children, dropping old ones based on a retention policy). pg_partman also has features to turn an existing table into a partitioned table or vice versa.

A background worker (BGW) process is included to automatically run partition maintenance without the need of an external scheduler (cron, etc) in most cases.

Bug reports & feature requests can be directed to the Issues section on Github - https://github.com/pgpartman/pg_partman/issues

For questions, comments, or if you're just not sure where to post, please use the Discussions section on Github. Feel free to post here no matter how minor you may feel your issue or question may be - https://github.com/pgpartman/pg_partman/discussions

DOCUMENTATION

The following list of files is found in the doc folder of the pg_partman github repository. For installation instructions, please see the next section of this README.

File	Description
pg_partman	Main reference documentation for pg_partman.

File	Description
pg_partman_howto	A How-To guide for general usage of pg_partman. Provides examples for setting up new partition sets and migrating existing tables to partitioned tables.
migrate_to_partman	How to migrate existing partition sets to being managed by pg_partman.
migrate_to_declarative	How to migrate from trigger-based partitioning to declarative partitioning.
pg_partman_5.0.1_upgrade	If pg_partman is being upgraded to version 5.x from any prior version, special considerations may need to be made. Please carefully review this document before performing any upgrades to 5.x or higher.
fix_missing_procedures	If pg_partman had been installed prior to PostgreSQL 11 and upgraded since then, it may be missing procedures. This document outlines how to restore those procedures and preserve the current configuration.

INSTALLATION

Requirement:

- PostgreSQL >= 14

Recommended:

- pg_jobmon (>=v1.4.0). PG Job Monitor will automatically be used if it is installed and setup properly.

From Source

In the directory where you downloaded pg_partman, run

```
make install
```

If you do not want the background worker compiled and just want the plain PL/PGSQL functions, you can run this instead:

```
make NO_BGW=1 install
```

Package

I do not personally maintain any OS packages for pg_partman, but several repository maintainers from the PostgreSQL Development Group (PGDG) have kindly been maintaining packages for the community. Please check the PostgreSQL Downloads page to see if your OS has a package available

Setup

The background worker must be loaded on database start by adding the library to shared_preload_libraries in postgresql.conf

```
shared_preload_libraries = 'pg_partman_bgw'     # (change requires restart)
```

You can also set other control variables for the BGW in postgresql.conf. "dbname" is required at a minimum for maintenance to run on the given database(s). These can be added/changed at anytime with a simple reload. See the documentation for more details. An example with some of them:

```
pg_partman_bgw.interval = 3600
pg_partman_bgw.role = 'keith'
pg_partman_bgw.dbname = 'keith'
```

Log into PostgreSQL and run the following commands. Schema is optional (but recommended) and can be whatever you wish, but it cannot be changed after installation. If you're using the BGW, the database cluster can be safely started without having the extension first created in the configured database(s). You can create the extension at any time and the BGW will automatically pick up that it exists without restarting the cluster (as long as shared_preload_libraries was set) and begin running maintenance as configured.

```
CREATE SCHEMA partman;
CREATE EXTENSION pg_partman SCHEMA partman;
```

pg_partman does not require a superuser to run, but currently still requires it to be installed. If not using a superuser, it is recommended that a dedicated role is created for running pg_partman functions and to be the owner of all partition sets that pg_partman maintains. At a minimum this role will need the following privileges (assuming pg_partman is installed to the `partman` schema and that dedicated role is called `partman_user`):

```
CREATE ROLE partman_user WITH LOGIN;
GRANT ALL ON SCHEMA partman TO partman_user;
GRANT ALL ON ALL TABLES IN SCHEMA partman TO partman_user;
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA partman TO partman_user;
GRANT EXECUTE ON ALL PROCEDURES IN SCHEMA partman TO partman_user;
GRANT ALL ON SCHEMA my_partition_schema TO partman_user;
GRANT TEMPORARY ON DATABASE mydb to partman_user; -- allow creation of temp tables to move d
```

If you need the role to also be able to create schemas, you will need to grant

create on the database as well. In general this shouldn't be required as long as you give the above role CREATE privileges on any pre-existing schemas that will contain partition sets.

```
GRANT CREATE ON DATABASE mydb TO partman_user;
```

I've received many requests for being able to install this extension on Amazon RDS. As of PostgreSQL 12.5, RDS has made the pg_partman extension available. Many thanks to the RDS team for including this extension in their environment!

https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/PostgreSQL_Partitions.html

UPGRADE

Run "make install" same as above or update your respective packages to put the new script files and libraries in place. Then run the following in PostgreSQL itself:

```
ALTER EXTENSION pg_partman UPDATE TO '<latest version>';
```

If you are doing a `pg_dump`/`pg_restore` and you've upgraded pg_partman in place from previous versions, it is recommended you use the `--column-inserts` option when dumping and/or restoring pg_partman's configuration tables. This is due to ordering of the configuration columns possibly being different (upgrades just add the columns onto the end, whereas the default of a new install may be different).

If upgrading between any major versions of pg_partman (4.x -> 5.x, etc), please carefully read all intervening version notes in the CHANGELOG, especially those notes for the major version. There are often additional instructions and other important considerations for the updates. Extra special considerations are needed if you are upgrading to 5+ from any version less than 5.0.0. Please see pg_partman_5.0.1_upgrade.

IMPORTANT NOTE: Some updates to pg_partman must drop and recreate its own database objects. If you are revoking PUBLIC privileges from functions/procedures, that can be added back to objects that are recreated as part of an update. If restrictions from PUBLIC use are desired for pg_partman, it is recommended to install it into its own schema as shown above and the revoke undesired access to that schema. Otherwise you may have to add an additional step to your extension upgrade procedures to revoke PUBLIC access again.

EXAMPLES

For setting up partitioning with pg_partman on a brand new table, or to migrate an existing normal table to partitioning, see pg_partman_howto.

For migrating a trigger-based partitioned table to declarative partitioning using pg_partman, see migrate_to_declarative. Note that if you plan to migrate to

pg_partman, you will first have to migrate to a declarative partitioned table before it can be managed by pg_partman.

Other documents are also available in the doc folder.

See the pg_partman reference file in the doc folder for full details on all commands and options for pg_partman.

TESTING

This extension can use the pgTAP unit testing suite to evaluate if it is working properly - http://www.pgtap.org.

WARNING: You MUST increase max_locks_per_transaction above the default value of 64. A value of 128 has worked well so far with existing tests. This is due to the subpartitioning tests that create/destroy several hundred tables in a single transaction. If you don't do this, you risk a cluster crash when running subpartitioning tests.

See the README file contained in the test folder for more information on testing.

About

PostgreSQL Partition Manager is an extension to help make managing time or number/id based table partitioning easier. It has many options, but usually only a few are needed, so it's much easier to use than it may first appear (and definitely easier than implementing it yourself).

As of version 5.0.1, the minimum version of PostgreSQL required is 14 and trigger-based partitioning is no longer supported. All partitioning is done using built-in declarative partitioning. Currently only ranged partitioning is supported for time- and number-based intervals. Version 4.x of pg_partman, which still has trigger-based support, is no longer in active development and will only be receiving critical bug fixes for a limited time. If partitioning is a critical part of your infrastructure, please make plans to upgrade in the near future.

A default partition to catch data outside the existing child boundaries is automatically created for all partition sets. The `check_default()` function provides monitoring for any data getting inserted into the default table and the `partition_data_*` set of functions can easily partition that data for you if it is valid data. That is much easier than automatically creating new child tables on demand and having to clean up potentially hundreds or thousands of unwanted partitions. And also better than throwing an error and losing the data!

Note that future child table creation is based on the data currently in the partition set and, by default, ignores data in the default. It is recommended that

you set the `premake` value high enough to encompass your expected data range being inserted. See below for further explanations on these configuration values.

If you have an existing partition set and you'd like to migrate it to pg_partman, please see the migrate_to_partman.md file in the doc folder.

Child Table Property Inheritance

For this extension, most of the attributes of the child partitions are all obtained from the parent table. With declarative partitioning, certain features are not able to be inherited from the parent depending on the version of PostgreSQL. So pg_partman uses a template table instead. The following table matrix shows how certain property inheritances are managed with pg_partman. The number given is the version of PostgreSQL. If a property is not listed here, then assume it is managed via the parent.

Feature	Parent Inheritance	Template Inheritance
non-partition column primary key		14+
non-partition column unique index		14+
non-partition column unique index tablespace		14+
unlogged table state*		14+
non-unique indexes	14+	
privileges/ownership	14+	

If a property is managed via the template table, it likely will not be retroactively applied to all existing child tables if that property is changed. It will apply to any newly created children, but will have to be manually applied to any existing children.

Privileges & ownership are NOT inherited by default. If enabled by pg_partman, note that this inheritance is only at child table creation and isn't automatically retroactive when changed (see `reapply_privileges()`). Unless you need direct access to the child tables, this should not be needed. You can set the `inherit_privileges` option if this is needed (see config table information below).

If you are using the IDENTITY feature for sequences, the automatic generation of new sequence values using this feature is only supported when data is inserted through the parent table, not directly into the children.

IMPORTANT NOTES:

- The template table feature is only a temporary solution to help speed up declarative partitioning adoption. As things are handled better in core, the use of the template table will be phased out quickly from pg_partman. If a feature that was managed by the template is supported in core in the future, it will eventually be removed from template management in

pg_partman, so please plan ahead for that during major version upgrading if it applies to you.

- The UNLOGGED status is managed via pg_partman's template due to an inconsistency in the way the property is handled when either enabling or disabling UNLOGGED on the parent table of a partition set. That property does not actually change on the parent table when the ALTER command is written so new child tables will continue to use the property that existed before. So if you wanted to change a partition set from UNLOGGED to LOGGED for all future children, it does not work. With the property now being managed on the template table, changing it there will allow the change to propagate to newly created children. Pre-existing child tables will have to be changed manually, but that has always been the case. See reported bug at https://www.postgresql.org/message-id/flat/15954-b61523bed4b110c4%40postgresql.org

Time Zones

It is important to ensure that the time zones for all systems that will be running pg_partman maintenance operations are consistent, especially when running time-based partitioning. The calls to pg_partman functions will use the time zone that is set by the client at the time the functions are called. This is consistent with the way libpq clients work in general.

In general, it is highly recommended to always run your database system in UTC time. It makes handling any time-related issues tremendously easier and especially to overcome issues that are currently not possible to solve due to Daylight Saving Time (DST) changes. In addition to this, also ensure the client that will be creating partition sets and running the maintenance calls is also set to UTC. For example, trying to partition hourly will either break when the time changes or skip creating a child table.

Subpartitioning

Subpartitioning with multiple levels is supported, but it is of very limited use in PostgreSQL and provides next to NO PERFORMANCE BENEFIT outside of extremely large data in a single partition set (100s of terabytes, petabytes). If you're looking for performance benefits, adjust your partition interval before considering subpartitioning. It's main use is in data organization and retention management.

You can do time->time, id->id, time->id and id->time. There is no set limit on the level of subpartitioning you can do, but be sensible and keep in mind performance considerations on managing many tables in a single inheritance set. Also, if the number of tables in a single partition set gets very high, you may have to adjust the `max_locks_per_transaction` postgresql.conf setting above the default of 64. Otherwise you may run into shared memory issues or even crash the cluster.

If you have contention issues when `run_maintenance()` is called for general maintenance of all partition sets, you can set the **`automatic_maintenance`** column in the **`part_config`** table to false if you do not want that general call to manage your subpartition set. But you must then call `run_maintenance(parent_table)` directly, and often enough, to have to future partitions made. You can use the run_maintenance_proc() procedure instead of the base function to cause less contention issues since it automatically commits after each partition set's maintenance.

PUBLICATION/SUBSCRIPTION for logical replication is NOT supported with subpartitioning.

See the `create_sub_parent()` & `run_maintenance()` functions below for more information.

Retention

If you don't need to keep data in older partitions, a retention system is available to automatically drop unneeded child partitions. By default, they are only uninherited/detached not actually dropped, but that can be configured if desired. There is also a method available to dump the tables out if they don't need to be in the database anymore but still need to be kept. To set the retention policy, enter either an interval or integer value into the **retention** column of the **part_config** table. For time-based partitioning, the interval value will set that any partitions containing only data older than that will be dropped (including safely handling cases where the retention interval is not a multiple of the partition size). For id-based partitioning, the integer value will set that any partitions with an id value less than the current maximum id value minus the retention value will be dropped. For example, if the current max id is 100 and the retention value is 30, any partitions with id values less than 70 will be dropped. The current maximum id value at the time the drop function is run is always used. Keep in mind that for subpartition sets, when a parent table has a child dropped, if that child table is in turn partitioned, the drop is a CASCADE and ALL child tables down the entire inheritance tree will be dropped. Also note that a partition set managed by pg_partman must always have at least one child, so retention will never drop the last child table in a set.

Constraint Exclusion

One of the big advantages of partitioning is a feature called **constraint exclusion** (see docs for explanation of functionality and examples http://www.postgresql.org/docs/current/static/ddl-partitioning.html#DDL-PARTITIONING-CONSTRAINT-EXCLUSION). The problem with most partitioning setups however, is that this will only be used on the partitioning control column. If you use a WHERE condition on any other column in the partition set, a scan across all child tables will occur unless there are also constraints on those columns. And predicting what a column's values will

be to pre-create constraints can be very hard or impossible. `pg_partman` has a feature to apply constraints on older tables in a partition set that no longer have any edits done to them ("old" being defined as older than the `optimize_constraint` config value). It checks the current min/max values in the given columns and then applies a constraint to that child table. This can allow the constraint exclusion feature to potentially eliminate scanning older child tables when other columns are used in WHERE conditions. Be aware that this limits being able to edit those columns, but for the situations where it is applicable it can have a tremendous affect on query performance for very large partition sets. So if you are only inserting new data this can be very useful, but if data is regularly being inserted/updated throughout the entire partition set, this is of limited use. Functions for easily recreating constraints are also available if data does end up having to be edited in those older partitions. Note that constraints managed by PG Partman SHOULD NOT be renamed in order to allow the extension to manage them properly for you. For a better example of how this works, please see this blog post: http://www.keithf4.com/managing-constraint-exclusion-in-table-partitioning

Adding these constraints could potentially cause contention with the data contained in those tables and also make pg_partman maintenance take a long time to run. There is a "constraint_valid" column in the part_config(_sub) table to set whether these constraints should be set NOT VALID on creation. While this can make the creation of the constraint(s) nearly instantaneous, constraint exclusion cannot be used until it is validated. This is why constraints are added as valid by default.

NOTE: This may not work with subpartitioning. It will work on the first level of partitioning, but is not guaranteed to work properly on further subpartition sets depending on the interval combinations and the optimize_constraint value. Ex: Weekly -> Daily with a daily optimize_constraint of 7 won't work as expected. Weekly constraints will get created but daily subpartition ones likely will not.

Time Interval Considerations

The smallest time interval supported is 1 second and the upper limit is bounded by the minimum and maximum timestamp values that PostgreSQL supports (http://www.postgresql.org/docs/current/static/datatype-datetime.html).

When first running `create_parent()` to create a partition set, intervals less than a day round down when determining what the first partition to create will be. Intervals less than 24 hours but greater than 1 minute use the nearest hour rounded down. Intervals less than 1 minute use the nearest minute rounded down. However, enough partitions will be made to support up to what the real current time is. This means that when `create_parent()` is run, more previous partitions may be made than expected and all future partitions may not be made. The first run of `run_maintenance()` will fix the missing future partitions. This happens due to the nature of being able to support custom time intervals.

Any intervals greater than or equal to 24 hours should set things up as would be expected.

Keep in mind that for intervals equal to or greater than 100 years, the extension will use the real start of the century or millennium to determine the partition name & constraint rules. For example, the 21st century and 3rd millennium started January 1, 2001 (not 2000). This also means there is no year "0".

For weekly partitions, note that the default "start" of the week will be based on the day of the week that you run `create_parent()`. For example, if you ran it on a Tuesday or Friday, the time boundaries of the child tables would all start on those respective days vs the expected Monday or Sunday start of the week. The easiest way to handle this is to use the `date_trunc()` function to start the weeks on a Monday using the `p_start_partition` parameter to `create_parent()`. Starting on Sundays is likely possible as well, but trickier and outside the scope of the documentation at this time.

```
SELECT partman.create_parent('public.time_table', 'col3', '1 week', p_start_partition := to_
```

Naming Length Limits

PostgreSQL has an object naming length limit of 63 bytes (NOT characters). If you try and create an object with a longer name, it truncates off any characters at the end to fit that limit. This can cause obvious issues with partition names that rely on having a specifically named suffix. PG Partman automatically handles this for all child table names. It will truncate off the existing parent table name to fit the required suffix. Be aware that if you have tables with very long, similar names, you may run into naming conflicts if they are part of separate partition sets. With number based partitioning, be aware that over time the table name will be truncated more and more to fit a longer partition suffix. So while the extension will try and handle this edge case for you, it is recommended to keep table names that will be partitioned as short as possible.

Unique Constraints

Table inheritance in PostgreSQL does not allow a primary key or unique index/constraint on the parent to apply to all child tables. The constraint is applied to each individual table, but not on the entire partition set as a whole. For example, this means a careless application can cause a primary key value to be duplicated in a partition set. In the mean time, a python script is included with `pg_partman` that can provide monitoring to help ensure the lack of this feature doesn't cause long term harm. See **`check_unique_constraint.py`** in the **Scripts** section.

Logging/Monitoring

The PG Jobmon extension (https://github.com/omniti-labs/pg_jobmon) is optional and allows auditing and monitoring of partition maintenance. If jobmon

is installed and configured properly, it will automatically be used by partman with no additional setup needed. Jobmon can also be turned on or off individually for each partition set by using the `jobmon` column in the `part_config` table or with the option to `create_parent()` during initial setup. Note that if you try to partition `pg_jobmon`'s tables you **MUST** set the jobmon option in `create_parent()` to false, otherwise it will be put into a permanent lockwait since `pg_jobmon` will be trying to write to the table it's trying to partition. By default, any function that fails to run successfully 3 consecutive times will cause jobmon to raise an alert. This is why the default pre-make value is set to 4 so that an alert will be raised in time for intervention with no additional configuration of jobmon needed. You can of course configure jobmon to alert before (or later) than 3 failures if needed. If you're running partman in a production environment it is HIGHLY recommended to have jobmon installed and some sort of 3rd-party monitoring configured with it to alert when partitioning fails (Nagios, Circonus, etc).

