Migrating An Existing Partition Set to PG Partition Manager

PG Partition Manager

pg_partman is an extension to create and manage both time-based and serial-based table partition sets. Native partitioning in PostgreSQL 10 is supported as of pg_partman v3.0.1 and much more extensively as of 4.0.0 along with PostgreSQL 11. Note that all the features of trigger-based partitioning are not yet supported in native, but performance in both reads & writes is significantly better.

Child table creation is all managed by the extension itself. For non-native, trigger function maintenance is also handled. For non-native partitioning, tables with existing data can have their data partitioned in easily managed smaller batches. For native partitioning, the creation of a new partitioned parent must be done first and the data migrated over after setup is complete.

Optional retention policy can automatically drop partitions no longer needed for both native and non-native partitioning.

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

If you're looking for a partitioning system that handles any range type beyond just time & serial, the new native partitioning features in PostgreSQL 10+ are likely the best method for the foreseeable future. If this is something critical to your environment, start planning your upgrades now!

If you're still trying to evaluate whether partitioning is a good choice for your environment, keep an eye on the HypoPG project. Version 2 will have a hypothetical partitioning feature that will let you evaluate different partitioning schemes without requiring you to actually partition your data. I may see about integrating this feature into pg_partman once it is available. - https://hypopg.readthedocs.io

INSTALLATION

Requirement:

- PostgreSQL >= 10

Recommended:

- Native partitioning is highly recommended over trigger-based and PG11+ is HIGHLY recommended over PG10.
- pg_jobmon (>=v1.4.0). PG Job Monitor will automatically be used if it is installed and setup properly. https://github.com/omniti-labs/pg_jobmon

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
```

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;
```

As of version 4.1.0, pg_partman no longer requires a superuser to run for native partitioning. Trigger-based partitioning still requires it, so if you want to not require superuser, look to migrating to native partitioning. Superuser is still required to install pg_partman. 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”):

```
CREATE ROLE partman WITH LOGIN;
GRANT ALL ON SCHEMA partman TO partman;
GRANT ALL ON ALL TABLES IN SCHEMA partman TO partman;
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA partman TO partman;
GRANT EXECUTE ON ALL PROCEDURES IN SCHEMA partman TO partman;  -- PG11+ only
GRANT ALL ON SCHEMA my_partition_schema TO partman;
GRANT TEMPORARY ON DATABASE mydb to partman; -- allow creation of temp tables to move data
```