Background Worker

`pg_partman`'s BGW is basically just a scheduler that runs the `run_maintenance()` function for you so that you don't have to use an external scheduler (cron, etc). Right now it doesn't do anything differently than calling `run_maintenance()` directly, but that may change in the future. See the README.md file for installation instructions. If you need to call `run_maintenance()` directly on any specific partition sets, you will still need to do so manually using an outside scheduler. This only maintains partition sets that have `automatic_maintenance` in `**part_config**` set to true. LOG messages are output to the normal PostgreSQL log file to indicate when the BGW runs. Additional logging messages are available if *log_min_messages* is set to "DEBUG1".

REMEMBER: You must have `pg_partman_bgw` in your `shared_preload_libraries` (requires a restart).

The following configuration options are available to add into postgresql.conf to control the BGW process:

- `pg_partman_bgw.dbname`
  - Required. The database(s) that `run_maintenance()` will run on. If more than one, use a comma separated list. If not set, BGW will do nothing.
- `pg_partman_bgw.interval`
  - Number of seconds between calls to `run_maintenance()`. Default is 3600 (1 hour).
  - See further documentation below on suggested values for this based on partition types & intervals used.
- `pg_partman_bgw.role`
  - The role that `run_maintenance()` will run as. Default is "postgres". Only a single role name is allowed.

- `pg_partman_bgw.analyze`
  - Same purpose as the p_analyze argument to `run_maintenance()`. See below for more detail. Set to 'on' for TRUE. Set to 'off' for FALSE (Default is 'off').
- `pg_partman_bgw.jobmon`
  - Same purpose as the p_jobmon argument to `run_maintenance()`. See below for more detail. Set to 'on' for TRUE. Set to 'off' for FALSE. Default is 'on'.

If for some reason the main background worker process crashes, it is set to try and restart every 10 minutes. Check the postgres logs for any issues if the background worker is not starting.

Extension Objects

Requiring a superuser to use pg_partman is completely optional. To run as a non-superuser, the role(s) that run pg_partman functions and maintenance must have ownership of all partition sets they manage and permissions to create objects in any schema that will contain partition sets that it manages. For ease of use and privilege management, it is recommended to create a role dedicated to partition management. Please see the main README.md file for role & privileges setup instructions.

As a note for people that were not aware, you can name arguments in function calls to make calling them easier and avoid confusion when there are many possible arguments. If a value has a default listed, it is not required to pass a value for that argument. As an example: `SELECT create_parent('schema.table', 'col1', '1 day', p_start_partition := '2023-03-20');`

Creation Functions

```
create_parent(
    p_parent_table text
    , p_control text
    , p_interval text
    , p_type text DEFAULT 'range'
    , p_epoch text DEFAULT 'none'
    , p_premake int DEFAULT 4
    , p_start_partition text DEFAULT NULL
    , p_default_table boolean DEFAULT true
    , p_automatic_maintenance text DEFAULT 'on'
    , p_constraint_cols text[] DEFAULT NULL
    , p_template_table text DEFAULT NULL
    , p_jobmon boolean DEFAULT true
    , p_date_trunc_interval text DEFAULT NULL
)
RETURNS boolean
```

- Main function to create a partition set with one parent table and inherited children. Parent table must already exist and be declared as partitioned before calling this function. All options passed to this function must match that definition. Please apply all defaults, indexes, constraints, privileges & ownership to parent table so they will propagate to children. See notes above about handling unique indexes and other table properties.
- An ACCESS EXCLUSIVE lock is taken on the parent table during the running of this function. No data is moved when running this function, so lock should be brief
- A default partition and template table are created by default unless otherwise configured
- `p_parent_table` - the existing parent table. MUST be schema qualified, even if in public schema
- `p_control` - the column that the partitioning will be based on. Must be a time or integer based column
- `p_interval` - the time or integer range interval for each partition. No matter the partitioning type, value must be given as text.
  - *<interval>* - Any valid value for the interval data type. Do not type cast the parameter value, just leave as text.
  - *<integer>* - For ID based partitions, the integer value range of the ID that should be set per partition. Enter this as an integer in text format ('100' not 100). Currently must be greater than or equal to 2.
- `p_type` - the type of partitioning to be done. Currently only **range** is supported.
- `p_epoch` - tells `pg_partman` that the control column is an integer type, but actually represents and epoch time value. Valid values for this option are: 'seconds', 'milliseconds', 'nanoseconds', and 'none'. The default is 'none'. All table names will be time-based. In addition to a normal index on the control column, be sure you create a functional, time-based index on the control column (to_timestamp(controlcolumn)) as well so this works efficiently.
- `p_premake` - is how many additional partitions to always stay ahead of the current partition. Default value is 4. This will keep at minimum 5 partitions made, including the current one. For example, if today was Sept 6th, and `premake` was set to 4 for a daily partition, then partitions would be made for the 6th as well as the 7th, 8th, 9th and 10th. Note some intervals may occasionally cause an extra partition to be premade or one to be missed due to leap years, differing month lengths, etc. This usually won't hurt anything and should self-correct (see **About** section concerning timezones and non-UTC). If partitioning ever falls behind the `premake` value, normal running of `run_maintenance()` and data insertion should automatically catch things up.
- `p_start_partition` - allows the first partition of a set to be specified instead of it being automatically determined. Must be a valid timestamp (for time-based) or positive integer (for id-based) value. Be aware, though, the actual parameter data type is text. For time-based partitioning, all par-

titions starting with the given timestamp up to CURRENT_TIMESTAMP (plus `premake`) will be created. For id-based partitioning, only the partition starting at the given value (plus `premake`) will be made. Note that for subpartitioning, this only applies during initial setup and not during ongoing maintenance.

- `p_default_table` - boolean flag to determine whether a default table is created. Defaults to true.
- `p_automatic_maintenance` - parameter to set whether maintenance is managed automatically when `run_maintenance()` is called without a table parameter or by the background worker process. Current valid values are "on" and "off". Default is "on". When set to off, `run_maintenance()` can still be called on an individual partition set by passing it as a parameter to the function. See **run_maintenance** in Maintenance Functions section below for more info.
- `p_constraint_cols` - an optional array parameter to set the columns that will have additional constraints set. See the **About** section above for more information on how this works and the **apply_constraints()** function for how this is used.
- `p_template_table` - If you do not pass a value here, a template table will automatically be made for you in same schema that pg_partman was installed to. If you pre-create a template table and pass its name here, then the initial child tables will obtain these properties discussed in the **About** section immediately.
- `p_jobmon` - allow `pg_partman` to use the `pg_jobmon` extension to monitor that partitioning is working correctly. Defaults to TRUE.
- `p_date_trunc_interval` - By default, pg_partman's time-based partitioning will truncate the child table starting values to line up at the beginning of typical boundaries (midnight for daily, day 1 for monthly, Jan 1 for yearly, etc). If a custom time interval that does not fall on those boundaries is desired, this option may be required to ensure the child table has the expected boundaries (especially if you also set p_start_partition). The valid values allowed for this parameter are the interval values accepted by the built-in date_trunc() function (day, week, month, etc). For example, if you set a 9-week interval, by default pg_partman would truncate the tables by month (since the interval is greater than one month but less than 1 year) and unexpectedly start on the first of the month in some cases. Set this value to week, so that the child table start values are properly truncated on a weekly basis to line up with the 9-week interval. If you are using a custom time interval, please experiment with this option to get the expected set of child tables you desire or use a more typical partitioning interval to simplify partition management.

```
create_sub_parent(
    p_top_parent text
    , p_declarative_check text DEFAULT NULL
    , p_control text
```

```
    , p_interval text
    , p_type text DEFAULT 'range'
    , p_epoch text DEFAULT 'none'
    , p_premake int DEFAULT 4
    , p_start_partition text DEFAULT NULL
    , p_default_table boolean DEFAULT true
    , p_constraint_cols text[] DEFAULT NULL
    , p_jobmon boolean DEFAULT true
    , p_date_trunc_interval text DEFAULT NULL
)
RETURNS boolean
```

- Create a subpartition set of an already existing partitioned set. See important notes about Subpartitioning in **About** section.
- `p_top_parent` - This parameter is the parent table of an already existing partition set. It tells `pg_partman` to turn all child tables of the given partition set into their own parent tables of their own partition sets using the rest of the parameters for this function.
- `p_declarative_check` - Turning an existing partition set into a subpartitioned set is a **destructive** process. A table must be declared partitioned at creation time and cannot be altered later. Therefore existing child tables must be dropped and recreated as partitioned parent tables. This flag is here to help ensure this function is not run without prior consent that all data in the partition set will be destroyed as part of the creation process. It must be set to "yes" to proceed with subpartitioning.
- All other parameters to this function have the same exact purpose as those of `create_parent()`, but instead are used to tell `pg_partman` how each child table shall itself be partitioned.
- For example if you have an existing partition set done by year and you then want to partition each of the year partitions by day, you would use this function.
- It is advised that you keep table names short for subpartition sets if you plan on relying on the table names for organization. The suffix added on to the end of a table name is always guaranteed to be there for whatever partition type is active for that set. Longer table names may cause the original parent table names to be truncated and possibly cut off the top level partitioning suffix. This cannot be controlled and ensures the lowest level partitioning suffix survives.
- Note that for the first level of subpartitions, the `p_parent_table` argument you originally gave to `create_parent()` would be the exact same value you give to `create_sub_parent()`. If you need further subpartitioning, you would then start giving `create_sub_parent()` a different value (the child tables of the top level partition set).
- The template table that is already set for the given p_top_parent will automatically be used.

`partition_data_time(`

```
        p_parent_table text
        , p_batch_count int DEFAULT 1
        , p_batch_interval interval DEFAULT NULL
        , p_lock_wait numeric DEFAULT 0
        , p_order text DEFAULT 'ASC'
        , p_analyze boolean DEFAULT true
        , p_source_table text DEFAULT NULL
        , p_ignored_columns text[] DEFAULT NULL
)
RETURNS bigint
```

- This function is used to partition data that may have existed prior to setting up the parent table as a time-based partition set. It also fixes data that gets inserted into the default table.
- If the needed partition does not exist, it will automatically be created. If the needed partition already exists, the data will be moved there.
- If you are trying to partition a large amount of data automatically, it is recommended to use the `partition_data_proc` procedure to commit data in smaller batches. This will greatly reduce issues caused by long running transactions and data contention.
- For subpartitioned sets, you must start partitioning data at the highest level and work your way down each level. This means you must first run this function before running create_sub_parent() to create the additional partitioning levels. Then continue running this function again on each new sub-parent once they're created. See the pg_partman_howto.md document for a full example. IMPORTANT NOTE: Be VERY cautious with subpartition sets and using this function since subpartitioning can be a destructive operation. See create_sub_parent().
- `p_parent_table` - the existing parent table. MUST be schema qualified, even if in public schema.
- `p_batch_count` - optional argument, how many times to run the `batch_interval` in a single call of this function. Default value is 1. Currently sets how many child tables will be processed in a single run, but when p_batch_interval is working again will refer explicitly to how many batches to run.
- `p_batch_interval` - optional argument, sets the interval of data to be moved in each batch. Defaults to the configured partition interval if not given or if you give an interval larger than the partition interval. IMPORTANT NOTE: This cannot be set smaller than the partition interval if moving data out of the default table. Work is being done to allow this, but with some limitations. If you are moving data from a source table that is not the partition set's default table, you can set this interval smaller than the partitioning interval to help avoid moving large amounts of data in long running transactions.
- `p_lock_wait` - optional argument, sets how long in seconds to wait for a row to be unlocked before timing out. Default is to wait forever.

- `p_order` - optional argument, by default data is migrated out of the default in ascending order (ASC). Allows you to change to descending order (DESC).
- `p_analyze` - optional argument, by default whenever a new child table is created, an analyze is run on the parent table of the partition set to ensure constraint exclusion works. This analyze can be skipped by setting this to false and help increase the speed of moving large amounts of data. If this is set to false, it is highly recommended that a manual analyze of the partition set be done upon completion to ensure statistics are updated properly.
- `p_source_table` - This option can be used when you need to move data into a partitioned table. Pass a schema qualified tablename to this parameter and any data in that table will be MOVED to the partition set designated by p_parent_table, creating any child tables as needed.
- `p_ignored_columns` - This option allows for filtering out specific columns when moving data from the default/source to the target child table(s). This is generally only required when using columns with a GENERATED ALWAYS value since directly inserting a value would fail when moving the data. Value is a text array of column names.
- Returns the number of rows that were moved from the parent table to partitions. Returns zero when source table is empty and partitioning is complete.

```
partition_data_id(p_parent_table text
    , p_batch_count int DEFAULT 1
    , p_batch_interval bigint DEFAULT NULL
    , p_lock_wait numeric DEFAULT 0
    , p_order text DEFAULT 'ASC'
    , p_analyze boolean DEFAULT true
    , p_source_table text DEFAULT NULL
    , p_ignored_columns text[] DEFAULT NULL
)
RETURNS bigint
```

- This function is used to partition data that may have existed prior to setting up the parent table as a number-based partition set. It also fixes data that gets inserted into the default.
- If the needed partition does not exist, it will automatically be created. If the needed partition already exists, the data will be moved there.
- If you are trying to partition a large amount of data automatically, it is recommended to use the `partition_data_proc` procedure to commit data in smaller batches. This will greatly reduce issues caused by long running transactions and data contention.
- For subpartitioned sets, you must start partitioning data at the highest level and work your way down each level. This means you must first run this function before running create_sub_parent() to create the additional partitioning levels. Then continue running this function again on each

new sub-parent once they're created. See the pg_partman_howto.md document for a full example. IMPORTANT NOTE: Be VERY cautious with subpartition sets and using this function since subpartitioning can be a destructive operation. See create_sub_parent().

- `p_parent_table` - the existing parent table. MUST be schema qualified, even if in public schema.
- `p_batch_count` - optional argument, how many times to run the `batch_interval` in a single call of this function. Default value is 1. This sets how many child tables will be processed in a single run.
- `p_batch_interval` - optional argument, sets the interval of data to be moved in each batch. Defaults to the configured partition interval if not given or if you give an interval larger than the partition interval. IMPORTANT NOTE: This cannot be set smaller than the partition interval if moving data out of the default table. Work is being done to allow this, but with some limitations. If you are moving data from a source table that is not the partition set's default table, you can set this interval smaller than the partitioning interval to help avoid moving large amounts of data in long running transactions.
- `p_lock_wait` - optional argument, sets how long in seconds to wait for a row to be unlocked before timing out. Default is to wait forever.
- `p_order` - optional argument, by default data is migrated out of the parent in ascending order (ASC). Allows you to change to descending order (DESC).
- `p_analyze` - optional argument, by default whenever a new child table is created, an analyze is run on the parent table of the partition set to ensure constraint exclusion works. This analyze can be skipped by setting this to false and help increase the speed of moving large amounts of data. If this is set to false, it is highly recommended that a manual analyze of the partition set be done upon completion to ensure statistics are updated properly.
- `p_source_table` - This option can be used when you need to move data into a partitioned table. Pass a schema qualified tablename to this parameter and any data in that table will be MOVED to the partition set designated by p_parent_table, creating any child tables as needed.
- `p_ignored_columns` - This option allows for filtering out specific columns when moving data from the default/source to the target child table(s). This is generally only required when using columns with a GENERATED ALWAYS value since directly inserting a value would fail when moving the data. Value is a text array of column names.
- Returns the number of rows that were moved from the parent table to partitions. Returns zero when source table is empty and partitioning is complete.

```
partition_data_proc (
    p_parent_table text
    , p_loop_count int DEFAULT NULL
```

```
    , p_interval text DEFAULT NULL
    , p_lock_wait int DEFAULT 0
    , p_lock_wait_tries int DEFAULT 10
    , p_wait int DEFAULT 1, p_order text DEFAULT 'ASC'
    , p_order text DEFAULT 'ASC',
    , p_source_table text DEFAULT NULL
    , p_ignored_columns text[] DEFAULT NULL
    , p_quiet boolean DEFAULT false
)
```

- A procedure that can partition data in distinct commit batches to avoid long running transactions and data contention issues.
- Calls either partition_data_time() or partition_data_id() in a loop depending on partitioning type.
- `p_parent_table` - Parent table of an already created partition set.
- `p_loop_count` - How many times to loop through the value given for p_interval. If p_interval not set, will use default partition interval and make at most this many partition(s). Procedure commits at the end of each loop (NOT passed as p_batch_count to partitioning function). If not set, all data in the parent/source table will be partitioned in a single run of the procedure.
- `p_interval` - Parameter that is passed on to the partitioning function as p_batch_interval argument. See underlying functions for further explanation.
- `p_lock_wait` - Parameter that is passed directly through to the underlying partition_data_*() function. Number of seconds to wait on rows that may be locked by another transaction. Default is to wait forever (0).
- `p_lock_wait_tries` - Parameter to set how many times the procedure will attempt waiting the amount of time set for p_lock_wait. Default is 10 tries.
- `p_wait` - Cause the procedure to pause for a given number of seconds between commits (batches) to reduce write load
- `p_order` - Same as the p_order option in the called partitioning function
- `p_source_table` - Same as the p_source_table option in the called partitioning function
- `p_ignored_columns` - This option allows for filtering out specific columns when moving data from the default/parent to the proper child table(s). This is generally only required when using columns with a GENERATED ALWAYS value since directly inserting a value would fail when moving the data. Value is a text array of column names.
- `p_quiet` - Procedures cannot return values, so by default it emits NOTICE's to show progress. Set this option to silence these notices.

```
create_partition_time(
    p_parent_table text
    , p_partition_times timestamptz[]
    , p_start_partition text DEFAULT NULL
```

```
)
RETURNS boolean
```

- This function is used to create child partitions for the given parent table.
- Normally this function is never called manually since partition creation is managed by run_maintenance(). But if you need to force the creation of specific child tables outside of normal maintenance, this function makes it easier.
- `p_parent_table` - parent table to create new child table(s) in.
- `p_partition_times` - An array of timestamptz values to create children for. If the child table does not exist, it will be created. If it does exist, that one will be used and the function will still exit cleanly. Be aware that the value given will be used as the lower boundary for the child table and also influence the name given to the child table. So ensure the timestamp value given is consistent with other children or you may encounter a gap in value coverage.
- `p_start_partition` - When using subpartitioning, allows passing along the start partition value for the subpartition child tables.
- Returns TRUE if any child tables were created for the given timestamptz values. Returns false if no child tables were created.

```
create_partition_id(
    p_parent_table text
    , p_partition_ids bigint[]
    , p_start_partition text DEFAULT NULL
)
RETURNS boolean
```

- This function is used to create child partitions for the given parent table.
- Normally this function is never called manually since partition creation is managed by run_maintenance(). But if you need to force the creation of specific child tables outside of normal maintenance, this function can make it easier.
- `p_parent_table` - parent table to create new child table(s) in.
- `p_partition_ids` - An array of integer values to create children for. If the child table does not exist, it will be created. If it does exist, that one will be used and the function will still exit cleanly. Be aware that the value given will be used as the lower boundary for the child table and also influence the name given to the child table. So ensure the integer value given is consistent with other children or you may encounter a gap in value coverage.
- `p_start_partition` - When using subpartitioning, allows passing along the start partition value for the subpartition child tables.
- Returns TRUE if any child tables were created for the given integer values. Returns false if no child tables were created.