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;
```

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 to put the 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/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 (2.x -> 3.x, etc), please carefully read all intervening version notes in the CHANGELOG, especially those notes for the major version. There are often additional instructions (Ex. updating trigger functions) and other important considerations for the updates.

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 native partitioning with pg_partman on a brand new table, or to migrate an existing normal table to native partitioning, see pg_partman_howto_native.md.

For migrating a trigger-based partitioned table to native partitioning using pg_partman, see migrate_to_native.md.

Other HowTo documents are also available in the documents folder.

See the pg_partman command guide 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 evalutate if it is working properly (http://www.pgtap.org). WARNING: You MUST increase max_locks_per_transaction above the default value of 64. For me, 128 has worked well so far. This is due to the sub-partitioning 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.

About

PostgreSQL Partition Manager is an extension to help make managing time or serial 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). Currently the trigger functions only handle inserts to the parent table. Updates that would move a value from one partition to another are only supported in PostgreSQL 11 native partitioning. Some features of this extension have been expanded upon in the author's blog - http://www.keithf4.com/tag/pg_partman

As of version 3.0.1, this extension will support the native partitioning methods that were introduced in PostgreSQL 10. A trigger function is no longer necessary in native partitioning, but automatic child table creation is not handled natively, which is where this extension comes into play. Version 4.0.0 adds even more native support for features introduced in PG11 (easier index/fk inheritance, default partition).

For non-native partitioning, if you attempt to insert data into a partition set that contains data for a partition that does not exist, that data will be placed into the set's parent table. This is preferred over automatically creating new partitions to match that data since a mistake that is causing non-partitioned data to be inserted could cause a lot of unwanted child tables to be made as well as contention due to transactional DDL. The `check_default()` function provides monitoring for any data getting inserted into the parent/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 having to clean up potentially hundreds or thousands of unwanted partitions. And also better than throwing an error and losing the data! For native partitioning, inserting data with no relevant child causes an error in PostgreSQL 10. A default partition for native is only available in PostgreSQL 11+.

Note that future child table creation is based on the data currently in the partition set. This means that if you put "future" data in, newly created tables will be based off that value. This may cause intervening data to go to the parent/default as stated above if no child table exists. It is recommended that you set the `premake` value high enough to encompass your expected data range being inserted. And for non-native partitioning, set the `optimize_trigger` value to efficiently handle your most frequent data range. 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 migration.md file in the doc folder. This is for non-native partitioning only at this time. I'm working on a migration plan for getting non-native partition sets moved to native partition sets. If it all works out, will be included in a future version of pg_partman.

Child Table Property Inheritance

For this extension, most of the attributes of the child partitions are all obtained from the parent table. For non-native, trigger-based partititioning, all properties are managed via the parent and always will be. However, with native 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 for native partitioning. If a property is not listed here, then assume it is managed via the parent. Note that if you will be upgrading your major version, you will have to change how the properties are managed appropriately if something has moved from being managed by the template to being managed by the real parent (Ex. foreign keys going from 10 to 11+). The WITH OIDS property is no longer officially supported by pg_partman at all (native or non-native) as of the release of PostgreSQL 12 since it was dropped there.

Feature	Parent Inheritance	Template Inheritance
non-partition column primary key		All
non-partition column unique index		All
non-partition column unique index tablespace		All

Feature	Parent Inheritance	Template Inheritance
unlogged table state*		All
non-unique indexes	11, 12	10
foreign keys	11, 12	10
tablespaces	12	10, 11
privileges/ownership	All	
constraints	All	
defaults	All	
publications	14	10,11,12,13

Privileges & ownership are inherited by default for non-native partitioning, but NOT for native partitioning. Also 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 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.

The new IDENTITY feature introduced in PG10 is only supported in natively partitioned tables and 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 in use for PostgreSQL 10+ to handle certain features is only a temporary solution to help speed up native partitioning adoption. As things are handled better natively (as they were in PG11), the use of the template table will be phased out quickly from pg_partman. So please plan ahead for quick major version upgrades if you use this feature and check release notes carefully for changes.
- The UNLOGGED status was moved to the template table as of v4.2.0 of pg_partman. This is due to an inconsistency in the way the property is handled when either enabling or disabling UNLOGGED on the parent table of a native partition set. That property does not actually change 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 PostgreSQL clients work in general.

It is highly recommended to run your database system in UTC time to overcome issues that are currently not possible to solve due to Daylight Saving time changes. Then also ensure the client that will be creating partition sets and running the maintenance calls is also set to UTC.

There is currently an open issue for anyone looking to help try and solve some of these DST issues - https://github.com/pgpartman/pg_partman/issues/334

Sub-partitioning

Sub-partitioning 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 sub-partitioning. 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. If you're on PG11+, you can use the new run_maintenance_proc() procedure to cause less contention issues since it automatically commits after each partition set's maintenance.

PUBLICATION/SUBSCRIPTION for logical replication is NOT supported with native sub-partitioning.

See the `create_parent_sub()` & `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 may 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. As of version 4.2+ of pg_partman, there is now 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 sub-partitioning. It will work on the first level of partitioning, but is not guaranteed to work properly on further sub-partition 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 sub-partition ones likely will not.

Custom Time Interval Considerations

The list of time intervals given for create_parent() below are optimized to work as fast as possible with non-native, trigger-based partitioning. Intervals other than those values are possible, but performance will take a non-trivial hit to allow such flexibility. For native partitioning, unlike pg_partman's trigger based method, there's no differing method of partitioning for any given intervals. All possible intervals that use the native method have the same performance characteristics and are better than any trigger-based method. If you are still using trigger-based partitioning and you need a different partiton interval than the ones pg_partman provides, it is HIGHLY recommended to upgrade to the latest version of PostgreSQL and migrate to native partitioning.

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). The smallest integer interval supported at this time is 10.

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".

Naming Length Limits

PostgreSQL has an object naming length limit of 63 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 tables, trigger functions and triggers. 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 serial 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 & Upsert

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.

IMPORTANT NOTE: Upsert is no longer supported in pg_partman for native partitioning as of version 4.6.0 and PostgreSQL 11+. Please use the INSERT…ON CONFLICT feature built into PostgreSQL.

For non-native partitioning and PG10 native partitioning, INSERT … ON CONFLICT (upsert) is supported in the partitioning trigger as well as native partitioning, but is very limited. The major limitations are that the constraint violations that would trigger the ON CONFLICT clause only occur on individual child tables that actually contain data due to reasons explained above. Of a larger concern than data duplication is an ON CONFLICT DO UPDATE clause which may not fire and cause wildly inconsistent data if not accounted for. For situations where only new data is being inserted, upsert can provide significant performance improvements. However, if you're relying on data in older partitions to cause a constraint violation that upsert would normally handle, it likely will not work. Also, if the resulting UPDATE would end up violating the partitioning constraint of that chld table, it will fail. Neither pg_partman & PG10 native partitioning currently support UPDATES that would require moving a row from one child table to another. This is only supported in PG11+.

pg_partman's upsert feature is optional, turned off by default and was only included since there was no native support on the core PostgreSQL roadmap at the time it was implemented. At this time, if you have not implemented this feature, it is highly recommended you upgrade to PG11.

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

With PostgreSQL 9.4, the ability to create custom background workers and dynamically load them during runtime was introduced. `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 (default for PG10 and older). Set to 'off' for FALSE (Default for PG11+).
- `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.

As of version 4.0.0, the background worker still uses the normal run_maintenance() function. An option to use the new procedure is in the works.

Extension Objects

As of 4.4.0, SECURITY DEFINER has been removed from all functions in pg_partman. Requiring a superuser to use pg_partman is now completely optional for native partitioning. To run as nonsuperuser, 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 to that argument. As an example: `SELECT create_parent('schema.table', 'col1', 'partman', 'daily', p_start_partition := '2015-10-20');`

Creation Functions

create_parent(p_parent_table text, p_control text, p_type text, p_interval text, p_constraint_cols text[] DEFAULT NULL, p_premake int DEFAULT 4, p_automatic_maintenance text DEFAULT 'on', p_start_partition text DEFAULT NULL, p_inherit_fk boolean DEFAULT true, p_epoch text DEFAULT 'none', p_upsert text DEFAULT '', p_publications text[] DEFAULT NULL, p_trigger_return_null boolean DEFAULT true, 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. Please apply all defaults, indexes, constraints, privileges & ownership to parent table so they will propagate to children. For native partitioning, the parent table must already be declared as such and config options passed to this function must match that definition. See notes above about handling indexes & foreign keys.
- 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.
- For PG11+, a default partition is automatically created. A "_default" suffix is added onto the current table name.
- 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_type - one of the following values to set the partitioning type that will be used:
  - **native**
    - Use the native partitioning methods that are built into PostgreSQL 10+.
    - For PG11+, it is highly recommended that native partitioning be used over trigger-based partitioning. PG10 is still lacking significant features for native partitioning, so please see notes above for more info.
    - Provides significantly better write & read performance than "partman" partitioning.
    - Child table creation is kept up to date by running `run_maintenance(_proc)`. There is no trigger maintenance.
  - **partman**
    - Create a trigger-based partition set using pg_partman's method of partitioning.
    - Whether it is time or serial based is determined by the control column's data type and if the p_epoch flag is set.
    - The number of partitions most efficiently managed behind and ahead of the current one is determined by the **optimize_trigger**

config value in the part_config table (default of 4 means data for 4 previous and 4 future partitions are handled best).
    - *Beware setting the optimize_trigger value too high as that will lessen the efficiency boost.*
    - Inserts to the parent table outside the optimize_trigger window will go to the proper child table if it exists, but performance will be degraded due to the higher overhead of handling that condition.
    - If the child table does not exist for the value given, the row will go to the parent.
    - Child table creation & trigger function is kept up to date by the `run_maintenance()` function.

- `p_interval` - the time or integer range interval for each partition. No matter the partitioning type, value must be given as text. The generic intervals of "yearly -> quarter-hour" are for time partitioning and giving one of these explicit values when using pg_partman's trigger-based partitioning will allow significantly better performance than using an arbitrary time interval. For native partitioning, any interval value is valid and will have the same performance which is always better than trigger-based.
  - *yearly* - One partition per year
  - *quarterly* - One partition per yearly quarter. Partitions are named as YYYYqQ (ex: 2012q4)
  - *monthly* - One partition per month
  - *weekly* - One partition per week. Follows ISO week date format (http://en.wikipedia.org/wiki/ISO_week_date). Partitions are named as IYYYwIW (ex: 2012w36)
  - *daily* - One partition per day
  - *hourly* - One partition per hour
  - *half-hour* - One partition per 30 minute interval on the half-hour (1200, 1230)
  - *quarter-hour* - One partition per 15 minute interval on the quarter-hour (1200, 1215, 1230, 1245)
  - *<interval>* - Any other interval besides the values above that is valid for the PostgreSQL interval type. Note this will have a significant performance penalty if not using native partitioning. 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). Must be greater than or equal to 10.
- `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_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, daylight savings (on non-UTC systems), etc. This won't hurt anything and will self-correct. If partitioning ever falls behind the `premake` value, normal running of `run_maintenance()` and data insertion should automatically catch things up.

- `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_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 partitions 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 sub-partitioning, this only applies during initial setup and not during ongoing maintenance.
- `p_inherit_fk` - allows `pg_partman` to automatically manage inheriting any foreign keys that exist on the parent (or template for native) table to all its children. Defaults to TRUE. Note this option is only relevant for PostgreSQL 10 and older. PG11+ automatically inherits any foreign keys placed on the parent and is not optional.
- `p_epoch` - tells `pg_partman` that the control column is an integer type, but actually represents and epoch time value. You can also specify whether the value is seconds, milliseconds or nanoseconds. Valid values for this option are: 'seconds', 'milliseconds', 'nanoseconds', and 'none'. The default is 'none'. All triggers, constraints & 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_upsert` - adds upsert to insert queries in the partition trigger to allow handling of conflicts Defaults to " (empty string) which means it's inactive.
  - IMPORTANT NOTE: This feature was deprecated in v4.6.0. Please plan on migrating to PG11, native partitioning and INSERT… ON CONFLICT if you are using this feature.
  - the value entered here is the entire ON CONFLICT clause which will then be appended to the INSERT statement(s) in the trigger
  - Ex: to ignore conflicting rows on a table with primary key "id" set p_upsert to `'ON CONFLICT (id) DO NOTHING'`

    - Ex: to update a conflicting row on a table with columns (id(pk), val) set p_upsert to `'ON CONFLICT (id) DO UPDATE SET val=EXCLUDED.val'`
    - Requires postgresql 9.5
    - See *About* section above for more info.
- `p_publications` - Option to add child tables to publications for use with logical replication. Value is an array list of publication names, so multiple publications can be added to each child. Note that if you are replicating to a partition set on the subscriber side, you will have to set the `subscription_refresh` option in the `part_config` table on the subscriber side to pick up new tables from the source publication. Currently does not support sub-partitioning for native partition sets since a publication cannot be added to the parent of a natively partitioned table.
- `p_trigger_return_null` - Only applies to non-native, trigger-based partitioning. Boolean value that allows controlling the behavior of the partition trigger RETURN. By default this is true and the trigger returns NULL to prevent data going into the parent table as well as the children. However, if you have multiple triggers and are relying on the return to be the NEW column value, this can cause a problem. Setting this config value to false will cause the partition trigger to RETURN NEW. You are then responsible for handling the return value in another trigger appropriately. Otherwise, this will cause new data to go into both the child and parent table of the partition set.
- `p_template_table` - For native partitioning in PG10, indexes, foreign keys & tablespaces cannot be set on the parent table. For PG11, only unique indexes that don't include the partition key cannot be created on the parent. Therefore, if you want them to be automatically created on child tables, they must be managed elsewhere. 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. Note that until indexes, foreign keys or tablespaces are made on the template, no child tables will have any. Use the python scripts to reapply the indexes and foreign keys to the partition set when the template table is ready. For tablespaces, you will have to manually move any previously existing child tables. If you pre-create a template table and pass its name here, then the initial child tables will obtain these properties 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_control text, p_type text, p_interval text, p_native_check text DEFAULT NULL, p_constraint_cols text[] DEFAULT NULL, p_premake int DEFAULT 4, p_start_partition text DEFAULT NULL, p_inherit_fk boolean DEFAULT true, p_epoch text DEFAULT 'none', p_upsert text DEFAULT '', p_trigger_return_null boolean DEFAULT true, 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 above.
- `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_native_check` - Turning an existing native partition set into a subpartitioned set is a **destructive** process. A table must be declared natively 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 knowledge 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 sub-partitioning a native partition set. This option can be ignored if you are created a trigger-based pg_partman partition set.
- 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, but if the total length is longer than 63 chars, the original name will get truncated. Longer table names may cause the original parent table names to be truncated and possibly cut off the top level partitioning suffix. I cannot control this and made the requirement that 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).
- For native partitioning, 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 accidentally gets inserted into the parent table (trigger-based only) or default table (native, PG11+ only).
- 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 either use the `partition_data.py` script to commit data in smaller batches. Or if you're on PG11+, use the `partition_data_proc()` procedure to do the same thing. This will greatly reduce issues caused by long running transactions and data contention.
- For sub-partitioned 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: this may not work as expected for native partitioning since subpartitioning a native set in pg_partman is a destructive operation. See create_sub_parent().
- `p_parent_table` - the existing parent table. For non-native partitioning, this is assumed to be where the unpartitioned data is located. MUST be schema qualified, even if in public schema.
- `p_batch_interval` - optional argument, only relevant for non-native partitioning. A time interval of how much of the data to move. This can be smaller than the partition interval, allowing for very large sized 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: This option CANNOT be used when moving data out of a default partition in PostgreSQL 11+.
- `p_batch_count` - optional argument, how many times to run the `batch_interval` in a single call of this function. Default value is 1. For native partitioning, this sets how many child tables will be processed in a single run.
- `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 natively partitioned set. 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/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.
- Returns the number of rows that were moved from the parent table to partitions. Returns zero when source/parent 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 serial id partition set. It also fixes data that accidentally gets inserted into the parent (trigger-based only).