Maintenance Functions

```
run_maintenance(
    p_parent_table text DEFAULT NULL
    , p_analyze boolean DEFAULT false
    , p_jobmon boolean DEFAULT true
)
RETURNS void
```

- Run this function as a scheduled job (cron, etc) to automatically create child tables for partition sets configured to use it.
- You can also use the included background worker (BGW) to have this automatically run for you by PostgreSQL itself. Note that the `p_parent_table` parameter is not available with this method, so if you need to run it for a specific partition set, you must do that manually or scheduled externally. The other parameters have postgresql.conf values that can be set. See BGW section earlier in this documentation.
- This function also maintains the partition retention system for any partitions sets that have it turned on (see **About** and part_config table below).
- Every run checks for all tables listed in the **part_config** table with **automatic_maintenance** set to true and either creates new partitions for them or runs their retention policy.
- By default, all partition sets have automatic_maintenance set to true.
- New partitions are only created if the number of child tables ahead of the current one is less than the premake value, so you can run this more often than needed without fear of needlessly creating more partitions.
- `p_parent_table` - an optional parameter that if passed will cause `run_maintenance()` to be run for ONLY that given table, no matter what automatic_maintenance is set to. High transcation rate tables can cause contention when maintenance is being run for many tables at the same time, so this allows finer control of when partition maintenance is run for specific tables. Note that this will also cause the retention system to only be run for the given table as well.
- `p_analyze` - By default, an analyze is not run after new child tables are created For large partition sets, an analyze can take a while and if `run_maintenance()` is managing several partitions in a single run, this can cause contention while the analyze finishes. However, for constraint exclusion or partition prunning to be fully effective, an analyze must be done from the parent level at some point. Set this to true to have an analyze run on any partition sets that have at least one new child table created. If no new child tables are created in a partition set, no analyze will be run even if this is set to true.
- `p_jobmon` - an optional parameter to control whether `run_maintenance()` itself uses the `pg_jobmon` extension to log what it does. Whether the maintenance of a particular table uses `pg_jobmon` is controlled by the

setting in the **part_config** table and this setting will have no affect on that. Defaults to true if not set.

```
run_maintenance_proc(
    p_wait int DEFAULT 0
    , p_analyze boolean DEFAULT NULL
    , p_jobmon boolean DEFAULT true
)
```

- This procedure can be called instead of the `run_maintenance()` function to cause PostgreSQL to commit after each partition set's maintenance has finished. This greatly reduces contention issues with long running transactions when there are many partition sets to maintain.
- NOTE: The BGW does not yet use this procedure and still uses the standard run_maintenance() function.
- `p_wait` - How many seconds to wait between each partition set's maintenance run. Defaults to 0.
- `p_analyze` - See p_analyze option in run_maintenance.

```
check_default(
    p_exact_count boolean DEFAULT true
)
```

- Run this function to monitor if the default tables of the partition sets that `pg_partman` manages get rows inserted to them.
- Returns a row for each parent/default table along with the number of rows it contains. Returns zero rows if none found.
- `partition_data_time()` & `partition_data_id()` can be used to move data from these parent/default tables into the proper children.
- p_exact_count will tell the function to give back an exact count of how many rows are in each parent if any is found. This is the default if the parameter is left out. If you don't care about an exact count, you can set this to false and it will return if it finds even just a single row in any parent. This can significantly speed up the check if a lot of data ends up in a parent or there are many partitions being managed.

```
show_partitions (
    p_parent_table text
    , p_order text DEFAULT 'ASC'
    , p_include_default boolean DEFAULT false
)
RETURNS TABLE (
    partition_schemaname text
    , partition_tablename text
)
```

- List all child tables of a given partition set managed by pg_partman. Each child table returned as a single row.

- Tables are returned in the order that logically makes sense for the partition interval, not by the locale ordering of their names.
- The default partition can be returned in this result set as well if `p_include_default` is set to true. It is false by default since that is far more common with internal code.
- `p_order` - optional parameter to set the order the child tables are returned in. Defaults to ASCending. Set to 'DESC' to return in descending order. If the default is included, it is always listed first.

```
show_partition_name(
    p_parent_table text
    , p_value text
    , OUT partition_schema text
    , OUT partition_table text
    , OUT suffix_timestamp timestamptz
    , OUT suffix_id bigint
    , OUT table_exists boolean
)
RETURNS record
```

- Given a schema-qualified parent table managed by pg_partman (p_parent_table) and an appropriate value (time or id but given in text form for p_value), return the name of the child partition that that value would exist in.
- If using epoch time partitioning, give the timestamp value, NOT the integer epoch value (use to_timestamp() to convert an epoch value).
- Returns a child table name whether the child table actually exists or not
- Also returns a raw value (suffix_timestamp or suffix_id) for the partition suffix for the given child table
- Also returns a boolean value (table_exists) to say whether that child table actually exists

```
show_partition_info(p_child_table text
    , p_partition_interval text DEFAULT NULL
    , p_parent_table text DEFAULT NULL
    , OUT child_start_time timestamptz
    , OUT child_end_time timestamptz
    , OUT child_start_id bigint
    , OUT child_end_id bigint
    , OUT suffix text
)
RETURNS record
```

- Given a schema-qualified child table name (p_child_table), return the relevant boundary values of that child as well as the suffix appended to the child table name.
- Only works for existing child tables since the boundary values are calculated based on system catalogs of that table.

- `p_partition_interval` - If given, return boundary results based on this interval. If not given, function looks up the interval stored in the part_config table for this partition set.
- `p_parent_table` - Optional argument that can be given when parent_table is known and to avoid a catalog lookup for the parent table associated with p_child_table. Minor performance tuning tweak since this function is used a lot internally.
- `OUT child_start_times & child_end_time` - Function returns values for these output parameters if the partition set is time-based. Otherwise outputs NULL. Note that start value is INCLUSIVE and end value is EXCLUSIVE of the given child table boundaries, exactly as they are defined in the database.
- `OUT child_start_id & child_end_id` - Function returns values for these output parameters if the partition set is integer-based. Otherwise outputs NULL. Note that start value is INCLUSIVE and end value is EXCLUSIVE of the given child table boundaries, exactly as they are defined in the database.
- `OUT suffix` - Outputs the text portition appended to the child table that identifies its contents minus the "_p" (Ex "20230324" OR "920000"). Useful for generating your own suffixes for partitioning similar to how pg_partman does it.

```
dump_partitioned_table_definition(
  p_parent_table text,
  p_ignore_template_table boolean DEFAULT false
)
RETURNS text
```

- Function to return the necessary commands to recreate a partition set in pg_partman for the given parent table (p_parent_table).
- Returns both the `create_parent()` call as well as an UPDATE statement to set additional parameters stored in part_config.
- NOTE: This currently only works with single level partition sets. Looking for contributions to add support for subpartition sets
- `p_ignore_template` - The template table needs to be created before the SQL generated by this function will work properly. If you haven't modified the template table at all then it's safe to pass TRUE here to have the generated SQL tell partman to generate a new template table. But for safety it's preferred to use pg_dump to dump the template tables and restore them prior to using the generated SQL so that you can maintain any template overrides.

```
partition_gap_fill(
    p_parent_table text
)
RETURNS integer
```

- Function to fill in any gaps that may exist in the series of child tables for

a given parent table (p_parent_table).
- Starts from current minimum child table and fills in any gaps encountered based on the partition interval, up to the current maximum child table
- Returns how many child tables are created. Returns 0 if none are created.

```
apply_constraints(
    p_parent_table text
    , p_child_table text DEFAULT NULL
    , p_analyze boolean DEFAULT FALSE
    , p_job_id bigint DEFAULT NULL
)
RETURNS void
```

- Apply constraints to child tables in a given partition set for the columns that are configured (constraint names are all prefixed with "partmanconstr_").
- Note that this does not need to be called manually to maintain custom constraints. The creation of new partitions automatically manages adding constraints to old child tables.
- Columns that are to have constraints are set in the **part_config** table **constraint_cols** array column or during creation with the parameter to `create_parent()`.
- If the `pg_partman` constraints already exists on the child table, the function will cleanly skip over the ones that exist and not create duplicates.
- If the column(s) given contain all NULL values, no constraint will be made.
- If the child table parameter is given, only that child table will have constraints applied.
- If the child table parameter is NOT given, constraints are placed on the last child table older than the `optimize_constraint` value. For example, if the optimize_constraint value is 30, then constraints will be placed on the child table that is 31 back from the current partition (as long as partition pre-creation has been kept up to date).
- If you need to apply constraints to all older child tables, use the `reapply_constraints_proc` procedure. This method has options to make constraint application easier with as little impact on performance as possible.
- The p_job_id parameter is optional. It's for internal use and allows job logging to be consolidated into the original job that called this function if applicable.

```
drop_constraints(
    p_parent_table text
    , p_child_table text
    , p_debug boolean DEFAULT false
)
RETURNS void
```

- Drop constraints that have been created by `pg_partman` for the columns that are configured in *part_config*. This makes it easy to clean up con-

straints if old data needs to be edited and the constraints aren't allowing it.
- Will only drop constraints that begin with `partmanconstr_*` for the given child table and configured columns.
- If you need to drop constraints on all child tables, use the `reapply_constraints_proc` procedure. This has options to make constraint removal easier with as little impact on performance as possible.
- The debug parameter will show you the constraint drop statement that was used.

```
reapply_constraints_proc(
    p_parent_table text
    , p_drop_constraints boolean DEFAULT false
    , p_apply_constraints boolean DEFAULT false
    , p_wait int DEFAULT 0
    , p_dryrun boolean DEFAULT false
)
```

- Procedure to reapply the extra constraint(s) managed by pg_partman (see "Constraint Exclusion" section in "About" section above).
- Calls `drop_constraints()` and/or `apply_constraint()` in a loop, committing after each object is either dropped or added. This helps to avoid long running transaction and contention when doing this on large partition sets.
- Typical usage would be to drop constraints first, edit the data as needed, then apply constraints again.
- `p_parent_table` - Parent table of an already created partition set.
- `p_drop_constraints` - Drop all constraints managed by pg_partman. Drops constraints on all child tables including current & future tables.
- `p_apply_constraints` - Apply constraints on configured columns to all child tables older than the optimize_constraint value.
- `p_wait` - Wait the given number of seconds after a table has had its constraints dropped or applied before moving on to the next.
- `p_dryrun` - Do not actually apply the drop/apply constraint commands when this procedure is run. Just outputs which tables the commands will be applied to as NOTICEs.

```
reapply_privileges(
    p_parent_table text
)
RETURNS void
```

- This function is used to reapply ownership & grants on all child tables based on what the parent table has set.
- Privileges that the parent table has will be granted to all child tables and privilges that the parent does not have will be revoked (with CASCADE).
- Privilges that are checked for are SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, & TRIGGER.

- Be aware that for large partition sets, this can be a very long running operation and is why it was made into a separate function to run independently. Only privileges that are different between the parent & child are applied, but it still has to do system catalog lookups and comparisons for every single child partition and all individual privileges on each.
- `p_parent_table` - parent table of the partition set. Must be schema qualified and match a parent table name already configured in `pg_partman`.

```
stop_sub_partition(
    p_parent_table text
    , p_jobmon boolean DEFAULT true
)
RETURNS boolean
```

- By default, if you undo a child table that is also partitioned, it will not stop additional sibling children of the parent partition set from being subpartitioned unless that parent is also undone. To handle this situation where you may not be removing the parent but don't want any additional subpartitioned children, this function can be used.
- This function simply deletes the parent_table entry from the part_config_sub table. But this gives a predictable, programmatic way to do so and also provides jobmon logging for the operation.

Destruction Functions

```
undo_partition(
    p_parent_table text
    , p_target_table text
    , p_loop_count int DEFAULT 1
    , p_batch_interval text DEFAULT NULL
    , p_keep_table boolean DEFAULT true
    , p_lock_wait numeric DEFAULT 0
    , p_ignored_columns text[] DEFAULT NULL
    , p_drop_cascade boolean DEFAULT false
    , OUT partitions_undone int
    , OUT rows_undone bigint)
RETURNS record
```

- Undo a partition set created by `pg_partman`. This function MOVES the data from the child tables to the given target table.
- If you are trying to un-partition a large amount of data automatically, it is recommended to use the `undo_partition_data()` procedure to do the same thing. This will greatly reduce issues caused by long running transactions and data contention.
- When this function is run, the **`undo_in_progress`** column in the configuration table is set to true. This causes all partition creation and retention management to stop.

- By default, partitions are not DROPPED, they are DETTACHed. This leave previous child tables as empty, independent tables.
- Without setting either batch argument manually, each run of the function will move all the data from a single partition into the target.
- Once all child tables have been uninherited/dropped, the configuration data is removed from `pg_partman` automatically.
- For subpartitioned tables, you may have to start at the lowest level parent table and undo from there then work your way up.
- `p_parent_table` - parent table of the partition set. Must be schema qualified and match a parent table name already configured in `pg_partman`.
- `p_target_table` - A schema-qualified table to move the old partitioned table's data to. Required since a partition table cannot be converted into a non-partitioned table. Schema can be different from original table.
- `p_loop_count` - an optional argument, this sets how many times to move the amount of data equal to the `p_batch_interval` argument (or default partition interval if not set) in a single run of the function. Defaults to 1.
- `p_batch_interval` - optional argument. A time or id interval of how much of the data to move. This can be smaller than the partition interval, allowing for very large partitions to be broken up into smaller commit batches. Defaults to the configured partition interval if not given or if you give an interval larger than the partition interval. Note that the value must be given as text to this parameter.
- `p_keep_table` - an optional argument, setting this to false will cause the old child table to be dropped instead of detached after all of its data has been moved. Note that it takes at least two batches to actually drop a table from the set.
- `p_lock_wait` - optional argument, sets how long in seconds to wait for either the table or a row to be unlocked before timing out. Default is to wait forever.
- `p_ignored_columns` - This option allows for filtering out specific columns when moving data from the child tables to the target table. This is generally only required when using columns with a GENERATED ALWAYS value since directly inserting a value would fail when moving the data. Value is a text array of column names.
- `p_drop_cascade` - Allow undoing subpartition sets from parent tables higher in the inheritance tree. Only applies when `p_keep_tables` is set to false. Note this causes all child tables below a subpartition parent to be dropped when that parent is dropped.
- Returns the number of partitions undone and the number of rows moved to the parent table. The partitions undone value returns -1 if a problem is encountered.

```
undo_partition_proc(
    p_parent_table text
    , p_target_table text DEFAULT NULL
    , p_loop_count int DEFAULT NULL
```

```
    , p_interval text DEFAULT NULL
    , p_keep_table boolean DEFAULT true
    , p_lock_wait int DEFAULT 0
    , p_lock_wait_tries int DEFAULT 10
    , p_wait int DEFAULT 1
    , p_ignored_columns text[] DEFAULT NULL
    , p_drop_cascade boolean DEFAULT false
    , p_quiet boolean DEFAULT false
)
```

- A procedure that can un-partition data in distinct commit batches to avoid long running transactions and data contention issues.
- Calls the undo_partition() function in a loop, committing as needed.
- `p_parent_table` - Parent table of an already created partition set.
- `p_target_table` - Same as the p_target_table option in the undo_partition() function.
- `p_loop_count` - How many times to loop through the value given for p_interval. If p_interval not set, will use default partition interval and undo at most this many partition(s). Procedure commits at the end of each loop (NOT passed as p_batch_count to partitioning function). If not set, all data in the partition set will be moved in a single run of the procedure.
- `p_interval` - Value that is passed on to the undo_partition function as p_batch_interval argument. Use this to set an interval smaller than the partition interval to commit data in smaller batches. Defaults to the partition interval if not given.
- `p_keep_table` - Same as the p_keep_table option in the undo_partition() function.
- `p_lock_wait` - Parameter passed directly through to the underlying undo_partition() function. Number of seconds to wait on rows that may be locked by another transaction. Default is to wait forever (0).
- `p_lock_wait_tries` - Parameter to set how many times the procedure will attempt waiting the amount of time set for p_lock_wait. Default is 10 tries.
- `p_wait` - Cause the procedure to pause for a given number of seconds between commits (batches) to reduce write load
- `p_ignored_columns` - This option allows for filtering out specific columns when moving data from the child tables to the target table. This is generally only required when using columns with a GENERATED ALWAYS value since directly inserting a value would fail when moving the data. Value is a text array of column names.
- `p_drop_cascade` - Allow undoing subpartition sets from parent tables higher in the inheritance tree. Only applies when `p_keep_tables` is set to false. Note this causes all child tables below a subpartition parent to be dropped when that parent is dropped.
- `p_quiet` - Procedures cannot return values, so by default it emits NO-

TICE's to show progress. Set this option to silence these notices.

```
drop_partition_time(
    p_parent_table text
    , p_retention interval DEFAULT NULL
    , p_keep_table boolean DEFAULT NULL
    , p_keep_index boolean DEFAULT NULL
    , p_retention_schema text DEFAULT NULL
    , p_reference_timestamp timestamptz DEFAULT CURRENT_TIMESTAMP
)
RETURNS int
```

- This function is used to drop child tables from a time-based partition set based on a retention policy. By default, the table is just uninherited and not actually dropped. For automatically dropping old tables, it is recommended to use the `run_maintenance()` function with retention configured instead of calling this directly.
- `p_parent_table` - the existing parent table of a time-based partition set. MUST be schema qualified, even if in public schema.
- `p_retention` - optional parameter to give a retention time interval and immediately drop tables containing only data older than the given interval. If you have a retention value set in the config table already, the function will use that, otherwise this will override it. If not, this parameter is required. See the **About** section above for more information on retention settings.
- `p_keep_table` - optional parameter to tell partman whether to keep or drop the table in addition to uninheriting it. TRUE means the table will not actually be dropped; FALSE means the table will be dropped. This function will use the value configured in **part_config** if not explicitly set. This option is ignored if retention_schema is set.
- `p_keep_index` - optional parameter to tell partman whether to keep or drop the indexes of the child table when it is uninherited. TRUE means the indexes will be kept; FALSE means all indexes will be dropped. This function will use the value configured in **part_config** if not explicitly set. This option is ignored if p_keep_table is set to FALSE.
- `p_retention_schema` - optional parameter to tell partman to move a table to another schema instead of dropping it. Set this to the schema you want the table moved to. This function will use the value configured in **part_config** if not explicitly set. If this option is set, the retention `p_keep_table` parameter is ignored.
- `p_reference_timestamp` - optional parameter to tell partman to use a different reference timestamp from which to determine which partitions should be affected, default value is `CURRENT_TIMESTAMP`.
- Returns the number of partitions affected.

```
drop_partition_id(
    p_parent_table text
```

```
    , p_retention bigint DEFAULT NULL
    , p_keep_table boolean DEFAULT NULL
    , p_keep_index boolean DEFAULT NULL
    , p_retention_schema text DEFAULT NULL
)
RETURNS int
```

- This function is used to drop child tables from an integer-based partition set based on a retention policy. By default, the table just uninherited and not actually dropped. For automatically dropping old tables, it is recommended to use the `run_maintenance()` function with retention configured instead of calling this directly.
- `p_parent_table` - the existing parent table of a time-based partition set. MUST be schema qualified, even if in public schema.
- `p_retention` - optional parameter to give a retention integer interval and immediately drop tables containing only data less than the current maximum id value minus the given retention value. If you have a retention value set in the config table already, the function will use that, otherwise this will override it. If not, this parameter is required. See the **About** section above for more information on retention settings.
- `p_keep_table` - optional parameter to tell partman whether to keep or drop the table in addition to uninheriting it. TRUE means the table will not actually be dropped; FALSE means the table will be dropped. This function will use the value configured in **part_config** if not explicitly set. This option is ignored if retention_schema is set.
- `p_keep_index` - optional parameter to tell partman whether to keep or drop the indexes of the child table when it is uninherited. TRUE means the indexes will be kept; FALSE means all indexes will be dropped. This function will use the value configured in **part_config** if not explicitly set. This option is ignored if p_keep_table is set to FALSE.
- `p_retention_schema` - optional parameter to tell partman to move a table to another schema instead of dropping it. Set this to the schema you want the table moved to. This function will use the value configured in `part_config` if not explicitly set. If this option is set, the retention `p_keep_table` parameter is ignored.
- Returns the number of partitions affected.

Tables

`part_config`

Stores all configuration data for partition sets mananged by the extension.

```
parent_table text NOT NULL
, control text NOT NULL
, partition_interval text NOT NULL
, partition_type text NOT NULL
```

```
, premake int NOT NULL DEFAULT 4
, automatic_maintenance text NOT NULL DEFAULT 'on'
, template_table text
, retention text
, retention_schema text
, retention_keep_index boolean NOT NULL DEFAULT true
, retention_keep_table boolean NOT NULL DEFAULT true
, epoch text NOT NULL DEFAULT 'none'
, constraint_cols text[]
, optimize_constraint int NOT NULL DEFAULT 30
, infinite_time_partitions boolean NOT NULL DEFAULT false
, datetime_string text
, jobmon boolean NOT NULL DEFAULT true
, sub_partition_set_full boolean NOT NULL DEFAULT false
, undo_in_progress boolean NOT NULL DEFAULT false
, inherit_privileges boolean DEFAULT false
, constraint_valid boolean DEFAULT true NOT NULL
, subscription_refresh text
, ignore_default_data boolean NOT NULL DEFAULT true
```

- `parent_table`
  - Parent table of the partition set
- `control`
  - Column used as the control for partition constraints. Must be a time or integer based column.
- `partition_interval`
  - Text type value that determines the interval for each partition.
  - Must be a value that can either be cast to the interval or bigint data types.
- `partition_type`
  - Type of partitioning. Must be one of the types mentioned above in the `create_parent()` info.
- `premake`
  - How many partitions to keep pre-made ahead of the current partition. Default is 4.
- `automatic_maintenance`
  - Flag to set whether maintenance is managed automatically when `run_maintenance()` is called without a table parameter or by the background worker process.
  - Current valid values are "on" and "off". Default is "on".
  - When set to off, `run_maintenance()` can still be called on in individual partition set by passing it as a parameter to the function.
- `template_table`
  - The schema-qualified name of the table used as a template for applying any inheritance options not handled by core PostgreSQL partitioning options in PG.

- `retention`
  - Text type value that determines how old the data in a child partition can be before it is dropped.
  - Must be a value that can either be cast to the interval (for time-based partitioning) or bigint (for number partitioning) data types.
  - Leave this column NULL (the default) to always keep all child partitions. See **About** section for more info.
- `retention_schema`
  - Schema to move tables to as part of the retentions system instead of dropping them. Overrides retention_keep_table option.
- `retention_keep_index`
  - Boolean value to determine whether indexes are dropped for child tables that are detached.
  - Default is TRUE. Set to FALSE to have the child table's indexes dropped when it is detached.
- `retention_keep_table`
  - Boolean value to determine whether dropped child tables only detached or actually dropped.
  - Default is TRUE to keep the table and only uninherit it. Set to FALSE to have the child tables removed from the database completely.
- `epoch`
  - Flag the table to be partitioned by time by an integer epoch value instead of a timestamp. See `create_parent()` function for more info. Default 'none'.
- `constraint_cols`
  - Array column that lists columns to have additional constraints applied. See **About** section for more information on how this feature works.
- `optimize_constraint`
  - Manages which old tables get additional constraints set if configured to do so. See **About** section for more info. Default 30.
- `infinite_time_partitions`
  - By default, new partitions in a time-based set will not be created if new data is not inserted to keep an infinite amount of empty tables from being created.
  - If you'd still like new partitions to be made despite there being no new data, set this to TRUE.
  - Defaults to FALSE.
- `datetime_string`
  - For time-based partitioning, this is the datetime format string used when naming child partitions.
- `jobmon`
  - Boolean value to determine whether the `pg_jobmon` extension is used to log/monitor partition maintenance. Defaults to true.
- `sub_partition_set_full`
  - Boolean value to denote that the final partition for a subpartition set has been created. Allows run_maintenance() to run more efficiently

when there are large numbers of subpartition sets.

- `undo_in_progress`
  - Set by the undo_partition functions whenever they are run. If true, this causes all partition creation and retention management by the `run_maintenance()` function to stop. Default is false.
- `inherit_privileges`
  - Sets whether to inherit the ownership/privileges of the parent table to all child tables. Defaults to false and should only be necessary if you need direct access to child tables, by-passing the parent table.
- `constraint_valid`
  - Boolean value that allows the additional constraints that pg_partman can manage for you to be created as NOT VALID. See "Constraint Exclusion" section at the beginning for more details on these constraints. This can allow maintenance to run much quicker on large partition sets since the existing data is not validated before additing the constraint. Newly inserted data is validated, so this is a perfectly safe option to set for data integrity. Note that constraint exclusion WILL NOT work until the constraints are validated. Defaults to true so that constraints are created as VALID. Set to false to set new constraints as NOT VALID.
- `subscription_refresh` - Name of a logical replication subscription to refresh when maintenance runs. If the partition set is subscribed to a publication that will be adding/removing tables and you need your partition set to be aware of these changes, you must name that subscription with this option. Otherwise the subscription will never become aware of the new tables added to the publisher unless you are refreshing the subscription via some other means. See the PG documentation for ALTER SUBSCRIPTION for more info on refreshing subscriptions - https://www.postgresql.org/docs/current/sql-altersubscription.html
- `ignore_default_data` - By default, maintenance will ignore data in the default table when determining whether a new child table should be made. This means that if data is in the default and new child table would contain that data, an error will be thrown. If you need maintenance to acknowledge data in the default to fix a maintenance issue, this can be set to false. Note this can cause gaps in child table coverage, which can made data going into the default even worse, so it should not be left enabled once maintenance issues have been fixed.

`part_config_sub`

- Stores all configuration data for subpartitioned sets managed by `pg_partman`.
- The **`sub_parent`** column is the parent table of the subpartition set and all other columns govern how that parent's children are subpartitioned.
- All other columns work the same exact way as their counterparts in either the **`part_config`** table or as the parameters passed to `create_parent()`.

Scripts

If the extension was installed using *make*, the below script files should have been installed to the PostgreSQL binary directory.

dump_partition.py

- A python script to dump out tables contained in the given schema. Uses pg_dump, creates a SHA-512 hash file of the dump file, and then drops the table.
- When combined with the retention_schema configuration option, provides a way to reliably dump out tables that would normally just be dropped by the retention system.
- Tables are not dropped if pg_dump does not return successfully.
- The connection options for psycopg and pg_dump were separated out due to distinct differences in their requirements depending on your database connection configuration.
- All dump_* option defaults are the same as they would be for pg_dump if they are not given.
- Will work on any given schema, not just the one used to manage `pg_partman` retention.
- `--schema (-n)`: The schema that contains the tables that will be dumped. (Required).
- `--connection (-c)`: Connection string for use by psycopg. Role used must be able to select from pg_catalog.pg_tables in the relevant database and drop all tables in the given schema. Defaults to "host=" (local socket). Note this is distinct from the parameters sent to pg_dump.
- `--output (-o)`: Path to dump file output location. Default is where the script is run from.
- `--dump_database (-d)`: Used for pg_dump, same as its –dbname option or final database name parameter.
- `--dump_host`: Used for pg_dump, same as its –host option.
- `--dump_username`: Used for pg_dump, same as its –username option.
- `--dump_port`: Used for pg_dump, same as its –port option.
- `--pg_dump_path`: Path to pg_dump binary location. Must set if not in current PATH.
- `--Fp`: Dump using pg_dump plain text format. Default is binary custom (-Fc).
- `--nohashfile`: Do NOT create a separate file with the SHA-512 hash of the dump. If dump files are very large, hash generation can possibly take a long time.
- `--nodrop`: Do NOT drop the tables from the given schema after dumping/hashing.
- `--verbose (-v)`: Provide more verbose output.
- `--version`: Print out the minimum version of `pg_partman` this script is meant to work with. The version of `pg_partman` installed may be greater than this.

vacuum_maintenance.py

- A python script to perform additional VACUUM maintenance on a given partition set. The main purpose of this is to provide an easier means of freezing tuples in older partitions that are no longer written to. This allows autovacuum to skip over them safely without causing transaction id wraparound issues. See the PostgreSQL documentation for more information on this maintenance issue: http://www.postgresql.org/docs/current/static/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND.
- Vacuums all child tables in a given partition set who's age(relfrozenxid) is greater than vacuum_freeze_min_age, including the parent table.
- Highly recommend scheduled runs of this script with the –freeze option if you have child tables that never have writes after a certain period of time.
- –parent (-p): Parent table of an already created partition set. (Required)
- –connection (-c): Connection string for use by psycopg. Defaults to "host=" (local socket).
- –freeze (-z): Sets the FREEZE option to the VACUUM command.
- –full (-f): Sets the FULL option to the VACUUM command. Note that –freeze is not necessary if you set this. Recommend reviewing –dryrun before running this since it will lock all tables it runs against, possibly including the parent.
- –vacuum_freeze_min_age (-a): By default the script obtains this value from the system catalogs. By setting this, you can override the value obtained from the database. Note this does not change the value in the database, only the value this script uses.
- –noparent: Normally the parent table is included in the list of tables to vacuum if its age(relfrozenxid) is higher than vacuum_freeze_min_age. Set this to force exclusion of the parent table, even if it meets that criteria.
- –dryrun: Show what the script will do without actually running it against the database. Highly recommend reviewing this before running for the first time.
- –quiet (-q): Turn off all output.
- –debug: Show additional debugging output.

check_unique_constraints.py

- Declarative partitioning has the shortcoming of not allowing a unique constraint if the constraint does not include the partition column. This is often not possible, especially with time-based partitioning. This script is used to check that all rows in a partition set are unique for the given columns.
- Note that on very large partition sets this can be an expensive operation to run that can consume large amounts of storage space. The amount of storage space required is enough to dump out the entire index's column data as a plaintext file.
- If there is a column value that violates the unique constraint, this script

will return those column values along with a count of how many of each value there are. Output can also be simplified to a single, total integer value to make it easier to use with monitoring applications.
- `--parent (-p)`: Parent table of the partition set to be checked. (Required)
- `--column_list (-l)`: Comma separated list of columns that make up the unique constraint to be checked. (Required)
- `--connection (-c)`: Connection string for use by psycopg. Defaults to "host=" (local socket).
- `--temp (-t)`: Path to a writable folder that can be used for temp working files. Defaults system temp folder.
- `--psql`: Full path to psql binary if not in current PATH.
- `--simple`: Output a single integer value with the total duplicate count. Use this for monitoring software that requires a simple value to be checked for.
- `--quiet (-q)`: Suppress all output unless there is a constraint violation found.
- `--version`: Print out the minimum version of `pg_partman` this script is meant to work with. The version of `pg_partman` installed may be greater than this.

- Simple Time Based: 1 Partition Per Day
- Simple Serial ID: 1 Partition Per 10 ID Values
- Partitioning an Existing Table
  - Offline Partitioning
  - Online Partitioning
- Undoing Native Partitioning

This HowTo guide will show you some examples of how to set up simple, single level partitioning. It will also show you several methods to partition data out of a table that has existing data (see Partitioning an Existing Table) and undo the partitioning of an existing partition set (see Undoing Native Partitioning). For more details on what each function does and the additional features in this extension, please see the **pg_partman.md** documentation file.

The examples in this document assume you are running at least 5.0.1 of pg_partman with PostgreSQL 14 or higher.

Simple Time Based: 1 Partition Per Day

For native partitioning, you must start with a parent table that has already been set up to be partitioned in the desired type. Currently pg_partman only supports the RANGE type of partitioning (both for time & id). You cannot turn a non-partitioned table into the parent table of a partitioned set, which can make migration a challenge. This document will show you some techniques for how to manage this later. For now, we will start with a brand new table in this example. Any non-unique indexes can also be added to the parent table in PG11+ and they will automatically be created on all child tables.

```
CREATE SCHEMA IF NOT EXISTS partman_test;

CREATE TABLE partman_test.time_taptest_table
    (col1 int,
    col2 text default 'stuff',
    col3 timestamptz NOT NULL DEFAULT now())
PARTITION BY RANGE (col3);

CREATE INDEX ON partman_test.time_taptest_table (col3);
```

```
\d+ partman_test.time_taptest_table
                                    Partitioned table "partman_test.time_taptest_table"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | Compr
--------+--------------------------+-----------+----------+---------------+----------+-----
 col1   | integer                  |           |          |               | plain    |
 col2   | text                     |           |          | 'stuff'::text | extended |
 col3   | timestamp with time zone |           | not null | now()         | plain    |
Partition key: RANGE (col3)
Indexes:
    "time_taptest_table_col3_idx" btree (col3)
Number of partitions: 0
```

Unique indexes (including primary keys) cannot be created on a natively partitioned parent unless they include the partition key. For time-based partitioning that generally doesn't work out since that would limit only a single timestamp value in each child table. pg_partman helps to manage this by using a template table to manage properties that currently are not supported by native partitioning. Note that this does *not* solve the issue of the constraint *not* being enforced across the entire partition set. See the main documentation to see which properties are managed by the template.