- 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 either use the `partition_data.py` script to commit data in smaller batches. Or if you're on PG11+, use the `partition_data_proc()` procedure to do the same thing. This will greatly reduce issues caused by long running transactions and data contention.
- For sub-partitioned 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: this may not work as expected for native partitioning since subpartitioning a native set in

pg_partman is a destructive operation. See create_sub_parent().

- `p_parent_table` - the existing parent table. For non-native partitioning, this is assumed to be where the unpartitioned data is located. MUST be schema qualified, even if in public schema.
- `p_batch_interval` - optional argument, only relevant for non-native partitioning. A time interval of how much of the data to move. This can be smaller than the partition interval, allowing for very large sized 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: This option CANNOT be used when moving data out of a default partition in PostgreSQL 11+.
- `p_batch_count` - optional argument, how many times to run the `batch_interval` in a single call of this function. Default value is 1. For native partitioning, this sets how many child tables will be processed in a single run.
- `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 natively partitioned set. 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/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.
- Returns the number of rows that were moved from the parent table to partitions. Returns zero when source/parent table is empty and partitioning is complete.

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

- A procedure that can partition data in distinct commit batches to avoid long running transactions and data contention issues.

- Only works with PostgreSQL 11+
- 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_interval` - Value that is passed on to the partitioning 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. NOTE: This option CANNOT be used when moving data out of a default partition in PostgreSQL 11+.
- `p_batch` - 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 -b partition(s). Procedure commits at the end of each individual batch. (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_wait` - Cause the procedure to pause for a given number of seconds between commits (batches) to reduce write load
- `p_source_table` - Same as the p_source_table option in the called partitioning function.
- `p_order` - Allows you to specify the order that data is migrated from the parent/default to the children, either ascending (ASC) or descending (DESC). Default is ASC.
- `p_lock_wait` - Parameter 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_quiet` - Procedures cannot return values, so by default it emits NOTICE's to show progress. Set this option to silence these notices.
- `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.

`create_partition_time(p_parent_table text, p_partition_times timestamptz[], p_analyze boolean DEFAULT true, 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.
- For non-native partitioning, you may also need to call create_function_time()

to update the partitioning trigger if you created partitions in the "current" optimization window.
- `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, it will not be created 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_analyze` - If a new child table is created, an analyze is normally kicked off so that the statistics are aware of the constraint boundaries for constraint exclusion. For larger partition sets, this analyze can take a long time. Set this to false to skip this automatic analyze.
- `p_start_partition` - When using sub-partitioning, allows passing along the start partition value for the sub-partition 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_analyze boolean DEFAULT true, 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.
- For non-native partitioning, you may need to also call create_function_id() to update the partitioning trigger if you created partitions in the "current" optimization window.
- `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, it will not be created 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_analyze` - If a new child table is created, an analyze is normally kicked off so that the statistics are aware of the constraint boundaries for constraint exclusion. For larger partition sets, this analyze can take a long time. Set this to false to skip this automatic analyze.
- `p_start_partition` - When using sub-partitioning, allows passing along the start partition value for the sub-partition child tables.
- Returns TRUE if any child tables were created for the given integer values. Returns false if no child tables were created.

create_function_time(p_parent_table text, p_job_id bigint DEFAULT NULL) RETURNS void * This function is used to create the trigger function for non-native time-based partitioning. * Normally this function is never called manually since function creation is managed by run_maintenance(). But if you need to force the re-creation of the trigger function, this will let you do that. * `p_parent_table` - parent table to recreate trigger function on. * 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.

create_function_id(p_parent_table text, p_job_id bigint DEFAULT NULL) RETURNS void

- This function is used to create the trigger function for non-native serial partitioning.
- Normally this function is never called manually since function creation is managed by run_maintenance(). But if you need to force the re-creation of the trigger function, this will let you do that.
- `p_parent_table` - parent table to recreate trigger function on.
- 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.

Maintenance Functions

run_maintenance(p_parent_table text DEFAULT NULL, p_analyze boolean DEFAULT NULL, 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 as noted above. The other parameters have postgresql.conf values that can be set. See BGW section above.
- 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.
- Will automatically update the trigger function for non-native partition sets to keep the parent table pointing at the correct partitions. When

using time, run this function more often than the partitioning interval to keep the trigger function running its most efficient. For example, if using quarter-hour, run every 5 minutes; if using daily, run at least twice a day, etc.

- `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` - For non-native partitioning and native partitioning in PG10, when a new child table is created, an analyze is run on the parent to ensure statistics are updated. For PG11+, this is no longer done, so it is not run by default then. For large partition sets, this 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. Set this to false (or just leave NULL for PG11+) to disable the analyze run and avoid this contention. For PG10 and older, please note that you must then schedule an analyze of the parent table at some point.
- `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)`

- For PG11+, this is the preferred method to run partition maintenance vs directly calling the run_maintenance() function.
- 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 that the parent tables (non-native) or default tables (native PG11+) of the partition sets that `pg_partman` manages do not 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 makes sense for the partition interval, not by the locale ordering of their names.
- For PG11+, 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_table text, OUT suffix_timestamp timestamp, OUT suffix_id bigint, OUT table_exists boolean)

- Given a 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

@extschema@.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.
- `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.
- `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 "2020_01_30" OR "920000"). Useful for generating your own suffixes for partitioning similar to how pg_partman does it.

`@extschema@.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 sub-partition sets
- `p_ignore_template` - For native partitioned tables 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 p_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 included python script (reapply_constraint.py). Or if you're on PG11+, use the `reapply_constraints_proc` procedure. Both these methods have 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)`

- 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 constraints 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 included python script (`reapply_constraint.py`). Or if you're on PG11+, use reapply_constraints_proc(). These both have 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)` * Procedures for PG11+ to reapply the extra constraint 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. * `p_apply_constraints` - Apply constraints on configured columns to all child tables older than the premake 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.

`reapply_privileges(p_parent_table text)`

- 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`.

`apply_foreign_keys(p_parent_table text, p_child_table text DEFAULT NULL, p_job_id bigint DEFAULT NULL, p_debug boolean DEFAULT false)` * IMPORTANT: This function is no longer necessary for PG11+ since FK inheritance is automatically managed. * Applies any foreign keys that exist on a parent table in a partition set to all the child tables for PG10 and older. * This function is automatically called whenever a new child table is created, so there is no need to manually run it unless you need to fix an existing child table. * If you need to apply this to an entire partition set, see the **reapply_foreign_keys.py** python script. This will commit after every FK creation to avoid contention. * This function can be used on any table inheritance set, not just ones managed by `pg_partman`. * 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. * The p_debug parameter will show you the constraint creation statement that was used.

`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_batch_count int DEFAULT 1, p_batch_interval text DEFAULT NULL, p_keep_table boolean DEFAULT true, p_lock_wait numeric DEFAULT 0, p_target_table text DEFAULT NULL, 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 either the parent table (non-native) or the given target table (native).
- If you are trying to un-partition a large amount of data automatically, it is recommended to either use the `undo_partition.py` script to commit data in smaller batches. Or if you're on PG11+, 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 UNINHERITED/UNATTACHED. This leave previous child tables as empty, independent tables.
- For non-native, when this function is run, the trigger on the parent table & the trigger function are immediately dropped (if they still exist). This means any further writes are done to the parent.
- Without setting either batch argument manually, each run of the function will move all the data from a single partition into the parent/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_batch_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 sized 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 uninherited/unattached 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_target_table` - A schema-qualified table to move the old partitioned table's data to. Required for undoing a native partition set since data cannot be moved to the parent. Schema can be different from original table.
- `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 sub-partition 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 sub-partition 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_interval text DEFAULT NULL, p_batch int DEFAULT NULL, p_wait int DEFAULT 1, p_target_table text DEFAULT NULL, p_keep_table boolean DEFAULT true, p_lock_wait int DEFAULT 0, p_lock_wait_tries int DEFAULT 10, p_quiet boolean DEFAULT false, p_ignored_columns text[] DEFAULT NULL, p_drop_cascade boolean DEFAULT false)

- A procedure that can un-partition data in distinct commit batches to avoid long running transactions and data contention issues.
- Only works with PostgreSQL 11+
- Calls either undo_partition() function in a loop committing as needed.
- `p_parent_table` - Parent table of an already created partition set.
- `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_batch` - How many times to loop through the value given for –interval. If –interval not set, will use default partition interval and undo at most -b partition(s). Procedure commits at the end of each individual batch.

(NOT passed as p_batch_count to undo_partition function). If not set, all data in the entire partition set will be moved in a single run of the procedure.

- `p_wait` - Cause the procedure to pause for a given number of seconds between commits (batches) to reduce write load
- `p_target_table` - Same as the p_target_table option in the undo_partition() function.
- `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 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_quiet` - Procedures cannot return values, so by default it emits NOTICE's to show progress. Set this option to silence these notices.
- `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 sub-partition 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 sub-partition parent to be dropped when that parent is dropped.

`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) RETURNS int`

- This function is used to drop child tables from a time-based partition set. 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 just 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 just use the value configured in **part_config** if not explicitly set. This option is ignored if p_keep_table is set to FALSE or if retention_schema is set.
- `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 just use the value configured in **`part_config`** if not explicitly set. If this option is set, the retention `p_keep_table` & `p_keep_index` parameters are 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 id-based partition set. 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 just 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 just use the value configured in **part_config** if not explicitly set. This option is ignored if p_keep_table is set to FALSE or if retention_schema is set.

- `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 just use the value configured in **part_config** if not explicitly set. If this option is set, the retention p_keep_table & p_keep_index parameters are ignored.
- Returns the number of partitions affected.

`drop_partition_column(p_parent_table text, p_column text) RETURNS void`

- Depending on when a column was added (before or after partitioning was set up), dropping it on the parent may or may not drop it from all children. This function is used to ensure a column is always dropped from the parent and all children in a partition set.
- This should only be relevant for non-native partition sets.
- Uses the IF EXISTS clause in all drop statements, so it may spit out notices/warnings that a column was not found. You can safely ignore these warnings. It should not spit out any errors.

Tables

`part_config`

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

- `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_type`
  - Type of partitioning. Must be one of the types mentioned above in the `create_parent()` info.
- `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.
- `constraint_cols`
  - Array column that lists columns to have additional constraints applied. See **About** section for more information on how this feature works.
- `premake`
  - How many partitions to keep pre-made ahead of the current partition. Default is 4.
- `optimize_trigger`
  - Manages number of partitions which are handled most efficiently by trigger. See `create_parent()` function for more info. Default 4.
  - This option is ignored for native partitioning.
- `optimize_constraint`

    - Manages which old tables get additional constraints set if configured to do so. See **About** section for more info. Default 30.
- `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'.
- `inherit_fk`
  - Set whether `pg_partman` manages inheriting foreign keys from the parent table to all children.
  - Defaults to TRUE. Can be set with the `create_parent()` function at creation time as well.
  - This option is currently ignored for native partitioning.
- `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 serial 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_* options.
- `retention_keep_table`
  - Boolean value to determine whether dropped child tables are kept 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.
- `retention_keep_index`
  - NOTE: This setting has no affect on native partitioning in PG11+. You cannot drop natively inherited child indexes.
  - Boolean value to determine whether indexes are dropped for child tables that are uninherited.
  - Default is TRUE. Set to FALSE to have the child table's indexes dropped when it is uninherited.
- `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.
- `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.
- `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 sub-partition 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.
- `trigger_exception_handling`
  - This option is ignored for native partitioning.
  - Boolean value that can be set to allow the partitioning trigger function to handle any exceptions encountered while writing to this table. Handling it in this case means putting the data into the parent table to try and ensure no data loss in case of errors. Be aware that catching the exception here will override any other exception handling that may be done when writing to this partitioned set (Ex. handling a unique constraint violation to ignore it). Just the existence of this exception block will also increase xid consumption since every row inserted will increment the global xid value. If this is table has a high insert rate, you can quickly reach xid wraparound, so use this carefully. This option is set to false by default to avoid causing unexpected behavior in other exception handling situations.
- `upsert`
  - Please note this option will be going away in the near future once PG11 has been out for a while.
  - text value of the ON CONFLICT clause to include in the partition trigger Defaults to " (empty string) which means it's inactive. See *create_parent()* function definition & *About* section for more info.
  - This option is currently ignored for native partitioning since there is no trigger, but upsert is still able to work in a limited fashion.
- `trigger_return_null`
  - Boolean value that allows controlling the behavior of the partition trigger RETURN. By default this is true and the trigger returns NULL to prevent data going into the parent table as well as the children. However, if you have multiple triggers and are relying on the return to be the NEW column value, this can cause a problem. Setting this config value to false will cause the partition trigger to RETURN NEW. You are then responsible for handling the return

    value in another trigger appropriately. Otherwise, this will cause new data to go into both the child and parent table of the partition set.
  - This option is ignored for native partitioning
- `template_table`
  - The schema-qualified name of the table used as a template for applying any inheritance options not handled by the native partitioning options in PG.
- `inherit_privileges`
  - Sets whether to inherit the ownership/privileges of the parent table to all child tables. Defaults to true for non-native & PG10. Defaults to false for native PG11+ 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
- `drop_cascade_fk` - Allow dropping of foreign key references to be cascaded when a child table is dropped. This option is only allowed with non-native partitioning and is not supported with subpartitioning.

`part_config_sub`

- Stores all configuration data for sub-partitioned 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.

`partition_data.py`

- NOTE: This script is only installed for PostgreSQL 10 and lower. It has been replaced by `partition_data_proc()`.
- A python script to make partitioning in committed batches easier.
- Script currently does not work with native partitioning.
- Calls either partition_data_time() or partition_data_id() depending on the value given for –type.
- A commit is done at the end of each –interval and/or fully created partition.
- Returns the total number of rows moved to partitions. Automatically stops when parent is empty.
- To help avoid heavy load and contention during partitioning, autovacuum is turned off for the parent table and all child tables when this script is run. When partitioning is complete, autovacuum is set back to its default value and the parent table is vacuumed when it is emptied.
- `--parent (-p)`: Parent table of an already created partition set. Required.
- `--type (-t)`: Type of partitioning. Valid values are "time" and "id". Required.
- `--connection (-c)`: Connection string for use by psycopg. Defaults to "host=" (local socket).
- `--interval (-i)`: Value that is passed on to the partitioning 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.
- `--batch (-b)`: How many times to loop through the value given for –interval. If –interval not set, will use default partition interval and make at most -b partition(s). Script commits at the end of each individual batch. (NOT passed as p_batch_count to partitioning function). If not set, all data in the parent table will be partitioned in a single run of the script.
- `--wait (-w)`: Cause the script to pause for a given number of seconds between commits (batches).
- `--order (-o)`: Allows you to specify the order that data is migrated from the parent to the children, either ascending (ASC) or descending (DESC). Default is ASC.
- `--lockwait (-l)`: Have a lock timeout of this many seconds on the data move. If a lock is not obtained, that batch will be tried again.
- `--lockwait_tries`: Number of times to allow a lockwait to time out before giving up on the partitioning. Defaults to 10.
- `--autovacuum_on`: Turning autovacuum off requires a brief lock to ALTER the table property. Set this option to leave autovacuum on and avoid the lock attempt.
- `--quiet (-q)`: Switch setting to stop all output during and after parti-

tioning.

- `--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.
- `--debug` Show additional debugging output
- Examples:

```
Partition all data in a parent table. Commit after each partition is made.
      python partition_data.py -c "host=localhost dbname=mydb" -p schema.parent_table -t tim
Partition by id in smaller intervals and pause between them for 5 seconds (assume >100 parti
      python partition_data.py -p schema.parent_table -t id -i 100 -w 5
Partition by time in smaller intervals for at most 10 partitions in a single run (assume mor
      python partition_data.py -p schema.parent_table -t time -i "1 week" -b 10
```

undo_partition.py

- NOTE: This script is only installed for PostgreSQL 10 and lower. It has been replaced by `undo_partition_proc()`.
- A python script to make undoing partitions in committed batches easier.
- Can also work on any non-native parent/child partition set not managed by `pg_partman` if –type option is not set.
- This script calls either undo_partition(), undo_partition_time() or undo_partition_id depending on the value given for –type.
- A commit is done at the end of each –interval and/or emptied partition.
- Returns the total number of child tables undone. Automatically stops when last child table is undone.
- `--parent (-p)`: Parent table of the partition set. Required.
- `--type (-t)`: Type of partitioning. Valid values are "time", "id", & "native". Not setting this argument will use undo_partition() and work on any non-native parent/child table set.
- `--connection (-c)`: Connection string for use by psycopg. Defaults to "host=" (local socket).
- `--interval (-i)`: Value that is passed on to the partitioning function as `p_batch_interval`. 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.
- `--batch (-b)`: How many times to loop through the value given for –interval. If –interval not set, will use default partition interval and undo at most -b partition(s). Script commits at the end of each individual batch. (NOT passed as p_batch_count to undo function). If not set, all data will be moved to the parent table in a single run of the script.
- `--wait (-w)`: Cause the script to pause for a given number of seconds between commits (batches).
- `--droptable (-d)`: Switch setting for whether to drop child tables when they are empty. Leave off option to just uninherit.
- `--quiet (-q)`: Switch setting to stop all output during and after partitioning undo.

- `--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.
- `--debug`: Show additional debugging output

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.

reapply_indexes.py

- A python script for reapplying indexes on child tables in a partition set after they are changed on the parent table.
- NOTE: This script only works with non-native partitioning and native partitioning in PG10. It does not worth with PG11+ and likely won't be needed since most indexes are now managed automatically in those versions. The only exception may be primary & unique indexes that don't include the partition column. There's unfortunately no easy way to manage index inheritance in PG11+ via this script and will have to be handled manually.
- Any indexes that currently exist on the children and match the definition

on the parent will be left as is. There is an option to recreate matching as well indexes if desired, as well as the primary key.

- Indexes that do not exist on the parent will be dropped from all children.
- Commits are done after each index is dropped/created to help prevent long running transactions & locks.
- NOTE: New index names are made based off the child table name & columns used, so their naming may differ from the name given on the parent. This is done to allow the tool to account for long or duplicate index names. If an index name would be duplicated, an incremental counter is added on to the end of the index name to allow it to be created. Use the –dryrun option first to see what it will do and which names may cause dupes to be handled like this.
- `--parent (-p)`: Parent table of an already created partition set. Required.
- `--connection (-c)`: Connection string for use by psycopg. Defaults to "host=" (local socket).
- `--concurrent`: Create indexes with the CONCURRENTLY option. Note this does not work on primary keys when –primary is given.
- `--drop_concurrent`: Drop indexes concurrently when recreating them (PostgreSQL >= v9.2). Note this does not work on primary keys when –primary is given.
- `--recreate_all (-R)`: By default, if an index exists on a child and matches the parent, it will not be touched. Setting this option will force all child indexes to be dropped & recreated. Will obey the –concurrent & –drop_concurrent options if given. Will not recreate primary keys unless –primary option is also given.
- `--primary`: By default the primary key is not recreated. Set this option if that is needed. Note this will cause an exclusive lock on the child table for the duration of the recreation.
- `--jobs (-j)`: Use the python multiprocessing library to recreate indexes in parallel. Note that this is per table, not per index. Be very careful setting this option if load is a concern on your systems.
- `--wait (-w)`: Wait the given number of seconds after indexes have finished being created on a table before moving on to the next. When used with `-j`, this will set the pause between the batches of parallel jobs instead.
- `--dryrun`: Show what the script will do without actually running it against the database. Highly recommend reviewing this before running. Note that if multiple indexes would get the same default name, the duplicated names will show in the dryrun (because the index doesn't exist in the catalog to check for it). When the real thing is run, the duplicated names will be handled as stated in the NOTE above.
- `--quiet`: Turn off all output.
- `--nonpartman` If the partition set you are running this on is not managed by pg_partman, set this flag otherwise this script may not work. Note that the pg_partman extension is still required to be installed for this to work since it uses certain internal functions. When this is set the order that the tables are reindexed is alphabetical instead of logical.

- `--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.

`reapply_constraints.py` * NOTE: This script is only installed for PostgreSQL 10 and lower. It has been replaced by `reapply_constraints_proc()`. * A python script for redoing constraints on child tables in a given partition set for the columns that are configured in **part_config** table. * Typical usage would be -d mode to drop constraints, edit the data as needed, then -a mode to reapply constraints. * `--parent (-p)`: Parent table of an already created partition set. (Required) * `--connection (-c)`: Connection string for use by psycopg. Defaults to "host=" (local socket). * `--drop_constraints (-d)`: Drop all constraints managed by `pg_partman`. Drops constraints on ALL child tables in the partition set. * `--add_constraints (-a)`: Apply constraints on configured columns to all child tables older than the premake value. * `--jobs (-j)`: Use the python multiprocessing library to recreate indexes in parallel. Value for -j is number of simultaneous jobs to run. Note that this is per table, not per index. Be very careful setting this option if load is a concern on your systems. * `--wait (-w)`: Wait the given number of seconds after a table has had its constraints dropped or applied before moving on to the next. When used with -j, this will set the pause between the batches of parallel jobs instead. * `--dryrun`: Show what the script will do without actually running it against the database. Highly recommend reviewing this before running. * `--quiet (-q)`: Turn off all 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.

`reapply_foreign_keys.py`

- NOTE: This script is only installed for PostgreSQL 10 and lower.
- A python script for redoing the inherited foreign keys for an entire partition set.
- Script currently does not work with native partitioning and FKs are handled natively as of PG11+.
- All existing foreign keys on all child tables are dropped and the foreign keys that exist on the parent at the time this is run will be applied to all children.
- Commits after each foreign key is created to avoid long periods of contention.
- `--parent (-p)`: Parent table of an already created partition set. (Required)
- `--connection (-c)`: Connection string for use by psycopg. Defaults to "host=" (local socket).
- `--quiet (-q)`: Switch setting to stop all output during and after partitioning undo.
- `--dryrun`: Show what the script will do without actually running it against the database. Highly recommend reviewing this before running.
- `--nonpartman` If the partition set you are running this on is not managed

by pg_partman, set this flag. Otherwise internal pg_partman functions are used and this script may not work. When this is set the order that the tables are rekeyed is alphabetical instead of logical.
- `--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.
- `--debug`: Show additional debugging output

`check_unique_constraints.py`

- Partitioning using inheritance (native still does this internally) has the shortcoming of not allowing a unique constraint to apply to all tables in the entire partition set without causing large performance issues once the partition set begins to grow very large. 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 a large chunk 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 4.4.1 of pg_partman with PostgreSQL 11 or higher.

Note that all examples here are for native partitioning. If you need to use non-native, trigger-based partitioning, please see the Trigger-based HowTo file.

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 for 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  | Stats
--------+--------------------------+-----------+----------+---------------+----------+------
 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, depending on the version of PostgreSQL.

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('partman_test.time_taptest_table', 'col3', 'native', 'daily', p
 create_parent
---------------
 t
(1 row)
```

```
\d+ partman_test.time_taptest_table
                               Partitioned table "partman_test.time_taptest_table"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | Stats
--------+--------------------------+-----------+----------+---------------+----------+------
 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_p2020_10_26 FOR VALUES FROM ('2020-10-26 00:00:0
            partman_test.time_taptest_table_p2020_10_27 FOR VALUES FROM ('2020-10-27 00:00:0
            partman_test.time_taptest_table_p2020_10_28 FOR VALUES FROM ('2020-10-28 00:00:0
            partman_test.time_taptest_table_p2020_10_29 FOR VALUES FROM ('2020-10-29 00:00:0
            partman_test.time_taptest_table_p2020_10_30 FOR VALUES FROM (’2020-10-30 00:00:0
            partman_test.time_taptest_table_p2020_10_31 FOR VALUES FROM (’2020-10-31 00:00:0
            partman_test.time_taptest_table_p2020_11_01 FOR VALUES FROM (’2020-11-01 00:00:0
            partman_test.time_taptest_table_p2020_11_02 FOR VALUES FROM (’2020-11-02 00:00:0
            partman_test.time_taptest_table_p2020_11_03 FOR VALUES FROM (’2020-11-03 00:00:0
            partman_test.time_taptest_table_default DEFAULT
```

```
\d+ partman_test.time_taptest_table_p2020_10_26
                               Table "partman_test.time_taptest_table_p2020_10_26"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | Stats
--------+--------------------------+-----------+----------+---------------+----------+------
 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 (’2020-10-26 00:00:00-04’) TO
Partition constraint: ((col3 IS NOT NULL) AND (col3 >= ’2020-10-26 00:00:00-04’::timestamp w
Indexes:
    "time_taptest_table_p2020_10_26_pkey" PRIMARY KEY, btree (col1)
    "time_taptest_table_p2020_10_26_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
    , col2 text not null
    , col3 timestamptz DEFAULT now()
    , 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  | Stats targe
--------+--------------------------+-----------+----------+---------+----------+------------
 col1   | bigint                   |           |          |         | plain    |
 col2   | text                     |           | not null |         | extended |
 col3   | timestamp with time zone |           |          | 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(’partman_test.id_taptest_table’, ’col1’, ’native’, ’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  | Stats targe
--------+--------------------------+-----------+----------+---------+----------+------------
 col1   | bigint                   |           |          |         | plain    |
 col2   | text                     |           | not null |         | extended |
 col3   | timestamp with time zone |           |          | 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
                 template_table
------------------------------------------------
 partman.template_partman_test_id_taptest_table
```

```
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  | Stats targe
--------+--------------------------+-----------+----------+---------+----------+------------
 col1   | bigint                   |           |          |         | plain    |
 col2   | text                     |           | not null |         | extended |
 col3   | timestamp with time zone |           |          | 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).