For this example, we are going to manually create the template table first so that when we run `create_parent()` the initial child tables that are created will have a primary key. If you do not supply a template table to pg_partman, it will create one for you in the schema that you installed the extension to. However properties you add to that template are only then applied to newly created child tables after that point. You will have to retroactively apply those properties manually to any child tables that already existed.

```
CREATE TABLE partman_test.time_taptest_table_template (LIKE partman_test.time_taptest_table)

ALTER TABLE partman_test.time_taptest_table_template ADD PRIMARY KEY (col1);
```

```
 \d partman_test.time_taptest_table_template
         Table "partman_test.time_taptest_table_template"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | integer                  |           | not null |
```

```
 col2   | text                        |           |          |
 col3   | timestamp with time zone    |           | not null |
Indexes:
    "time_taptest_table_template_pkey" PRIMARY KEY, btree (col1)

SELECT partman.create_parent(
    p_parent_table := 'partman_test.time_taptest_table'
    , p_control := 'col3'
    , p_interval := '1 day'
    , p_template_table := 'partman_test.time_taptest_table_template'
);
 create_parent
---------------
 t
(1 row)

\d+ partman_test.time_taptest_table
                                      Partitioned table "partman_test.time_taptest_table"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | Compr
--------+--------------------------+-----------+----------+---------------+----------+-----
 col1   | integer                  |           |          |               | plain    |
 col2   | text                     |           |          | 'stuff'::text | extended |
 col3   | timestamp with time zone |           | not null | now()         | plain    |
Partition key: RANGE (col3)
Indexes:
    "time_taptest_table_col3_idx" btree (col3)
Partitions: partman_test.time_taptest_table_p20230324 FOR VALUES FROM ('2023-03-24 00:00:00-
            partman_test.time_taptest_table_p20230325 FOR VALUES FROM ('2023-03-25 00:00:00-
            partman_test.time_taptest_table_p20230326 FOR VALUES FROM ('2023-03-26 00:00:00-
            partman_test.time_taptest_table_p20230327 FOR VALUES FROM ('2023-03-27 00:00:00-
            partman_test.time_taptest_table_p20230328 FOR VALUES FROM ('2023-03-28 00:00:00-
            partman_test.time_taptest_table_p20230329 FOR VALUES FROM ('2023-03-29 00:00:00-
            partman_test.time_taptest_table_p20230330 FOR VALUES FROM ('2023-03-30 00:00:00-
            partman_test.time_taptest_table_p20230331 FOR VALUES FROM ('2023-03-31 00:00:00-
            partman_test.time_taptest_table_p20230401 FOR VALUES FROM ('2023-04-01 00:00:00-
            partman_test.time_taptest_table_default DEFAULT

\d+ partman_test.time_taptest_table_p20230324
                                         Table "partman_test.time_taptest_table_p20230324"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | Compr
--------+--------------------------+-----------+----------+---------------+----------+-----
 col1   | integer                  |           | not null |               | plain    |
 col2   | text                     |           |          | 'stuff'::text | extended |
 col3   | timestamp with time zone |           | not null | now()         | plain    |
Partition of: partman_test.time_taptest_table FOR VALUES FROM ('2023-03-24 00:00:00-07') TO
Partition constraint: ((col3 IS NOT NULL) AND (col3 >= '2023-03-24 00:00:00-07'::timestamp w
Indexes:
```

```
    "time_taptest_table_p20230324_pkey" PRIMARY KEY, btree (col1)
    "time_taptest_table_p20230324_col3_idx" btree (col3)
Access method: heap
```

Simple Serial ID: 1 Partition Per 10 ID Values

For this use-case, the template table is not created manually before calling `create_parent()`. So it shows that if a primary/unique key is added later, it does not apply to the currently existing child tables. That will have to be done manually.

```
CREATE TABLE partman_test.id_taptest_table (
    col1 bigint not null
    , col2 text
    , col3 timestamptz DEFAULT now() not null
    , col4 text)
PARTITION BY RANGE (col1);

CREATE INDEX ON partman_test.id_taptest_table (col1);
```

```
\d+ partman_test.id_taptest_table
                                    Partitioned table "partman_test.id_taptest_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressior
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | bigint                   |           | not null |         | plain    |
 col2   | text                     |           |          |         | extended |
 col3   | timestamp with time zone |           | not null | now()   | plain    |
 col4   | text                     |           |          |         | extended |
Partition key: RANGE (col1)
Indexes:
    "id_taptest_table_col1_idx" btree (col1)
Number of partitions: 0
```

```
SELECT partman.create_parent(
    p_parent_table := 'partman_test.id_taptest_table'
    , p_control := 'col1'
    , p_interval := '10'
);
 create_parent
---------------
 t
(1 row)
```

```
\d+ partman_test.id_taptest_table
                                    Partitioned table "partman_test.id_taptest_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressior
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | bigint                   |           | not null |         | plain    |
```

```
 col2   | text                        |           |          |         | extended |
 col3   | timestamp with time zone    |           | not null | now()   | plain    |
 col4   | text                        |           |          |         | extended |
Partition key: RANGE (col1)
Indexes:
    "id_taptest_table_col1_idx" btree (col1)
Partitions: partman_test.id_taptest_table_p0 FOR VALUES FROM ('0') TO ('10'),
            partman_test.id_taptest_table_p10 FOR VALUES FROM ('10') TO ('20'),
            partman_test.id_taptest_table_p20 FOR VALUES FROM ('20') TO ('30'),
            partman_test.id_taptest_table_p30 FOR VALUES FROM ('30') TO ('40'),
            partman_test.id_taptest_table_p40 FOR VALUES FROM ('40') TO ('50'),
            partman_test.id_taptest_table_default DEFAULT
```

You can see the name of the template table by looking in the pg_partman configuration for that parent table

```
SELECT template_table
FROM partman.part_config
WHERE parent_table = 'partman_test.id_taptest_table';

                template_table
-----------------------------------------------
 partman.template_partman_test_id_taptest_table
(1 row)
```

```
ALTER TABLE partman.template_partman_test_id_taptest_table ADD PRIMARY KEY (col2);
```

Now if we add some data and run maintenance again to create new child tables. . .

```
INSERT INTO partman_test.id_taptest_table (col1, col2) VALUES (generate_series(1,20), genera

CALL partman.run_maintenance_proc();

\d+ partman_test.id_taptest_table
                                 Partitioned table "partman_test.id_taptest_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressio
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | bigint                   |           | not null |         | plain    |
 col2   | text                     |           |          |         | extended |
 col3   | timestamp with time zone |           | not null | now()   | plain    |
 col4   | text                     |           |          |         | extended |
Partition key: RANGE (col1)
Indexes:
    "id_taptest_table_col1_idx" btree (col1)
Partitions: partman_test.id_taptest_table_p0 FOR VALUES FROM ('0') TO ('10'),
            partman_test.id_taptest_table_p10 FOR VALUES FROM ('10') TO ('20'),
            partman_test.id_taptest_table_p20 FOR VALUES FROM ('20') TO ('30'),
            partman_test.id_taptest_table_p30 FOR VALUES FROM ('30') TO ('40'),
```

```
            partman_test.id_taptest_table_p40 FOR VALUES FROM ('40') TO ('50'),
            partman_test.id_taptest_table_p50 FOR VALUES FROM ('50') TO ('60'),
            partman_test.id_taptest_table_p60 FOR VALUES FROM ('60') TO ('70'),
            partman_test.id_taptest_table_default DEFAULT
```

. . . you'll see that only the new child tables (p50 & p60) have that primary key and the original tables do not (p40 and earlier).

```
keith=# \d partman_test.id_taptest_table_p40
             Table "partman_test.id_taptest_table_p40"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | bigint                   |           | not null |
 col2   | text                     |           |          |
 col3   | timestamp with time zone |           | not null | now()
 col4   | text                     |           |          |
Partition of: partman_test.id_taptest_table FOR VALUES FROM ('40') TO ('50')
Indexes:
    "id_taptest_table_p40_col1_idx" btree (col1)

keith=# \d partman_test.id_taptest_table_p50
             Table "partman_test.id_taptest_table_p50"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | bigint                   |           | not null |
 col2   | text                     |           | not null |
 col3   | timestamp with time zone |           | not null | now()
 col4   | text                     |           |          |
Partition of: partman_test.id_taptest_table FOR VALUES FROM ('50') TO ('60')
Indexes:
    "id_taptest_table_p50_pkey" PRIMARY KEY, btree (col2)
    "id_taptest_table_p50_col1_idx" btree (col1)

keith=# \d partman_test.id_taptest_table_p60
             Table "partman_test.id_taptest_table_p60"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | bigint                   |           | not null |
 col2   | text                     |           | not null |
 col3   | timestamp with time zone |           | not null | now()
 col4   | text                     |           |          |
Partition of: partman_test.id_taptest_table FOR VALUES FROM ('60') TO ('70')
Indexes:
    "id_taptest_table_p60_pkey" PRIMARY KEY, btree (col2)
    "id_taptest_table_p60_col1_idx" btree (col1)
```

Add them manually:

```
ALTER TABLE partman_test.id_taptest_table_p0 ADD PRIMARY KEY (col2);
ALTER TABLE partman_test.id_taptest_table_p10 ADD PRIMARY KEY (col2);
ALTER TABLE partman_test.id_taptest_table_p20 ADD PRIMARY KEY (col2);
ALTER TABLE partman_test.id_taptest_table_p30 ADD PRIMARY KEY (col2);
ALTER TABLE partman_test.id_taptest_table_p40 ADD PRIMARY KEY (col2);
```

Partitioning an Existing Table

Partitioning an existing table with native partitioning is not as straight forward as just altering a table. As stated above, you cannot turn an already existing table into the parent table of a native partition set. The parent of a native partitioned table must be declared partitioned at the time of its creation. However, there are still methods to take an existing table and partition it natively. Two of those are presented below.

One tip that may possibly help with the speed/load of partitioning a table would be to run a CLUSTER on the original source table using the partition key's index shortly before the partitioning is done. Since the data will be read sequentially from the source table, having it be in order could potentially help with IO efficiency on very large tables. See the PostgreSQL documentation - https://www.postgresql.org/docs/current/sql-cluster.html

Offline Partitioning This method is being labelled "offline" because, during points in this process, the data is not accessible to both the new and old table from a single object. The data is moved from the original table to a brand new table. The advantage of this method is that you can move your data in much smaller batches than even the target partition size, which can be a huge efficiency advantage for very large partition sets (you can commit in batches of several thousand vs several million). There are also less object renaming steps as we'll see in the online partitioning method next.

IMPORTANT NOTE REGARDING FOREIGN KEYS

Taking the partitioned table offline is the only method that realistically works when you have foreign keys TO the table being partitioned. Since a brand new table must be created no matter what, the foreign key must also be recreated, so an outage involving all tables that are part of the FK relationship must be taken. A shorter outage may be possible with the online method below, but if you have to take an outage, this offline method is easier.

Here is the original table with some generated data:

```
CREATE TABLE public.original_table (
    col1 bigint not null
    , col2 text not null
    , col3 timestamptz DEFAULT now()
    , col4 text);
```

```
CREATE INDEX ON public.original_table (col1);

INSERT INTO public.original_table (col1, col2, col3, col4) VALUES (generate_series(1,100000)
```

First, the original table should be renamed so the partitioned table can be made with the original table's name. This makes it so that, when the child tables are created, they have names that are associated with the original table name.

```
ALTER TABLE public.original_table RENAME to old_nonpartitioned_table;
```

We'll use the serial partitioning example from above. The initial setup is exactly the same, creating a brand new table that will be the parent and then running `create_parent()` on it. We'll make the interval slightly larger this time. Also, make sure you've applied all the same original properties to this new table that the old table had: privileges, constraints, defaults, indexes, etc. Privileges are especially important to make sure they match so that all users of the table will continue to work after the conversion.

Note that primary keys/unique indexes cannot be applied to a partitioned parent unless the partition key is part of it. In this case that would work, however it's likely not the intention since that would mean only one row per value is allowed and that would mean only 10,000 rows could ever exist in each child table. Partitioning is definitely not needed in that case then. The next example of online partitioning will show how to handle when you need a primary key for a column that is not part of the partition key.

```
CREATE TABLE public.original_table (
    col1 bigint not null
    , col2 text not null
    , col3 timestamptz DEFAULT now()
    , col4 text)
PARTITION BY RANGE (col1);

CREATE INDEX ON public.original_table (col1);

SELECT partman.create_parent(
    p_parent_table := 'public.original_table'
    , p_control := 'col1'
    , p_interval := '10000'
);

\d+ original_table;
                                        Partitioned table "public.original_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressio
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | bigint                   |           | not null |         | plain    |
 col2   | text                     |           | not null |         | extended |
 col3   | timestamp with time zone |           |          | now()   | plain    |
 col4   | text                     |           |          |         | extended |
```

```
Partition key: RANGE (col1)
Indexes:
    "original_table_col1_idx1" btree (col1)
    "original_table_col1_idx2" btree (col1)
Partitions: original_table_p0 FOR VALUES FROM ('0') TO ('10000'),
            original_table_p10000 FOR VALUES FROM ('10000') TO ('20000'),
            original_table_p20000 FOR VALUES FROM ('20000') TO ('30000'),
            original_table_p30000 FOR VALUES FROM ('30000') TO ('40000'),
            original_table_p40000 FOR VALUES FROM ('40000') TO ('50000'),
            original_table_default DEFAULT
```

If you happened to be using IDENTITY columns, or you created a new sequence for the new partitioned table, you'll want to get the value of those old sequences and reset the new sequences to start with those old values. Some tips for doing that are covered in the Online Partitioning section below. If you just re-used the same sequence on the new partitioned table, you should be fine.

Now we can use the `partition_data_proc()` procedure to migrate our data from the old table to the new table. And we're going to do it in 1,000 row increments vs the 10,000 interval that the partition set has. The batch value is used to tell it how many times to run through the given interval; the default value of 1 only makes a single child table. Since we want to partition all of the data, just give it a number equal to or greater than the expected child table count. This procedure has an option where you can tell it the source of the data, which is how we're going to migrate the data from the old table. Without setting this option, it attempts to clean the data out of the DEFAULT partition (which we'll see an example of next).

```
keith=# CALL partman.partition_data_proc(
    p_parent_table := 'public.original_table'
    , p_loop_count := 200
    , p_interval := '1000'
    , p_source_table := 'public.old_nonpartitioned_table'
);
NOTICE:  Loop: 1, Rows moved: 1000
NOTICE:  Loop: 2, Rows moved: 1000
NOTICE:  Loop: 3, Rows moved: 1000
NOTICE:  Loop: 4, Rows moved: 1000
NOTICE:  Loop: 5, Rows moved: 1000
NOTICE:  Loop: 6, Rows moved: 1000
NOTICE:  Loop: 7, Rows moved: 1000
NOTICE:  Loop: 8, Rows moved: 1000
NOTICE:  Loop: 9, Rows moved: 1000
NOTICE:  Loop: 10, Rows moved: 999
NOTICE:  Loop: 11, Rows moved: 1000
NOTICE:  Loop: 12, Rows moved: 1000
[...]
```

```
NOTICE:  Loop: 99, Rows moved: 1000
NOTICE:  Loop: 100, Rows moved: 1000
NOTICE:  Loop: 101, Rows moved: 1
NOTICE:  Total rows moved: 100000
NOTICE:  Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning c
```

```
VACUUM ANALYZE public.original_table;
```

Again, doing the commits in smaller batches like this can avoid transactions with huge row counts and long runtimes when you're partitioning a table that may have billions of rows. It's always advisable to avoid long running transactions to allow PostgreSQL's autovacuum process to work efficiently for the rest of the database. However, doing smaller batches per loop can cause the process to partition the data to take longer. You will have to find the balance between the load on your database and time required.

Using the `partition_data_proc()` PROCEDURE vs the `partition_data_id()` FUNCTION allows those commit batches. Functions in PostgreSQL always run entirely in a single transaction, even if you may tell it to do things in batches inside the function.

Now if we check our original table, it is empty

```
SELECT count(*) FROM old_nonpartitioned_table;
 count
-------
     0
(1 row)
```

And the new, partitioned table with the original name has all the data and child tables created

```
SELECT count(*) FROM original_table;
 count
--------
 100000
(1 row)
```

```
 \d+ public.original_table
                                        Partitioned table "public.original_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressior
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | bigint                   |           | not null |         | plain    |
 col2   | text                     |           | not null |         | extended |
 col3   | timestamp with time zone |           |          | now()   | plain    |
 col4   | text                     |           |          |         | extended |
Partition key: RANGE (col1)
```

```
Indexes:
    "original_table_col1_idx" btree (col1)
Partitions: original_table_p0 FOR VALUES FROM ('0') TO ('10000'),
            original_table_p10000 FOR VALUES FROM ('10000') TO ('20000'),
            original_table_p100000 FOR VALUES FROM ('100000') TO ('110000'),
            original_table_p20000 FOR VALUES FROM ('20000') TO ('30000'),
            original_table_p30000 FOR VALUES FROM ('30000') TO ('40000'),
            original_table_p40000 FOR VALUES FROM ('40000') TO ('50000'),
            original_table_p50000 FOR VALUES FROM ('50000') TO ('60000'),
            original_table_p60000 FOR VALUES FROM ('60000') TO ('70000'),
            original_table_p70000 FOR VALUES FROM ('70000') TO ('80000'),
            original_table_p80000 FOR VALUES FROM ('80000') TO ('90000'),
            original_table_p90000 FOR VALUES FROM ('90000') TO ('100000'),
            original_table_default DEFAULT

SELECT count(*) FROM original_table_p10000;
 count
-------
 10000
(1 row)
```

Now you should be able to start using your table the same as you were before!

Online Partitioning Sometimes it is not possible to take the table offline for an extended period of time to migrate it to a partitioned table. Below is one method to allow this to be done online. It's not as flexible as the offline method, but should allow a very minimal downtime and be mostly transparent to the end users of the table.

As mentioned above, these methods DO NOT account for there being foreign keys TO the original table. You can create foreign keys FROM the original table on the new partitioned table and things should work as expected. However, if you have foreign keys coming in to the table, I'm not aware of any migration method that does not require an outage to drop the original foreign keys and recreate them against the new partitioned table.

This will be a daily, time-based partition set with an IDENTITY sequence as the primary key

```
CREATE TABLE public.original_table (
    col1 bigint not null PRIMARY KEY GENERATED ALWAYS AS IDENTITY
    , col2 text not null
    , col3 timestamptz DEFAULT now() not null
    , col4 text);

CREATE INDEX CONCURRENTLY ON public.original_table (col3);
```

```
INSERT INTO public.original_table (col2, col3, col4) VALUES ('stuff', generate_series(now()
```

The process is still initially the same as the offline method since you cannot turn an existing table into the parent table of a partition set. However it is critical that all constraints, privileges, defaults and any other properties be applied to the new parent table before you move on to the next step of swapping the table names around.

```
CREATE TABLE public.new_partitioned_table (
    col1 bigint not null GENERATED BY DEFAULT AS IDENTITY
    , col2 text not null
    , col3 timestamptz DEFAULT now() not null
    , col4 text) PARTITION BY RANGE (col3);

CREATE INDEX ON public.new_partitioned_table (col3);
```

You'll notice I did not set "col1" as a primary key here. That is because we cannot.

```
CREATE TABLE public.new_partitioned_table (
    col1 bigint not null PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY
    , col2 text not null
    , col3 timestamptz DEFAULT now() not null
    , col4 text) PARTITION BY RANGE (col3);
ERROR:  unique constraint on partitioned table must include all partitioning columns
DETAIL:  PRIMARY KEY constraint on table "new_partitioned_table" lacks column "col3" which i
```

pg_partman does have a mechanism to still apply primary/unique keys that are not part of the partition column. Just be aware that they are NOT enforced across the entire partition set; only for the individual partition. This is done with a template table. And to ensure the keys are applied when the initial child tables are created, that template table must be pre-created and its name supplied to the `create_parent()` call. We're going to use the original table as the basis and give a name similar to that so it makes sense after the name swapping later.