```
\d partman_test.id_taptest_table_p40
             Table "partman_test.id_taptest_table_p40"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | bigint                   |           |          |
 col2   | text                     |           | not null |
 col3   | timestamp with time zone |           |          | 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)

\d partman_test.id_taptest_table_p50
             Table "partman_test.id_taptest_table_p50"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | bigint                   |           |          |
 col2   | text                     |           | not null |
 col3   | timestamp with time zone |           |          | 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)

\d partman_test.id_taptest_table_p60
             Table "partman_test.id_taptest_table_p60"
 Column |           Type           | Collation | Nullable | Default
--------+--------------------------+-----------+----------+---------
 col1   | bigint                   |           |          |
 col2   | text                     |           | not null |
 col3   | timestamp with time zone |         |            | 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 methods that could be employed with the old trigger-based methods. 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.

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 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('public.original_table', 'col1', 'native', '10000');
```

```
\d+ original_table;
                                 Partitioned table "public.original_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Stats targe
--------+--------------------------+-----------+----------+---------+----------+------------
 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)
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).

```
CALL partman.partition_data_proc('public.original_table', p_interval := '1000', p_batch := 2
NOTICE:  Batch: 1, Rows moved: 1000
NOTICE:  Batch: 2, Rows moved: 1000
NOTICE:  Batch: 3, Rows moved: 1000
NOTICE:  Batch: 4, Rows moved: 1000
NOTICE:  Batch: 5, Rows moved: 1000
NOTICE:  Batch: 6, Rows moved: 1000
NOTICE:  Batch: 7, Rows moved: 1000
NOTICE:  Batch: 8, Rows moved: 1000
NOTICE:  Batch: 9, Rows moved: 1000
NOTICE:  Batch: 10, Rows moved: 999
NOTICE:  Batch: 11, Rows moved: 1000
NOTICE:  Batch: 12, Rows moved: 1000
[...]
NOTICE:  Batch: 100, Rows moved: 1000
NOTICE:  Batch: 101, Rows moved: 1
NOTICE:  Total rows moved: 100000
NOTICE:  Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning d
CALL
Time: 103206.205 ms (01:43.206)


VACUUM ANALYZE public.original_table;
VACUUM
```

```
Time: 352.973 ms
```

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.

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  | Stats targe
--------+--------------------------+-----------+----------+---------+----------+------------
 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)
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. 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
-------------------------------+-------------------------------
 2020-12-02 19:04:08.559646-05 | 2020-12-09 19:04:08.559646-05
(1 row)

SELECT partman.create_parent('public.new_partitioned_table', 'col3', 'native', 'daily', p_te
```

The next step is to drop the DEFAULT partition that pg_partman creates for you.

```
DROP TABLE public.new_partitioned_table_default;
```

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
-------------------------------
 2020-12-09 19:05:15.358796-05

\d+ new_partitioned_table;
                                          Partitioned table "public.new_partitioned_table"
 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_p2020_12_11 FOR VALUES FROM ('2020-12-11 00:00:00-05') TO
```

You also 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 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. A query to obtain this is provided in the SQL statements below.

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) 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_p2020_12_11 RENAME TO original_table_p2020_12_11;

-- 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_batch := 200);

NOTICE:  Batch: 1, Rows moved: 60
NOTICE:  Batch: 2, Rows moved: 288
NOTICE:  Batch: 3, Rows moved: 288
NOTICE:  Batch: 4, Rows moved: 288
NOTICE:  Batch: 5, Rows moved: 288
NOTICE:  Batch: 6, Rows moved: 288
```

```
NOTICE:  Batch: 7, Rows moved: 288
NOTICE:  Batch: 8, Rows moved: 229
NOTICE:  Total rows moved: 2017
NOTICE:  Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning
CALL
Time: 8432.725 ms (00:08.433)

VACUUM ANALYZE original_table;
VACUUM
Time: 60.690 ms
```

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 there are no child table names that don’t conform to the pattern that pg_partman uses for the new child tables. pg_partman expects a specific format for the child tables it manages, so if they don’t all match, maintenance may not work as expected

```
\d+ original_table;
                                              Partitioned table "public.original_table"
 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: original_table_p2020_12_02 FOR VALUES FROM ('2020-12-02 00:00:00-05') TO ('2020-
            original_table_p2020_12_03 FOR VALUES FROM ('2020-12-03 00:00:00-05') TO ('2020-
            original_table_p2020_12_04 FOR VALUES FROM ('2020-12-04 00:00:00-05') TO ('2020-
            original_table_p2020_12_05 FOR VALUES FROM ('2020-12-05 00:00:00-05') TO ('2020-
            original_table_p2020_12_06 FOR VALUES FROM ('2020-12-06 00:00:00-05') TO ('2020-
            original_table_p2020_12_07 FOR VALUES FROM ('2020-12-07 00:00:00-05') TO ('2020-
            original_table_p2020_12_08 FOR VALUES FROM ('2020-12-08 00:00:00-05') TO ('2020-
            original_table_p2020_12_09 FOR VALUES FROM ('2020-12-09 00:00:00-05') TO ('2020-
            original_table_p2020_12_11 FOR VALUES FROM ('2020-12-11 00:00:00-05') TO ('2020-
```

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
--------+--------------------------+-----------+----------+---------------------------------
 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: original_table_p2020_12_02 FOR VALUES FROM (’2020-12-02 00:00:00-05’) TO (’2020-
            original_table_p2020_12_03 FOR VALUES FROM (’2020-12-03 00:00:00-05’) TO (’2020-
            original_table_p2020_12_04 FOR VALUES FROM (’2020-12-04 00:00:00-05’) TO (’2020-
            original_table_p2020_12_05 FOR VALUES FROM (’2020-12-05 00:00:00-05’) TO (’2020-
            original_table_p2020_12_06 FOR VALUES FROM (’2020-12-06 00:00:00-05’) TO (’2020-
            original_table_p2020_12_07 FOR VALUES FROM (’2020-12-07 00:00:00-05’) TO (’2020-
            original_table_p2020_12_08 FOR VALUES FROM (’2020-12-08 00:00:00-05’) TO (’2020-
            original_table_p2020_12_09 FOR VALUES FROM (’2020-12-09 00:00:00-05’) TO (’2020-
            original_table_p2020_12_11 FOR VALUES FROM (’2020-12-11 00:00:00-05’) TO (’2020-
            original_table_p2020_12_12 FOR VALUES FROM (’2020-12-12 00:00:00-05’) TO (’2020-
            original_table_p2020_12_13 FOR VALUES FROM (’2020-12-13 00:00:00-05’) TO (’2020-
            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 (Dec 10, 2020). 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)

\d+ original_table;
                                              Partitioned table "public.original_table"
 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: original_table_p2020_12_02 FOR VALUES FROM ('2020-12-02 00:00:00-05') TO ('2020-
            original_table_p2020_12_03 FOR VALUES FROM ('2020-12-03 00:00:00-05') TO ('2020-
            original_table_p2020_12_04 FOR VALUES FROM ('2020-12-04 00:00:00-05') TO ('2020-
            original_table_p2020_12_05 FOR VALUES FROM ('2020-12-05 00:00:00-05') TO ('2020-
            original_table_p2020_12_06 FOR VALUES FROM ('2020-12-06 00:00:00-05') TO ('2020-
            original_table_p2020_12_07 FOR VALUES FROM ('2020-12-07 00:00:00-05') TO ('2020-
            original_table_p2020_12_08 FOR VALUES FROM ('2020-12-08 00:00:00-05') TO ('2020-
            original_table_p2020_12_09 FOR VALUES FROM ('2020-12-09 00:00:00-05') TO ('2020-
            original_table_p2020_12_10 FOR VALUES FROM ('2020-12-10 00:00:00-05') TO ('2020-
            original_table_p2020_12_11 FOR VALUES FROM ('2020-12-11 00:00:00-05') TO ('2020-
            original_table_p2020_12_12 FOR VALUES FROM ('2020-12-12 00:00:00-05') TO ('2020-
            original_table_p2020_12_13 FOR VALUES FROM ('2020-12-13 00:00:00-05') TO ('2020-
            original_table_default DEFAULT
```

You can see that created the missing table for Dec 10th.

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 as the 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(’public.original_table’, p_interval := ’1 hour’::text, p_ba

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:  Batch: 160, Partitions undone this batch: 0, Rows undone this batch: 13
NOTICE:  Moved 5 row(s) to the target table. Removed 1 partitions.
NOTICE:  Batch: 161, Partitions undone this batch: 1, Rows undone this batch: 5
NOTICE:  Moved 0 row(s) to the target table. Removed 4 partitions.
NOTICE:  Total partitions undone: 13, Total rows moved: 2017
NOTICE:  Ensure to VACUUM ANALYZE the old parent & target table after undo has finished
CALL
Time: 163465.195 ms (02:43.465)

VACUUM ANALYZE original_table;
VACUUM
```

```
Time: 20.706 ms
VACUUM ANALYZE new_regular_table;
VACUUM
Time: 20.375 ms
```

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) FROM public.original_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 2;
```

This HowTo guide will show you some examples of how to set up both simple, single level partitioning as well as multi-level sub-partitioning. It will also show you how to partition data out of a table that has existing data (see **Sub-partition ID->ID->ID**) and undo the partitioning of an existing partition set. 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 v3.0.1 of pg_partman. If you need a howto for a previous version, please see an older release available on github.

Note that all examples here are for non-native, trigger-based partitioning. Documentation for native partitioning is in the works, but it will mostly be centered around PostgreSQL 11 since 10 was very limited in its partitioning support.

Simple Time Based: 1 Partition Per Day

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

```
keith@keith=# SELECT partman.create_parent('partman_test.time_taptest_table', 'col3', 'p
 create_parent
---------------
 t
(1 row)

keith@keith=# \d+ partman_test.time_taptest_table
                                  Table "partman_test.time_taptest_table"
 Column |           Type           | Collation | Nullable | Default | Storage  | Stats t
--------+--------------------------+-----------+----------+---------+----------+--------
 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 EA
Child tables: partman_test.time_taptest_table_p2017_03_23,
              partman_test.time_taptest_table_p2017_03_24,
              partman_test.time_taptest_table_p2017_03_25,
              partman_test.time_taptest_table_p2017_03_26,
              partman_test.time_taptest_table_p2017_03_27,
              partman_test.time_taptest_table_p2017_03_28,
              partman_test.time_taptest_table_p2017_03_29,
              partman_test.time_taptest_table_p2017_03_30,
              partman_test.time_taptest_table_p2017_03_31
```