Another important note is that we changed the IDENTITY column from GENERATED ALWAYS to GENERATED BY DEFAULT. This is because we need to move existing values for that identity column into place. ALWAYS generally prevents manually entered values.

```
CREATE TABLE public.original_table_template (LIKE public.original_table);

ALTER TABLE public.original_table_template ADD PRIMARY KEY (col1);
```

If you do not pre-create a template table, pg_partman will always create one for you in the same schema that the extension was installed into. You can see its name by looking at the `template_table` column in the `part_config` table. However, if you add the index onto that template table after the `create_parent()` call, the already existing child tables will not have that index applied and you will

have to go back and do that manually. However, any new child tables create after that will have the index.

The tricky part here is that we cannot yet have any child tables in the partition set that match data that currently exists in the original table. This is because we're going to be adding the old table as the DEFAULT table to our new partition table. If the DEFAULT table contains any data that matches a current child table's constraints, PostgreSQL will not allow that table to be added. So, with the below `create_parent()` call, we're going to start the partition set well ahead of the data we inserted and disable the automatic creation of a default table. In your case you will have to look at your current data set and pick a value well ahead of the current working set of data that may get inserted before you are able to run the table name swap process below. We're also setting the premake value to a low value to avoid having to rename too many child tables later. We'll increase premake back up to the default later (or you can set it to whatever you require).

```
SELECT min(col3), max(col3) FROM original_table;
```

```
              min               |              max
--------------------------------+--------------------------------
 2023-03-21 11:09:31.980586-07 | 2023-03-28 11:09:31.980586-07
```

```
SELECT partman.create_parent(
    p_parent_table := 'public.new_partitioned_table'
    , p_control := 'col3'
    , p_interval := '1 day'
    , p_template_table:= 'public.original_table_template'
    , p_premake := 1
    , p_start_partition := (CURRENT_TIMESTAMP+'2 days'::interval)::text
    , p_default_table := false
);
```

The state of the new partitioned table should now look something like this. The current date for when this HowTo was written is given for reference:

```
SELECT CURRENT_TIMESTAMP;
       current_timestamp
-------------------------------
 2023-03-28 11:23:55.971402-07

\d+ new_partitioned_table;
                                                  Partitioned table "public.new_partitioned_t
 Column |           Type           | Collation | Nullable |             Default
--------+--------------------------+-----------+----------+---------------------------------
 col1   | bigint                   |           | not null | generated by default as identity
 col2   | text                     |           | not null |
 col3   | timestamp with time zone |           | not null | now()
```

```
 col4   | text                        |           |          |
Partition key: RANGE (col3)
Indexes:
    "new_partitioned_table_col3_idx" btree (col3)
Partitions: new_partitioned_table_p20230330 FOR VALUES FROM ('2023-03-30 00:00:00-07') TO ('
```

You will need to update the `part_config` table to have the original table name. You can also update the template table if you didn't manually create one yourself, just be sure to both rename the table and update the `part_config` table as well. We'll reset the premake to the default value here as well.

```
UPDATE partman.part_config SET parent_table = 'public.original_table', premake = 4 WHERE par
UPDATE 1
```

The next step, which is actually multiple steps in a single transaction, is the only outage of any significance that needs to be anticipated.

1. BEGIN transaction
2. Take an exclusive lock on the original table and the new table to ensure no gaps exist for data to be misrouted
3. If using an IDENTITY column, get the original last value
4. Rename the original table to the DEFAULT table name for the partition set
5. If using an IDENTITY column, DROP the IDENTITY from the old table
6. Rename the new table to the original table's name and rename child tables & sequence to match.
7. If using an IDENTITY column, reset the new table's identity to the latest value so any new inserts pick up where old table's sequence left off.
8. Add original table as the DEFAULT for the partition set
9. COMMIT transaction

If you are using an IDENTITY column, it is important to get its last value while the original table is locked and BEFORE you drop the old identity. Then use that returned value in the statement to RESET the IDENTITY column in the new table. You may need to increment the value returned a bit to stay ahead of current table usage, so this can be a little tricky with a very busy table, especially if your sequence values must remain unbroken. A query to obtain this is provided in the SQL statements below. If you are not using an IDENTITY, you can simply ignore those steps.

If at any point there is a problem with one of these mini-steps, just perform a ROLLBACK and you should return to the previous state and allow your original table to work as it was before.

```
BEGIN;

LOCK TABLE public.original_table IN ACCESS EXCLUSIVE MODE;
LOCK TABLE public.new_partitioned_table IN ACCESS EXCLUSIVE MODE;
```

```
SELECT max(col1)+1 FROM public.original_table;

ALTER TABLE public.original_table RENAME TO original_table_default;

-- IF using an IDENTITY column
ALTER TABLE public.original_table_default ALTER col1 DROP IDENTITY;

ALTER TABLE public.new_partitioned_table RENAME TO original_table;
ALTER TABLE public.new_partitioned_table_p20230330 RENAME TO original_table_p20230330;

-- IF using an IDENTITY column
ALTER SEQUENCE public.new_partitioned_table_col1_seq RENAME TO original_table_col1_seq;

-- IF using an IDENTITY column
ALTER TABLE public.original_table ALTER col1 RESTART WITH <<<VALUE OBTAINED ABOVE>>>;

ALTER TABLE public.original_table ATTACH PARTITION public.original_table_default DEFAULT;

COMMIT; or ROLLBACK;
```

Once COMMIT is run, the new partitioned table should now take over from the original non-partitioned table. And as long as all the properties have been applied to the new table, it should be working without any issues. Any new data coming in should either be going to the relevant child table, or if it doesn't happen to exist yet, it should go to the DEFAULT. The latter is not an issue since...

The next step is to partition the data out of the default. You DO NOT want to leave data in the default partition set for any length of time and especially leave any significant amount of data. If you look at the constraint that exists on a partitioned default, it is basically the anti-constraint of all other child tables. And when a new child table is added, PostgreSQL manages updating that default constraint as needed. But it must check if any data that should belong in that new child table already exists in the default. If it finds any, it will fail. But more importantly, it has to check EVERY entry in the default which can take quite a long time even with an index if there are billions of rows. During this check, there is an exclusive lock on the entire partition set.

The `partition_data_proc()` can handle moving the data out of the default. However, it cannot move data in any interval smaller than the partition interval when moving data out of the DEFAULT. This is related to what was just mentioned: You cannot add a child table to a partition set if that new child table's constraint covers data that already exists in the default.

pg_partman handles this by first moving all the data for a given child table out to a temporary table, then creating the child table, and then moving the data from the temp table into the new child table. Since we're moving data out of the DEFAULT and we cannot use a smaller interval, the only parameter

that we need to pass is a batch size. The default batch size of 1 would only make a single child table then stop. If you want to move all the data in a single call, just pass a value large enough to cover the expected number of child tables. However, with a live table and LOTS rows, this could potentially generate A LOT of WAL files, especially since this method doubles the number of writes vs the offline method (default -> temp -> child table). So if keeping control of your disk usage is a concern, just give a smaller batch value and then give PostgreSQL some time to run through a few CHECKPOINTs and clean up its own WAL before moving on to the next batch.

```
CALL partman.partition_data_proc('public.original_table', p_loop_count := 200);
NOTICE:  Loop: 1, Rows moved: 134
NOTICE:  Loop: 2, Rows moved: 288
NOTICE:  Loop: 3, Rows moved: 288
NOTICE:  Loop: 4, Rows moved: 288
NOTICE:  Loop: 5, Rows moved: 288
NOTICE:  Loop: 6, Rows moved: 288
NOTICE:  Loop: 7, Rows moved: 288
NOTICE:  Loop: 8, Rows moved: 155
NOTICE:  Total rows moved: 2017
NOTICE:  Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning c

VACUUM ANALYZE original_table;
VACUUM
```

If you were using an IDENTITY column with GENERATED ALWAYS before, you'll want to change the identity on the partitioned table back to that from the current setting of BY DEFAULT

```
ALTER TABLE public.original_table ALTER col1 SET GENERATED ALWAYS;
```

Now, double-check that the child table creation was performed as expected. There is one missing here if you look closely but we'll discuss that below.

```
\d+ original_table
                                                  Partitioned table "public.original_table"
 Column |           Type           | Collation | Nullable |           Default            | S
--------+--------------------------+-----------+----------+------------------------------+--
 col1   | bigint                   |           | not null | generated always as identity | p
 col2   | text                     |           | not null |                              | e
 col3   | timestamp with time zone |           | not null | now()                        | p
 col4   | text                     |           |          |                              | e
Partition key: RANGE (col3)
Indexes:
    "new_partitioned_table_col3_idx" btree (col3)
Partitions: original_table_p20230321 FOR VALUES FROM ('2023-03-21 00:00:00-07') TO ('2023-03
            original_table_p20230322 FOR VALUES FROM ('2023-03-22 00:00:00-07') TO ('2023-03
            original_table_p20230323 FOR VALUES FROM ('2023-03-23 00:00:00-07') TO ('2023-03
```

```
            original_table_p20230324 FOR VALUES FROM ('2023-03-24 00:00:00-07') TO ('2023-03
            original_table_p20230325 FOR VALUES FROM ('2023-03-25 00:00:00-07') TO ('2023-03
            original_table_p20230326 FOR VALUES FROM ('2023-03-26 00:00:00-07') TO ('2023-03
            original_table_p20230327 FOR VALUES FROM ('2023-03-27 00:00:00-07') TO ('2023-03
            original_table_p20230328 FOR VALUES FROM ('2023-03-28 00:00:00-07') TO ('2023-03
            original_table_p20230330 FOR VALUES FROM ('2023-03-30 00:00:00-07') TO ('2023-03
            original_table_default DEFAULT
```

And now to ensure any new data coming in is going to proper child tables and not the default, run maintenance on the new partitioned table to ensure the current premake partitions are created

```
SELECT partman.run_maintenance('public.original_table');

\d+ original_table
                                                  Partitioned table "public.original_table"
 Column |           Type           | Collation | Nullable |           Default            | S
--------+--------------------------+-----------+----------+------------------------------+--
 col1   | bigint                   |           | not null | generated always as identity | p
 col2   | text                     |           | not null |                              | e
 col3   | timestamp with time zone |           | not null | now()                        | p
 col4   | text                     |           |          |                              | e
Partition key: RANGE (col3)
Indexes:
    "new_partitioned_table_col3_idx" btree (col3)
Partitions: original_table_p20230321 FOR VALUES FROM ('2023-03-21 00:00:00-07') TO ('2023-03
            original_table_p20230322 FOR VALUES FROM ('2023-03-22 00:00:00-07') TO ('2023-03
            original_table_p20230323 FOR VALUES FROM ('2023-03-23 00:00:00-07') TO ('2023-03
            original_table_p20230324 FOR VALUES FROM ('2023-03-24 00:00:00-07') TO ('2023-03
            original_table_p20230325 FOR VALUES FROM ('2023-03-25 00:00:00-07') TO ('2023-03
            original_table_p20230326 FOR VALUES FROM ('2023-03-26 00:00:00-07') TO ('2023-03
            original_table_p20230327 FOR VALUES FROM ('2023-03-27 00:00:00-07') TO ('2023-03
            original_table_p20230328 FOR VALUES FROM ('2023-03-28 00:00:00-07') TO ('2023-03
            original_table_p20230330 FOR VALUES FROM ('2023-03-30 00:00:00-07') TO ('2023-03
            original_table_p20230331 FOR VALUES FROM ('2023-03-31 00:00:00-07') TO ('2023-04
            original_table_p20230401 FOR VALUES FROM ('2023-04-01 00:00:00-07') TO ('2023-04
            original_table_default DEFAULT
```

Before this, depending on the child tables that were generated and the new data coming in, there may have been some data that still went to the default. You can check for that with a function that comes with pg_partman:

```
SELECT * FROM partman.check_default(p_exact_count := true);
```

If you don't pass "true" to the function, it just returns a 1 or 0 to indicate if any data exists in any default. This is convenient for monitoring situations and it can also be quicker since it stops checking as soon as it finds data in any child table. However, in this case we want to see exactly what our situation is, so

passing true will give us an exact count of how many rows are left in the default.

You'll also notice that there is a child table missing in the set above (March 29, 2023). This is because we set the partition table to start 2 days ahead and we didn't have any data for that date in the original table. You can fix this one of two ways:

1. Wait for data for that time period to be inserted and once you're sure that interval is done, partition the data out of the DEFAULT the same way we did before.
2. Run the `partition_gap_fill()` function to fill any gaps immediately:

```
SELECT * FROM partman.partition_gap_fill('public.original_table');
 partition_gap_fill
--------------------
                  1
(1 row)

keith=# \d+ original_table
                                                  Partitioned table "public.original_table"
 Column |           Type           | Collation | Nullable |           Default            | S
--------+--------------------------+-----------+----------+------------------------------+--
 col1   | bigint                   |           | not null | generated always as identity | p
 col2   | text                     |           | not null |                              | e
 col3   | timestamp with time zone |           | not null | now()                        | p
 col4   | text                     |           |          |                              | e
Partition key: RANGE (col3)
Indexes:
    "new_partitioned_table_col3_idx" btree (col3)
Partitions: original_table_p20230321 FOR VALUES FROM ('2023-03-21 00:00:00-07') TO ('2023-03
            original_table_p20230322 FOR VALUES FROM ('2023-03-22 00:00:00-07') TO ('2023-03
            original_table_p20230323 FOR VALUES FROM ('2023-03-23 00:00:00-07') TO ('2023-03
            original_table_p20230324 FOR VALUES FROM ('2023-03-24 00:00:00-07') TO ('2023-03
            original_table_p20230325 FOR VALUES FROM ('2023-03-25 00:00:00-07') TO ('2023-03
            original_table_p20230326 FOR VALUES FROM ('2023-03-26 00:00:00-07') TO ('2023-03
            original_table_p20230327 FOR VALUES FROM ('2023-03-27 00:00:00-07') TO ('2023-03
            original_table_p20230328 FOR VALUES FROM ('2023-03-28 00:00:00-07') TO ('2023-03
            original_table_p20230329 FOR VALUES FROM ('2023-03-29 00:00:00-07') TO ('2023-03
            original_table_p20230330 FOR VALUES FROM ('2023-03-30 00:00:00-07') TO ('2023-03
            original_table_p20230331 FOR VALUES FROM ('2023-03-31 00:00:00-07') TO ('2023-04
            original_table_p20230401 FOR VALUES FROM ('2023-04-01 00:00:00-07') TO ('2023-04
            original_table_default DEFAULT
```

You can see that created the missing table for March 29.

At this point your new partitioned table should already have been in use and working without any issues!

```
INSERT INTO original_table (col2, col3, col4) VALUES ('newstuff', now(), 'newstuff');
```

```
INSERT INTO original_table (col2, col3, col4) VALUES ('newstuff', now(), 'newstuff');
SELECT * FROM original_table ORDER BY col1 DESC limit 5;
```

Undoing Native Partitioning

Just as a normal table cannot be converted to a natively partitioned table, the opposite is also true. To undo native partitioning, you must move the data to a brand new table. There may be a way to do this online, but I do not currently have such a method planned out. If someone would like to submit a method or request that I look into it further, please feel free to make an issue on Github. The below method shows how to undo the daily partitioned example above, including handling an IDENTITY column if necessary.

First, we create a new table to migrate the data to. We can set a primary key, or any unique indexes that were made on the template. If there are any identity columns, they have to set the method to `GENERATED BY DEFAULT` since we will be adding values in manually as part of the migration. If it needs to be `ALWAYS`, this can be changed later.