The trigger function most efficiently covers a specific period of time for 4 days before and 4 days after today. This can be adjusted with the `optimize_trigger` config option in the `part_config` table. Outside of that, a dynamic statement tries to find the appropriate child table to put the data into. Note this dynamic statement is far less efficient since a catalog lookup is required and the statement plan cannot be cached as well as looking up the that the child table exists. If the child table does not exist at all for the time value given, the data goes to the parent:

```
keith@keith=# \sf partman_test.time_taptest_table_part_trig_func
CREATE OR REPLACE FUNCTION partman_test.time_taptest_table_part_trig_func()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
            DECLARE
            v_count                 int;
            v_partition_name        text;
            v_partition_timestamp   timestamptz;
        BEGIN
```

```
IF TG_OP = ’INSERT’ THEN
    v_partition_timestamp := date_trunc(’day’, NEW.col3);
    IF NEW.col3 >= ’2017-03-27 00:00:00-04’ AND NEW.col3 < ’2017-03-28 00:00:00-04’
    INSERT INTO partman_test.time_taptest_table_p2017_03_27 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-26 00:00:00-04’ AND NEW.col3 < ’2017-03-27 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_26 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-28 00:00:00-04’ AND NEW.col3 < ’2017-03-29 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_28 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-25 00:00:00-04’ AND NEW.col3 < ’2017-03-26 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_25 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-29 00:00:00-04’ AND NEW.col3 < ’2017-03-30 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_29 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-24 00:00:00-04’ AND NEW.col3 < ’2017-03-25 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_24 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-30 00:00:00-04’ AND NEW.col3 < ’2017-03-31 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_30 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-23 00:00:00-04’ AND NEW.col3 < ’2017-03-24 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_23 VALUES (NEW.*) ;
    ELSIF NEW.col3 >= ’2017-03-31 00:00:00-04’ AND NEW.col3 < ’2017-04-01 00:00:00-0
        INSERT INTO partman_test.time_taptest_table_p2017_03_31 VALUES (NEW.*) ;
    ELSE
        v_partition_name := partman.check_name_length(’time_taptest_table’, to_char(
        SELECT count(*) INTO v_count FROM pg_catalog.pg_tables WHERE schemaname = ’p
        IF v_count > 0 THEN
            EXECUTE format(’INSERT INTO %I.%I VALUES($1.*) ’, ’partman_test’, v_part
        ELSE
            RETURN NEW;
        END IF;
    END IF;
END IF;
RETURN NULL;
END $function$
```

Simple Serial ID: 1 Partition Per 10 ID Values Starting With Empty Table

```
keith=# \d partman_test.id_taptest_table
               Table "partman_test.id_taptest_table"
 Column |           Type           |           Modifiers
--------+--------------------------+--------------------------------
 col1   | integer                  | not null
 col2   | text                     | not null default ’stuff’::text
 col3   | timestamp with time zone | default now()
Indexes:
    "id_taptest_table_pkey" PRIMARY KEY, btree (col1)
```

```
keith=# SELECT create_parent('partman_test.id_taptest_table', 'col1', 'partman', '10');
 create_parent
---------------
 t
(1 row)


keith=# \d+ partman_test.id_taptest_table
                                   Table "partman_test.id_taptest_table"
 Column |           Type           |           Modifiers            | Storage  | Stats t
--------+--------------------------+--------------------------------+----------+--------
 col1   | integer                  | not null                       | plain    |
 col2   | text                     | not null default 'stuff'::text | extended |
 col3   | timestamp with time zone | default now()                  | plain    |
Indexes:
    "id_taptest_table_pkey" PRIMARY KEY, btree (col1)
Triggers:
    id_taptest_table_part_trig BEFORE INSERT ON partman_test.id_taptest_table FOR EACH R
Child tables: partman_test.id_taptest_table_p0,
              partman_test.id_taptest_table_p10,
              partman_test.id_taptest_table_p20,
              partman_test.id_taptest_table_p30,
              partman_test.id_taptest_table_p40
```

This trigger function most efficiently covers for 4x10 intervals above the current max (0). Once max id reaches higher values, it will also efficiently cover up to 4x10 intervals behind the current max. Outside of that, a dynamic statement tries to find the appropriate child table to put the data into. And like I said for time above, the dynamic part is less efficient.

```
keith@keith=# \sf partman_test.id_taptest_table_part_trig_func
CREATE OR REPLACE FUNCTION partman_test.id_taptest_table_part_trig_func()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
    DECLARE
        v_count                     int;
        v_current_partition_id      bigint;
        v_current_partition_name    text;
        v_id_position               int;
        v_last_partition            text := 'id_taptest_table_p40';
        v_next_partition_id         bigint;
        v_next_partition_name       text;
        v_partition_created         boolean;
    BEGIN
    IF TG_OP = 'INSERT' THEN
```

```
        IF NEW.col1 >= 0 AND NEW.col1 < 10 THEN
            INSERT INTO partman_test.id_taptest_table_p0 VALUES (NEW.*) ;
        ELSIF NEW.col1 >= 10 AND NEW.col1 < 20 THEN
            INSERT INTO partman_test.id_taptest_table_p10 VALUES (NEW.*) ;
        ELSIF NEW.col1 >= 20 AND NEW.col1 < 30 THEN
            INSERT INTO partman_test.id_taptest_table_p20 VALUES (NEW.*) ;
        ELSIF NEW.col1 >= 30 AND NEW.col1 < 40 THEN
            INSERT INTO partman_test.id_taptest_table_p30 VALUES (NEW.*) ;
        ELSIF NEW.col1 >= 40 AND NEW.col1 < 50 THEN
            INSERT INTO partman_test.id_taptest_table_p40 VALUES (NEW.*) ;
        ELSE
            v_current_partition_id := NEW.col1 - (NEW.col1 % 10);
            v_current_partition_name := partman.check_name_length(’id_taptest_table’, v_
            SELECT count(*) INTO v_count FROM pg_catalog.pg_tables WHERE schemaname = ’p
ame;
            IF v_count > 0 THEN
                EXECUTE format(’INSERT INTO %I.%I VALUES($1.*) ’, ’partman_test’, v_curr
            ELSE
                RETURN NEW;
            END IF;
        END IF;
    END IF;
    RETURN NULL;
    END $function$
```

Simple Serial ID: 1 Partition Per 10 ID Values Starting With Empty Table and using upsert to drop conflicting rows

```
Uses same example table as above

keith@keith=# SELECT partman.create_parent(’partman_test.id_taptest_table’, ’col1’, ’par
 create_parent
---------------
 t
(1 row)

keith@keith=# \d+ partman_test.id_taptest_table
                                      Table "partman_test.id_taptest_table"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | S
--------+--------------------------+-----------+----------+---------------+----------+--
 col1   | integer                  |           | not null |               | plain    |
 col2   | text                     |           | not null | ’stuff’::text | extended |
 col3   | timestamp with time zone |           |          | now()         | plain    |
Indexes:
    "id_taptest_table_pkey" PRIMARY KEY, btree (col1)
Triggers:
    id_taptest_table_part_trig BEFORE INSERT ON partman_test.id_taptest_table FOR EACH R
Child tables: partman_test.id_taptest_table_p0,
              partman_test.id_taptest_table_p10,
              partman_test.id_taptest_table_p20,
              partman_test.id_taptest_table_p30,
              partman_test.id_taptest_table_p40
```

Other than the new ON CONFLICT clause, this trigger function works exactly the same as the previous ID example.

```
keith@keith=# \sf partman_test.id_taptest_table_part_trig_func
CREATE OR REPLACE FUNCTION partman_test.id_taptest_table_part_trig_func()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
    DECLARE
        v_count                     int;
        v_current_partition_id      bigint;
        v_current_partition_name    text;
        v_id_position               int;
        v_last_partition            text := 'id_taptest_table_p40';
        v_next_partition_id         bigint;
        v_next_partition_name       text;
        v_partition_created         boolean;
    BEGIN
    IF TG_OP = 'INSERT' THEN
        IF NEW.col1 >= 0 AND NEW.col1 < 10 THEN
            INSERT INTO partman_test.id_taptest_table_p0 VALUES (NEW.*) ON CONFLICT (col
        ELSIF NEW.col1 >= 10 AND NEW.col1 < 20 THEN
            INSERT INTO partman_test.id_taptest_table_p10 VALUES (NEW.*) ON CONFLICT (co
        ELSIF NEW.col1 >= 20 AND NEW.col1 < 30 THEN
            INSERT INTO partman_test.id_taptest_table_p20 VALUES (NEW.*) ON CONFLICT (co
        ELSIF NEW.col1 >= 30 AND NEW.col1 < 40 THEN
            INSERT INTO partman_test.id_taptest_table_p30 VALUES (NEW.*) ON CONFLICT (co
        ELSIF NEW.col1 >= 40 AND NEW.col1 < 50 THEN
            INSERT INTO partman_test.id_taptest_table_p40 VALUES (NEW.*) ON CONFLICT (co
        ELSE
            v_current_partition_id := NEW.col1 - (NEW.col1 % 10);
            v_current_partition_name := partman.check_name_length('id_taptest_table', v_
            SELECT count(*) INTO v_count FROM pg_catalog.pg_tables WHERE schemaname = 'p
ame;
            IF v_count > 0 THEN
                EXECUTE format('INSERT INTO %I.%I VALUES($1.*) ON CONFLICT (col1) DO NOT
            ELSE
                RETURN NEW;
            END IF;
        END IF;
    END IF;
    RETURN NULL;
    END $function$
```

Running the following query will insert a row in the table

```
keith@keith=# INSERT INTO partman_test.id_taptest_table(col1,col2) VALUES(1,'insert1');
INSERT 0 0
Time: 4.876 ms
keith@keith=# SELECT * FROM partman_test.id_taptest_table;
 col1 |  col2   |             col3
------+---------+-------------------------------
    1 | insert1 | 2017-03-27 14:23:02.769999-04
(1 row)
```