If this table is going to continue to be used the same as the previous partitioned table, ensure all privileges, constraints & indexes are created on this table as well. Index & constraint creation can be delayed until after the data has been moved to speed up the migration.

```
CREATE TABLE public.new_regular_table (
    col1 bigint not null GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY
    , col2 text not null
    , col3 timestamptz DEFAULT now() not null
    , col4 text);

CREATE INDEX ON public.new_regular_table (col3);
```

Now we can use the `undo_partition_proc()` procedure to move the data out of our partitioned table to the regular table. We can even chose a smaller interval size for this as well to reduce the transaction runtime for each batch. The batch size is a default of 1, which would only run the given interval one time. We want to undo the entire thing with one call, so pass a number at high enough to run through all batches. It will stop when all the data has been moved, even if you passed a higher batch number. We also don't need to keep the old child tables once they're empty, so that is set to false. See the documentation for more information on other options for the undo functions/procedure.

```
CALL partman.undo_partition_proc(
    p_parent_table := 'public.original_table'
    , p_target_table := 'public.new_regular_table'
    , p_interval := '1 hour'::text
    , p_loop_count := 500
    , p_keep_table := false);
```

```
NOTICE:  Moved 13 row(s) to the target table. Removed 1 partitions.
NOTICE:  Batch: 1, Partitions undone this batch: 1, Rows undone this batch: 13
NOTICE:  Moved 13 row(s) to the target table. Removed 0 partitions.
NOTICE:  Batch: 2, Partitions undone this batch: 0, Rows undone this batch: 13
NOTICE:  Moved 13 row(s) to the target table. Removed 0 partitions.
NOTICE:  Batch: 3, Partitions undone this batch: 0, Rows undone this batch: 13
NOTICE:  Moved 13 row(s) to the target table. Removed 0 partitions.

[...]
NOTICE:  Batch: 160, Partitions undone this batch: 0, Rows undone this batch: 13
NOTICE:  Moved 12 row(s) to the target table. Removed 0 partitions.
NOTICE:  Batch: 161, Partitions undone this batch: 0, Rows undone this batch: 12
NOTICE:  Moved 2 row(s) to the target table. Removed 1 partitions.
NOTICE:  Batch: 162, Partitions undone this batch: 1, Rows undone this batch: 2
NOTICE:  Moved 0 row(s) to the target table. Removed 4 partitions.
NOTICE:  Total partitions undone: 13, Total rows moved: 2019
NOTICE:  Ensure to VACUUM ANALYZE the old parent & target table after undo has finished

VACUUM ANALYZE original_table;

VACUUM ANALYZE new_regular_table;
```

Now object names can be swapped around and the identity sequence reset and method changed if needed. Be sure to grab the original sequence value and use that when resetting.

```
SELECT max(col1)+1 FROM public.new_regular_table;

ALTER TABLE original_table RENAME TO old_partitioned_table;
ALTER SEQUENCE original_table_col1_seq RENAME TO old_partitioned_table_col1_seq;

ALTER TABLE new_regular_table RENAME TO original_table;
ALTER SEQUENCE new_regular_table_col1_seq RENAME TO original_table_col1_seq;
ALTER TABLE public.original_table ALTER col1 RESTART WITH <<<VALUE OBTAINED ABOVE>>>;
ALTER TABLE public.original_table ALTER col1 SET GENERATED ALWAYS;

INSERT INTO original_table (col2, col3, col4) VALUES ('newstuff', now(), 'newstuff');
INSERT INTO original_table (col2, col3, col4) VALUES ('newstuff', now(), 'newstuff');
SELECT * FROM original_table ORDER BY col1 DESC limit 5;
```

This document is an aid for migrating an existing, natively partitioned table set to using pg_partman. For migrating a trigger-based partition set to native partitioning, see the document `migrate_to_native.md` for assistance with doing so. You can then return to this document to have that partition set managed by pg_partman.

As always, if you can first test this migration on a development system, it is highly recommended. The full data set is not needed to test this and just the

schema with a smaller set of data in each child should be sufficient enough to make sure it works properly.

The following are the example tables we will be using:

```
CREATE TABLE tracking.hits_time (id int GENERATED BY DEFAULT AS IDENTITY NOT NULL, start tim
CREATE INDEX ON tracking.hits_time (start);
CREATE TABLE tracking.hits_time2023_02_26 partition of tracking.hits_time FOR VALUES FROM ('
CREATE TABLE tracking.hits_time2023_03_05 partition of tracking.hits_time FOR VALUES FROM ('
CREATE TABLE tracking.hits_time2023_03_12 partition of tracking.hits_time FOR VALUES FROM ('

INSERT INTO tracking.hits_time VALUES (1, generate_series('2023-02-27 01:00:00'::timestamptz
INSERT INTO tracking.hits_time VALUES (2, generate_series('2023-03-06 01:00:00'::timestamptz
INSERT INTO tracking.hits_time VALUES (3, generate_series('2023-03-12 01:00:00'::timestamptz

\d+ tracking.hits_time
                                                    Partitioned table "tracking.hits_time"
 Column |           Type           | Collation | Nullable |             Default
--------+--------------------------+-----------+----------+---------------------------------
 id     | integer                  |           | not null | generated by default as identity
 start  | timestamp with time zone |           | not null | now()
Partition key: RANGE (start)
Indexes:
    "hits_time_start_idx" btree (start)
Partitions: tracking.hits_time20230226 FOR VALUES FROM ('2023-02-26 00:00:00-05') TO ('2023-
            tracking.hits_time20230305 FOR VALUES FROM ('2023-03-05 00:00:00-05') TO ('2023-
            tracking.hits_time20230312 FOR VALUES FROM ('2023-03-12 00:00:00-05') TO ('2023-

CREATE TABLE tracking.hits_id (id int GENERATED BY DEFAULT AS IDENTITY NOT NULL, start times
CREATE INDEX ON tracking.hits_id (id);
CREATE TABLE tracking.hits_id1000 partition of tracking.hits_id FOR VALUES FROM (1000) TO (2
CREATE TABLE tracking.hits_id2000 partition of tracking.hits_id FOR VALUES FROM (2000) TO (3
CREATE TABLE tracking.hits_id3000 partition of tracking.hits_id FOR VALUES FROM (3000) TO (4

INSERT INTO tracking.hits_id VALUES (generate_series(1000,1999), now());
INSERT INTO tracking.hits_id VALUES (generate_series(2000,2999), now());
INSERT INTO tracking.hits_id VALUES (generate_series(3000,3999), now());

\d+ tracking.hits_id
                                                     Partitioned table "tracking.hits_id"
 Column |           Type           | Collation | Nullable |             Default
--------+--------------------------+-----------+----------+---------------------------------
 id     | integer                  |           | not null | generated by default as identity
 start  | timestamp with time zone |           | not null | now()
Partition key: RANGE (id)
Indexes:
    "hits_id_id_idx" btree (id)
```

```
Partitions: tracking.hits_id1000 FOR VALUES FROM (1000) TO (2000),
            tracking.hits_id2000 FOR VALUES FROM (2000) TO (3000),
            tracking.hits_id3000 FOR VALUES FROM (3000) TO (4000)

CREATE TABLE tracking.hits_stufftime (id int GENERATED BY DEFAULT AS IDENTITY NOT NULL, sta
CREATE INDEX ON tracking.hits_stufftime (start);
CREATE TABLE tracking.hits_stufftimeaa partition of tracking.hits_stufftime FOR VALUES FROM
CREATE TABLE tracking.hits_stufftimebb partition of tracking.hits_stufftime FOR VALUES FROM
CREATE TABLE tracking.hits_stufftimecc partition of tracking.hits_stufftime FOR VALUES FROM

INSERT INTO tracking.hits_stufftime VALUES (1, generate_series('2023-01-02 01:00:00'::timest
INSERT INTO tracking.hits_stufftime VALUES (2, generate_series('2023-01-09 01:00:00'::timest
INSERT INTO tracking.hits_stufftime VALUES (3, generate_series('2023-01-15 01:00:00'::timest

\d+ tracking.hits_stufftime
                                                  Partitioned table "tracking.hits_stufftim
 Column |           Type           | Collation | Nullable |             Default
--------+--------------------------+-----------+----------+---------------------------------
 id     | integer                  |           | not null | generated by default as identity
 start  | timestamp with time zone |           | not null | now()
Partition key: RANGE (start)
Indexes:
    "hits_stufftime_start_idx" btree (start)
Partitions: tracking.hits_stufftimeaa FOR VALUES FROM ('2023-01-01 00:00:00-05') TO ('2023-0
            tracking.hits_stufftimebb FOR VALUES FROM ('2023-01-08 00:00:00-05') TO ('2023-0
            tracking.hits_stufftimecc FOR VALUES FROM ('2023-01-15 00:00:00-05') TO ('2023-0

CREATE TABLE tracking.hits_stuffid (id int GENERATED BY DEFAULT AS IDENTITY NOT NULL, start
CREATE INDEX ON tracking.hits_stuffid (id);
CREATE TABLE tracking.hits_stuffidaa partition of tracking.hits_stuffid FOR VALUES FROM (100
CREATE TABLE tracking.hits_stuffidbb partition of tracking.hits_stuffid FOR VALUES FROM (200
CREATE TABLE tracking.hits_stuffidcc partition of tracking.hits_stuffid FOR VALUES FROM (300

See below for data inserted

\d+ tracking.hits_stuffid
                                                   Partitioned table "tracking.hits_stuffid
 Column |           Type           | Collation | Nullable |             Default
--------+--------------------------+-----------+----------+---------------------------------
 id     | integer                  |           | not null | generated by default as identity
 start  | timestamp with time zone |           | not null | now()
Partition key: RANGE (id)
Indexes:
    "hits_stuffid_id_idx" btree (id)
Partitions: tracking.hits_stuffidaa FOR VALUES FROM (1000) TO (2000),
            tracking.hits_stuffidbb FOR VALUES FROM (2000) TO (3000),
            tracking.hits_stuffidcc FOR VALUES FROM (3000) TO (4000)
```

Step 1

Disable calls to the run_maintenance()

If you have any partitions currently maintained by pg_partman, you may be calling this already for them. They should be fine for the period of time this conversion is being done. This is to avoid any issues with only a partial configuration existing during conversion. If you are using the background worker, commenting out the "pg_partman_bgw.dbname" parameter in postgresql.conf and then reloading (SELECT pg_reload_conf();) should be sufficient to stop it from running. If you're running pg_partman on several databases in the cluster and you don't want to stop them all, you can also just remove the one you're doing the migration on from that same parameter.

Step 2

Stop all writes to the partition set being migrated if possible. If you cannot do this for the period of time the conversion will take, all of these following steps must be done in a single transaction to avoid write errors due table names changing.

Step 3

Rename the existing partitions to new naming convention. pg_partman uses a static pattern of suffixes for all partitions, both time & serial. All suffixes start with the string "_p" and are listed here for reference.

```
_pYYYYMMDD          - All time intervals greater than 1 day
_pYYYYMMDD_HH24MISS - All time intervals less than 1 day
_p#####             - Serial/ID partition has a suffix that is the value of the lowest possi
```

You can use custom datetime_string formats to change the suffix the children will get with pg_partman 5.0.1 and greater, but that is not covered in this tutorial. This only covers how to convert to using pg_partman's default suffixes and pg_partman always adds `_p` to the beginning to distinguish the suffix boundary.

Step 3a

For converting either time or serial based partition sets, if you have the lower boundary value as part of the partition name already, then it's simply a matter of doing a rename with some substring formatting since that's the pattern that pg_partman itself uses. Say your table was partitioned weekly and your original format just had the first day of the week (sunday) for the partition name (as in the example above). You can see below that we had 3 partitions with the old naming pattern of "YYYYMMDD" with no _p prefix. Looking at the list above, you can see the new weekly pattern that pg_partman uses.

So a query like the following which first extracts the original name then reformats the suffix would work. It doesn't actually do the renaming, it just generates

all the ALTER TABLE statements for you for all the child tables in the set. If all of them don't quite have the same pattern for some reason, you can easily just re-run this, editing things as needed, and filter the resulting list of ALTER TABLE statements accordingly. Note the `to_timestamp()` function is given the old datetime string pattern and the `to_char()` function is given the new string pattern.

```
SELECT format(
    'ALTER TABLE %I.%I RENAME TO %I;'
    , n.nspname
    , c.relname
    , substring(c.relname from 1 for 9) || '_p' || to_char(to_timestamp(substring(c.relname
)
        FROM pg_inherits h
        JOIN pg_class c ON h.inhrelid = c.oid
        JOIN pg_namespace n ON c.relnamespace = n.oid
        WHERE h.inhparent = 'tracking.hits_time'::regclass
        ORDER BY c.relname;
                                ?column?
-------------------------------------------------------------------------
 ALTER TABLE tracking.hits_time20230226 RENAME TO hits_time_p20230226;
 ALTER TABLE tracking.hits_time20230305 RENAME TO hits_time_p20230305;
 ALTER TABLE tracking.hits_time20230312 RENAME TO hits_time_p20230312;
```

Running that should rename your tables to look like this now:

```
 table_schema |     table_name
--------------+--------------------
 tracking     | hits_time_p20230226
 tracking     | hits_time_p20230305
 tracking     | hits_time_p20230312
```

If you're migrating a serial/id based partition set, and also have the naming convention with the lowest possible value, you'd do something very similar. Everything would be the same as the time-series one above except the renaming would be slightly different. Using my second example table above, it would be something like this.

```
SELECT format(
    'ALTER TABLE %I.%I RENAME TO %I;'
    , n.nspname
    , c.relname
    , substring(c.relname from 1 for 7) || '_p' || substring(c.relname from 8)
)
    FROM pg_inherits h
    JOIN pg_class c ON h.inhrelid = c.oid
    JOIN pg_namespace n ON c.relnamespace = n.oid
    WHERE h.inhparent = 'tracking.hits_id'::regclass
```

```
    ORDER by c.relname;
                          ?column?
-------------------------------------------------------------
 ALTER TABLE tracking.hits_id1000 RENAME TO hits_id_p1000;
 ALTER TABLE tracking.hits_id2000 RENAME TO hits_id_p2000;
 ALTER TABLE tracking.hits_id3000 RENAME TO hits_id_p3000;

 table_schema |  table_name
--------------+---------------
 tracking     | hits_id
 tracking     | hits_id_p1000
 tracking     | hits_id_p2000
 tracking     | hits_id_p3000
```

Step 3b

If your partitioned sets are named in a manner that relates differently to the data contained, or just doesn't relate at all, you'll instead have to do the renaming based off the lowest value in the control column instead. I'll be using the example above with the _aa, _bb, & _cc suffixes.

We'll be using the the `hits_stufftime` table in the first example here which has child tables that don't relate at all to the data contained.

This next step takes advantage of anonymous code blocks. It's basically writing pl/pgsql function code without creating an actual function. Just run this block of code, adjusting values as needed, right inside a psql session. Note that in PostgreSQL, weeks start on Monday's by default for the date_trunc function. However, say we wanted them to start on Sundays like they did for our other time partitioning example to keep things consistent. In that case we have to do a little extra date math to get that result.

```
DO $rename$
DECLARE
    v_min_val           timestamp;
    v_row               record;
    v_sql               text;
BEGIN

-- Adjust your parent table name in the for loop query
FOR v_row IN
    SELECT n.nspname AS child_schema, c.relname AS child_table
        FROM pg_inherits h
        JOIN pg_class c ON h.inhrelid = c.oid
        JOIN pg_namespace n ON c.relnamespace = n.oid
        WHERE h.inhparent = 'tracking.hits_stufftime'::regclass
        ORDER BY c.relname
LOOP
```

```
    v_min_val := NULL;
    -- Substitute your control column's name here in the min() function
    v_sql := format('SELECT min(start) FROM %I.%I', v_row.child_schema, v_row.child_table);
    EXECUTE v_sql INTO v_min_val;

    -- Adjust the date_trunc here to account for whatever your partitioning interval is.
    -- This is a trick to have the week begin on sunday since PG defaults to monday
    v_min_val := date_trunc('week', v_min_val + '1 day'::interval) - '1 day'::interval;

    -- Build the sql statement to rename the child table
    v_sql := format('ALTER TABLE %I.%I RENAME TO %I'
            , v_row.child_schema
            , v_row.child_table
            , substring(v_row.child_table from 1 for 14)||'_p'||to_char(v_min_val, 'YYYYMMDI

    -- I just have it outputting the ALTER statement for review. If you'd like this code to
    RAISE NOTICE '%', v_sql;
    -- EXECUTE v_sql;
END LOOP;

END
$rename$;
```

This will output something like this:

```
NOTICE:  ALTER TABLE tracking.hits_stufftimeaa RENAME TO hits_stufftime_p20230101
NOTICE:  ALTER TABLE tracking.hits_stufftimebb RENAME TO hits_stufftime_p20230108
NOTICE:  ALTER TABLE tracking.hits_stufftimecc RENAME TO hits_stufftime_p20230115
```

I'd recommend running it at least once with the final EXECUTE commented out to review what it generates. If it looks good, you can uncomment the EXECUTE and rename your tables!

If you've got a serial/id partition set, calculating the proper suffix value can be done by taking advantage of modulus arithmetic. Assume the following values in the tracking.hits_stuffid table:

```
INSERT INTO tracking.hits_stuffid VALUES (generate_series(1100,1294), now());
INSERT INTO tracking.hits_stuffid VALUES (generate_series(2400,2991), now());
INSERT INTO tracking.hits_stuffid VALUES (generate_series(3602,3843), now());
```

We'll be partitioning by 1000 again and you can see none of the minimum values are that even.

```
DO $rename$
DECLARE
    v_min_val           bigint;
    v_row               record;
    v_sql               text;
```

```
BEGIN

-- Adjust your parent table name in the for loop query
FOR v_row IN
    SELECT n.nspname AS child_schema, c.relname AS child_table
        FROM pg_inherits h
        JOIN pg_class c ON h.inhrelid = c.oid
        JOIN pg_namespace n ON c.relnamespace = n.oid
        WHERE h.inhparent = 'tracking.hits_stuffid'::regclass
        ORDER BY c.relname
LOOP
    -- Substitute your control column's name here in the min() function
    v_sql := format('SELECT min(id) FROM %I.%I', v_row.child_schema, v_row.child_table);
    EXECUTE v_sql INTO v_min_val;

    -- Adjust the numerical value after the % to account for whatever your partitioning inte
    v_min_val := v_min_val - (v_min_val % 1000);

    -- Build the sql statement to rename the child table
    v_sql := format('ALTER TABLE %I.%I RENAME TO %I'
            , v_row.child_schema
            , v_row.child_table
            , substring(v_row.child_table from 1 for 12)||'_p'||v_min_val::text);

    -- I just have it outputting the ALTER statement for review. If you'd like this code to
    RAISE NOTICE '%', v_sql;
    -- EXECUTE v_sql;
END LOOP;

END
$rename$;
```

You can see this makes nice even partition names:

```
NOTICE:  ALTER TABLE tracking.hits_stuffidaa RENAME TO hits_stuffid_p1000
NOTICE:  ALTER TABLE tracking.hits_stuffidbb RENAME TO hits_stuffid_p2000
NOTICE:  ALTER TABLE tracking.hits_stuffidcc RENAME TO hits_stuffid_p3000
```

Step 4

Setup pg_partman to manage your partition set.

I mentioned at the beginning that if you had ongoing writes, pretty much everything from Step 2 and on had to be done in a single transaction. Even if you don't have to worry about writes, I'd still highly recommend doing step 4 in the same transaction.

Note that we're doing weekly partitioning, but pg_partman only knows that

interval as a period of 7 days. So say we're migrating this parent table on March 31, 2023 with the child tables that we have already. If we tried to just call the create_parent() function without telling it when to start the partition set, it would assume our week is starting on a Friday, throwing off both the names of the child tables as well as the interval boundaries. We want to be sure our partition set is being configured to start the weekly partitions on a Sunday, so we use the `p_start_partition` parameter to tell it that (March 26th is the Sunday for the week of March 31st).

You may or may not need to set the starting partition parameter. It all depends on the interval you are using, so please test things to see if they work as expected before running on production.

```
SELECT partman.create_parent('tracking.hits_time', 'start', '1 week', p_start_partition := '
COMMIT;
```

This single function call will add your old partition set into pg_partman's configuration and possibly create some new child tables as well. pg_partman always keeps a minimum number of future partitions premade (based on the *premake* value in the config table or as a parameter to the create_parent() function), so if you don't have those yet, this step will take care of that as well. Adjust the parameters as needed and see the documentation for additional options that are available. This call matches the time partition used in the example so far.

```
 \d+ tracking.hits_time
                                                    Partitioned table "tracking.hits_time"
 Column |           Type           | Collation | Nullable |             Default
--------+--------------------------+-----------+----------+---------------------------------
 id     | integer                  |           | not null | generated by default as identity
 start  | timestamp with time zone |           | not null | now()
Partition key: RANGE (start)
Indexes:
    "hits_time_start_idx" btree (start)
Partitions: tracking.hits_time_p20230226 FOR VALUES FROM ('2023-02-26 00:00:00-05') TO ('202
            tracking.hits_time_p20230305 FOR VALUES FROM ('2023-03-05 00:00:00-05') TO ('202
            tracking.hits_time_p20230312 FOR VALUES FROM ('2023-03-12 00:00:00-05') TO ('202
            tracking.hits_time_p20230326 FOR VALUES FROM ('2023-03-26 00:00:00-04') TO ('202
            tracking.hits_time_p20230402 FOR VALUES FROM ('2023-04-02 00:00:00-04') TO ('202
            tracking.hits_time_p20230409 FOR VALUES FROM ('2023-04-09 00:00:00-04') TO ('202
            tracking.hits_time_p20230416 FOR VALUES FROM ('2023-04-16 00:00:00-04') TO ('202
            tracking.hits_time_p20230423 FOR VALUES FROM ('2023-04-23 00:00:00-04') TO ('202
            tracking.hits_time_default DEFAULT
```

By default, the premake value is 4, so it created 4 weeks in the future. And since this was the initial creation, it also creates 4 tables in the past. Even though we already had some tables there that covered those past 4 weeks, pg_partman saw they were there and just worked anyway. If we hadn't of passed

the p_start_partition parameter, PostgreSQL actually would have thrown an error since the past tables with different boundaries it would've tried to create would overlap the boundary of an already existing older table.

This final step is exactly the same no matter the partitioning type or interval, so once you reach here, run COMMIT and you're done! Schedule the run_maintenance() function to run (either via cron or the BGW) and future partition maintenance will be handled for you. Review the pg_partman.md documentation for additional configuration options.

I did the time examples here with a weekly partitioning set starting on a Sunday to show that there may be some caveats to your migration to pg_partman that need to be considered. If you're doing something simpler like daily or serial integer values, it may not be quite as complicated. Or if you're doing something more complex like a 9 week interval, there may be even more caveats to consider that may not seem obvious at first without extensive testing in development. So please plan your migration carefully.

If you have any issues with this migration document, please create an issue on Github.

This document will cover how to migrate a partition set using the old method of triggers/inheritance/constraints to a partition set using the native features found in PostgreSQL. This document assumes you are on at least PostgreSQL 14. This does not migrate the partition set fully to be used by pg_partman, it just provides the general guidance for the trigger->native process. Please See the separate document `migrate_to_partman.md` for migrating your partition sets to pg_partman.

For sub-partitioning you should be able to follow all the same processes here, but you will have to work from the lowest level upward and perform the migration on each sub-parent all the way to the top-level parent.

As always, if you can first test this migration on a development system, it is highly recommended. The full data set is not needed to test this and just the schema with a smaller set of data in each child should be sufficient enough to make sure it works properly.

This is how our partition set currently looks before migration:

```
\d+ partman_test.time_taptest_table
                                        Table "partman_test.time_taptest_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressio
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | integer                  |           | not null |         | plain    |
 col2   | text                     |           |          |         | extended |
 col3   | timestamp with time zone |           | not null | now()   | plain    |
Indexes:
    "time_taptest_table_pkey" PRIMARY KEY, btree (col1)
Triggers:
```

```
    time_taptest_table_part_trig BEFORE INSERT ON partman_test.time_taptest_table FOR EACH R
Child tables: partman_test.time_taptest_table_p2023_03_26,
              partman_test.time_taptest_table_p2023_03_27,
              partman_test.time_taptest_table_p2023_03_28,
              partman_test.time_taptest_table_p2023_03_29,
              partman_test.time_taptest_table_p2023_03_30,
              partman_test.time_taptest_table_p2023_03_31,
              partman_test.time_taptest_table_p2023_04_01,
              partman_test.time_taptest_table_p2023_04_02,
              partman_test.time_taptest_table_p2023_04_03,
              partman_test.time_taptest_table_p2023_04_04,
              partman_test.time_taptest_table_p2023_04_05,
              partman_test.time_taptest_table_p2023_04_06,
              partman_test.time_taptest_table_p2023_04_07,
              partman_test.time_taptest_table_p2023_04_08,
              partman_test.time_taptest_table_p2023_04_09
Access method: heap
```

If your trigger-based partition set happens to be managed by pg_partman version prior to 5.0.0, it is best to remove it from partman management. This can be done by deleting it from the `part_config` and `part_config_sub` tables (if sub-partitioned, ensure all child tables are removed as well). After it has been migrated to native partitioning, see the Migrating to pg_partman document mentioned above to return it to being managed by partman.

```
DELETE FROM partman.part_config WHERE parent_table = 'partman_test.time_taptest_table';
```

If it is not managed by pg_partman and you have some other method of automated maintenance, ensure that process is disabled.

Next, we need to create a new parent table using native partitioning since you cannot currently convert an existing table into a native partition parent. Note in this case our original table had a primary key on `col1`. Since `col1` is not part of the partition key, native partitioning does not allow us to declare it as a primary key on the top level table. If you still need this as a primary key, pg_partman provides a template table you can set this on, but it will still not enforce uniqueness across the entire partition set, only on a per-child basis similar to how it worked before native.

Please see the `Child Table Property Inheritance` section of `docs/pg_partman.md` for which properties can be set on the native parent and which must be managed via the template .

```
CREATE TABLE partman_test.time_taptest_table_native
    (col1 int, col2 text default 'stuff', col3 timestamptz NOT NULL DEFAULT now())
    PARTITION BY RANGE (col3);

CREATE INDEX ON partman_test.time_taptest_table_native (col3);
```

Next check what the ownership and privileges on your original table were and ensure they exist on the new parent table. This will ensure all access to the table works the same after the migration. By default with native partitioning, privileges are no longer granted on child tables to provide direct access to them. If you'd like to keep that behavior, set the `inherit_privileges` column in `part_config` (and part_config_sub if needed) to true.

```
\dt partman_test.time_taptest_table
                  List of relations
    Schema    |        Name         | Type  |     Owner
--------------+---------------------+-------+---------------
 partman_test | time_taptest_table | table | partman_owner
(1 row)

\dp+ partman_test.time_taptest_table
                                          Access privileges
    Schema    |        Name         | Type  |           Access privileges           | Column pr
--------------+---------------------+-------+---------------------------------------+---------
 partman_test | time_taptest_table | table | partman_owner=arwdDxt/partman_owner+|
              |                     |       | partman_basic=arwd/partman_owner   +|
              |                     |       | testing=r/partman_owner             |
(1 row)

ALTER TABLE partman_test.time_taptest_table_native OWNER TO partman_owner;
GRANT SELECT, INSERT, UPDATE, DELETE ON partman_test.time_taptest_table_native TO partman_ba
GRANT SELECT ON partman_test.time_taptest_table_native TO testing;
```

It is best to halt all activity on the original table during the migration process to avoid any issues. This can be done by either revoking all permissions to the table temporarily or by taking out an exclusive lock on the parent table and running all of these steps in a single transaction. The transactional method is highly recommended for the simple fact that if you run into any issues before you've completed the migration process, you can simply rollback and return to the state your database was in before the migration started.

```
BEGIN;
LOCK TABLE partman_test.time_taptest_table IN ACCESS EXCLUSIVE MODE NOWAIT;
```

If this is a subpartitioned table, the lock on the top-level parent should lock out access on all child tables as long as you don't use the ONLY clause.

The first major step in this migration process is now to uninherit all the child tables from the old parent. You can use a query like the one below to generate the ALTER TABLE statements to uninherit all child tables from the given parent table. It's best to use generated SQL like this to avoid typos, especially with very large partition sets:

DO NOT RUN THE RESULTING STATEMENTS YET. A future query will not work if the child tables are no longer part of the inheritance set.

```
SELECT format('ALTER TABLE %s NO INHERIT %s;', inhrelid::regclass, inhparent::regclass)
FROM pg_inherits
WHERE inhparent::regclass = 'partman_test.time_taptest_table'::regclass;
                                                    ?column?
-------------------------------------------------------------------------------------------------
 ALTER TABLE partman_test.time_taptest_table_p2023_03_26 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_03_27 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_03_28 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_03_29 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_03_30 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_03_31 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_01 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_02 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_03 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_04 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_05 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_06 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_07 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_08 NO INHERIT partman_test.time_taptes
 ALTER TABLE partman_test.time_taptest_table_p2023_04_09 NO INHERIT partman_test.time_taptes
(15 rows)
```

Again, DO NOT RUN THESE STATEMENTS YET. The following query will not work if the child tables are no longer part of the inheritance set.

For any partition sets, even those not managed by pg_partman, the next step is that you need to figure out the boundary values of your existing child tables and feed those to the ATTACH PARTITION command used in native partitioning. You will have to figure out a method to determine the boundaries of your child tables to be able to convert them to the syntax needed for PostgreSQL's declarative commands. In our case here, our child table's suffixes have a fixed naming pattern (`YYYY_MM_DD`) that can be parsed to determine the starting boundary value. In this case we'll also need to know the partitioning interval (`1 day`) to be able to calculate the boundary end time.

If your child table names do not have a usable pattern like this, you'll have to figure out some method of determining each child table's boundaries.

Again, we can use some sql to generate statements to re-attach the children to the new parent:

```
WITH child_tables AS (
    SELECT
          inhrelid::regclass::text AS child_tablename_safe
        , relname AS child_tablename  -- need unquoted name for parsing
    FROM pg_inherits
    JOIN pg_class c ON inhrelid = c.oid
    WHERE inhparent = 'partman_test.time_taptest_table'::regclass
```

```
)
SELECT format(
    'ALTER TABLE %s ATTACH PARTITION %s FOR VALUES FROM (%L) TO (%L);'
    , 'partman_test.time_taptest_table_native'::regclass
    , child_tablename_safe
    , to_timestamp(right(x.child_tablename, 10), 'YYYY_MM_DD')
    , to_timestamp(right(x.child_tablename, 10), 'YYYY_MM_DD')+'1 day'::interval
)
FROM child_tables x;
                                                                                         ?co
-----------------------------------------------------------------------------------------------
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
 ALTER TABLE partman_test.time_taptest_table_native ATTACH PARTITION partman_test.time_tapte
(15 rows)
```

We can now run these two sets of ALTER TABLE statements to first uninherit them from the old trigger-based parent and attach them to the new native parent. After doing so, the old trigger-based parent should have no longer have children:

```
\d+ partman_test.time_taptest_table
```

```
                                          Table "partman_test.time_taptest_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Compressio
--------+--------------------------+-----------+----------+---------+----------+-----------
 col1   | integer                  |           | not null |         | plain    |
 col2   | text                     |           |          |         | extended |
 col3   | timestamp with time zone |           | not null | now()   | plain    |
Indexes:
    "time_taptest_table_pkey" PRIMARY KEY, btree (col1)
Triggers:
    time_taptest_table_part_trig BEFORE INSERT ON partman_test.time_taptest_table FOR EACH R
Access method: heap
```

And our new native parent should have now adopted all its new children:

```
\d+ partman_test.time_taptest_table_native
                                    Partitioned table "partman_test.time_taptest_table_native
 Column |           Type           | Collation | Nullable |    Default    | Storage  | Compr
--------+--------------------------+-----------+----------+---------------+----------+------
 col1   | integer                  |           |          |               | plain    |
 col2   | text                     |           |          | 'stuff'::text | extended |
 col3   | timestamp with time zone |           | not null | now()         | plain    |
Partition key: RANGE (col3)
Indexes:
    "time_taptest_table_native_col3_idx" btree (col3)
Partitions: partman_test.time_taptest_table_p2023_03_26 FOR VALUES FROM ('2023-03-26 00:00:0
            partman_test.time_taptest_table_p2023_03_27 FOR VALUES FROM ('2023-03-27 00:00:0
            partman_test.time_taptest_table_p2023_03_28 FOR VALUES FROM ('2023-03-28 00:00:0
            partman_test.time_taptest_table_p2023_03_29 FOR VALUES FROM ('2023-03-29 00:00:0
            partman_test.time_taptest_table_p2023_03_30 FOR VALUES FROM ('2023-03-30 00:00:0
            partman_test.time_taptest_table_p2023_03_31 FOR VALUES FROM ('2023-03-31 00:00:0
            partman_test.time_taptest_table_p2023_04_01 FOR VALUES FROM ('2023-04-01 00:00:0
            partman_test.time_taptest_table_p2023_04_02 FOR VALUES FROM ('2023-04-02 00:00:0
            partman_test.time_taptest_table_p2023_04_03 FOR VALUES FROM ('2023-04-03 00:00:0
            partman_test.time_taptest_table_p2023_04_04 FOR VALUES FROM ('2023-04-04 00:00:0
            partman_test.time_taptest_table_p2023_04_05 FOR VALUES FROM ('2023-04-05 00:00:0
            partman_test.time_taptest_table_p2023_04_06 FOR VALUES FROM ('2023-04-06 00:00:0
            partman_test.time_taptest_table_p2023_04_07 FOR VALUES FROM ('2023-04-07 00:00:0
            partman_test.time_taptest_table_p2023_04_08 FOR VALUES FROM ('2023-04-08 00:00:0
            partman_test.time_taptest_table_p2023_04_09 FOR VALUES FROM ('2023-04-09 00:00:0
```

Next is to swap the names of your old trigger-based parent and the new native parent.

```
ALTER TABLE partman_test.time_taptest_table RENAME TO time_taptest_table_old;
ALTER TABLE partman_test.time_taptest_table_native RENAME TO time_taptest_table;
```

PG11+ supports the feature of a default partition to catch any data that doesn't have a matching child. If your table names are particularly long, ensure that adding the `_default` suffix doesn't get truncated unexpectedly. The suffix isn't required for functionality, but provides good context for what the table is for, so it's better to shorten the table name itself to fit the suffix.

```
CREATE TABLE partman_test.time_taptest_table_default (LIKE partman_test.time_taptest_table I
ALTER TABLE partman_test.time_taptest_table ATTACH PARTITION partman_test.time_taptest_table
```

There was a primary key on the original parent table, but that is not possible with native partitioning unless the primary key also includes the partition key. This is typically not practical in time-based partitioning. You can place a primary key on each individual child table, but that only enforces the constraint for that child table, not across the entire partition set. You can add a primary key to each individual table using similar SQL generation above, but if you'd like a method to manage adding these to any new child tables, please see the features

available in pg_partman.

If you've run this process inside a transaction, be sure to commit your work now:

```
COMMIT;
```

This should complete the migration process. If you'd like to migrate your partition set to be managed by pg_partman, please see the `migrate_to_partman.md` documentation file.

If pg_partman was installed on your database instance before it was upgraded to at least PostgreSQL 11, it will likely be missing some or all of the new PROCEDUREs that were added over time. It may have some of them if you have since updated pg_partman to a more recent version, but that only installed PROCEDUREs that happened to be part of that update. There could still be some missing.

The best way to fix this and ensure all PROCEDUREs have been installed is to drop and recreate the extension once you are on PG11 or greater. It is recommended that you test the steps below in development/testing before running on any production systems so you are sure the outcome works as expected.

IMPORTANT NOTES: 1. If you installed pg_partman originally on PG11 or later, you DO NOT need to do any steps in this guide. 2. Since the entire extension is being dropped and recreated, you will lose any grants that had been given on any specific extension objects and default privileges that were revoked may be restored. Please make note of the users that were managing partition maintenance before and ensure they have their grants restored. 3. If you are still using trigger-based partitioning, you will have to take an outage for all trigger-based tables since objects that the triggers use will be dropped and restored. It is highly recommended to migrate away from trigger-based partitioning if possible. This is both for performance reasons as well as future-proofing since trigger-based partitioning is removed as of version 5.0.0. 4. The same version of pg_partman that is dropped **MUST** be reinstalled to restore the configuration. It is recommended that you install the latest version available before starting this update.

Update Steps

1. Perform a pg_dump of the data from the pg_partman configuration tables. Note that the contents of this dump will only contain the data and not the table definitions. The definitions are part of the CREATE EXTENSION step. This is just doing a plaintext dump to make it easier to review the contents if desired. Note the following command assumes pg_partman was installed in the `partman` schema.

```
pg_dump -d mydbname -Fp -a -f partman_update_procedures.sql -t partman.part_config -t partma
```

2. Drop the pg_partman extension. If it was installed in a specific schema make note of this and reinstall it to that same schema

```
\dx pg_partman
                              List of installed extensions
    Name    | Version | Schema  |                      Description
------------+---------+---------+-----------------------------------------------------------
 pg_partman | 4.7.0   | partman | Extension to manage partitioned tables by time or ID
```

```
DROP EXTENSION pg_partman;
```

3. Reinstall pg_partman to the same schema

```
CREATE EXTENSION pg_partman SCHEMA partman;
```

4. Reload the data back into the extension configuration tables

```
psql -d mydbname -i partman_update_procedures.sql
```

5. Restore privileges to pg_partman objects if needed

You should now have any missing PROCEDUREs available to use as well as your original pg_partman configuration.