Running the following query will not fail but the row in the table will not change and col2 will still be 'insert1'

```
keith@keith=# INSERT INTO partman_test.id_taptest_table(col1,col2) VALUES(1,'insert2');
INSERT 0 0
Time: 1.583 ms
keith@keith=# SELECT * FROM partman_test.id_taptest_table;
 col1 |  col2   |             col3
------+---------+-------------------------------
    1 | insert1 | 2017-03-27 14:23:02.769999-04
(1 row)
```

Simple Serial ID: 1 Partition Per 10 ID Values Starting With Empty Table and using upsert to update conflicting rows

```
Uses same example table as above

keith@keith=# SELECT partman.create_parent('partman_test.id_taptest_table', 'col1', 'par
 create_parent
---------------
 t
(1 row)

keith@keith=# \d+ partman_test.id_taptest_table
                                      Table "partman_test.id_taptest_table"
 Column |           Type           | Collation | Nullable |    Default    | Storage  | S
--------+--------------------------+-----------+----------+---------------+----------+--
 col1   | integer                  |           | not null |               | plain    |
 col2   | text                     |           | not null | 'stuff'::text | extended |
 col3   | timestamp with time zone |           |          | now()         | plain    |
Indexes:
    "id_taptest_table_pkey" PRIMARY KEY, btree (col1)
Triggers:
    id_taptest_table_part_trig BEFORE INSERT ON partman_test.id_taptest_table FOR EACH
Child tables: partman_test.id_taptest_table_p0,
              partman_test.id_taptest_table_p10,
              partman_test.id_taptest_table_p20,
              partman_test.id_taptest_table_p30,
              partman_test.id_taptest_table_p40
```

Other than the new ON CONFLICT clause, this trigger function works exactly the same as the previous ID example.

```
keith@keith=# \sf partman_test.id_taptest_table_part_trig_func
CREATE OR REPLACE FUNCTION partman_test.id_taptest_table_part_trig_func()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
    DECLARE
        v_count                     int;
        v_current_partition_id      bigint;
        v_current_partition_name    text;
        v_id_position               int;
        v_last_partition            text := ’id_taptest_table_p40’;
        v_next_partition_id         bigint;
        v_next_partition_name       text;
        v_partition_created         boolean;
    BEGIN
    IF TG_OP = ’INSERT’ THEN
        IF NEW.col1 >= 0 AND NEW.col1 < 10 THEN
            INSERT INTO partman_test.id_taptest_table_p0 VALUES (NEW.*) ON CONFLICT (col
        ELSIF NEW.col1 >= 10 AND NEW.col1 < 20 THEN
            INSERT INTO partman_test.id_taptest_table_p10 VALUES (NEW.*) ON CONFLICT (co
        ELSIF NEW.col1 >= 20 AND NEW.col1 < 30 THEN
            INSERT INTO partman_test.id_taptest_table_p20 VALUES (NEW.*) ON CONFLICT (co
        ELSIF NEW.col1 >= 30 AND NEW.col1 < 40 THEN
            INSERT INTO partman_test.id_taptest_table_p30 VALUES (NEW.*) ON CONFLICT (co
        ELSIF NEW.col1 >= 40 AND NEW.col1 < 50 THEN
            INSERT INTO partman_test.id_taptest_table_p40 VALUES (NEW.*) ON CONFLICT (co
        ELSE
            v_current_partition_id := NEW.col1 - (NEW.col1 % 10);
            v_current_partition_name := partman.check_name_length(’id_taptest_table’, v_
            SELECT count(*) INTO v_count FROM pg_catalog.pg_tables WHERE schemaname = ’p
ame;
            IF v_count > 0 THEN
                EXECUTE format(’INSERT INTO %I.%I VALUES($1.*) ON CONFLICT (col1) DO UP
’, v_current_partition_name) USING NEW;
            ELSE
                RETURN NEW;
            END IF;
        END IF;
    END IF;
    RETURN NULL;
    END $function$
```

Running the following query will insert a row in the table

```
keith@keith=# INSERT INTO partman_test.id_taptest_table(col1,col2) VALUES(1,’insert1’);
INSERT 0 0
Time: 6.012 ms
keith@keith=# SELECT * FROM partman_test.id_taptest_table;
 col1 |  col2   |             col3
------+---------+------------------------------
    1 | insert1 | 2017-03-27 14:32:26.59552-04
(1 row)
```

Running the following query will not fail and the row in the table will change and col2 will now be ‘insert2’ and the timestamp in col3 will update to the default value now()

```
keith@keith=# INSERT INTO partman_test.id_taptest_table(col1,col2) VALUES(1,’insert2’);
INSERT 0 0
Time: 4.235 ms
keith@keith=# SELECT * FROM partman_test.id_taptest_table;
 col1 |  col2   |             col3
------+---------+-------------------------------
    1 | insert2 | 2017-03-27 14:33:00.949928-04
(1 row)
```

Sub-partition Time->Time->Time: Yearly -> Monthly -> Daily

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

Create top yearly partition set that only covers 2 years forward/back

```
keith@keith=# SELECT partman.create_parent(’partman_test.time_taptest_table’, ’col3’, ’p
 create_parent
---------------
 t
(1 row)
```

Now tell pg_partman to partition all yearly child tables by month. Do this by giving it the parent table of the yearly partition set (happens to be the same as above)

```
keith@keith=# SELECT partman.create_sub_parent('partman_test.time_taptest_table', 'col3'
 create_sub_parent
-------------------
 t
(1 row)

keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' ORDER BY
             tablename
-----------------------------------
 time_taptest_table
 time_taptest_table_p2015
 time_taptest_table_p2015_p2015_01
 time_taptest_table_p2016
 time_taptest_table_p2016_p2016_01
 time_taptest_table_p2017
 time_taptest_table_p2017_p2017_01
 time_taptest_table_p2017_p2017_02
 time_taptest_table_p2017_p2017_03
 time_taptest_table_p2017_p2017_04
 time_taptest_table_p2017_p2017_05
 time_taptest_table_p2018
 time_taptest_table_p2018_p2018_01
 time_taptest_table_p2019
 time_taptest_table_p2019_p2019_01
(15 rows)
```

The day this tutorial was updated is 2017-03-27. You now see that this causes only 2 new future partitions to be created. And for the monthly partitions, they have been created to cover 2 months ahead as well. Note that the trigger will still cover 4 ahead and 4 behind for both partition levels unless you change the `optimize_trigger` option in the config table. A parent table ALWAYS has at least one child, so for the time period that is outside of what the premake covers, just a single table has been made for the lowest possible month in that yearly time period (January). Now tell pg_partman to partition every monthly table that currently exists by day. Do this by giving it the parent table of each monthly partition set (the parent with the just the year suffix since its children are the monthly partitions).

```
SELECT partman.create_sub_parent('partman_test.time_taptest_table_p2015', 'col3', 'partm
SELECT partman.create_sub_parent('partman_test.time_taptest_table_p2016', 'col3', 'partm
SELECT partman.create_sub_parent('partman_test.time_taptest_table_p2017', 'col3', 'partm
SELECT partman.create_sub_parent('partman_test.time_taptest_table_p2018', 'col3', 'partm
SELECT partman.create_sub_parent('partman_test.time_taptest_table_p2019', 'col3', 'partm
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = ’partman_test’ ORDER B
                  tablename
----------------------------------------------
 time_taptest_table
 time_taptest_table_p2015
 time_taptest_table_p2015_p2015_01
 time_taptest_table_p2015_p2015_01_p2015_01_01
 time_taptest_table_p2016
 time_taptest_table_p2016_p2016_01
 time_taptest_table_p2016_p2016_01_p2016_01_01
 time_taptest_table_p2017
 time_taptest_table_p2017_p2017_01
 time_taptest_table_p2017_p2017_01_p2017_01_01
 time_taptest_table_p2017_p2017_02
 time_taptest_table_p2017_p2017_02_p2017_02_01
 time_taptest_table_p2017_p2017_03
 time_taptest_table_p2017_p2017_03_p2017_03_25
 time_taptest_table_p2017_p2017_03_p2017_03_26
 time_taptest_table_p2017_p2017_03_p2017_03_27
 time_taptest_table_p2017_p2017_03_p2017_03_28
 time_taptest_table_p2017_p2017_03_p2017_03_29
 time_taptest_table_p2017_p2017_04
 time_taptest_table_p2017_p2017_04_p2017_04_01
 time_taptest_table_p2017_p2017_05
 time_taptest_table_p2017_p2017_05_p2017_05_01
 time_taptest_table_p2018
 time_taptest_table_p2018_p2018_01
 time_taptest_table_p2018_p2018_01_p2018_01_01
 time_taptest_table_p2019
 time_taptest_table_p2019_p2019_01
 time_taptest_table_p2019_p2019_01_p2019_01_01
(28 rows)
```

Again, assuming today’s date is 2017-03-27, it has created the sub-partitions to cover 2 days in the future. All other parent tables outside of the current time period have the lowest possible day created for them.

Sub-partition ID->ID->ID: 10,000 -> 1,000 -> 100

This partition set has existing data already in it. We will partition it out using the python script found in the “bin” directory of the repo. It is possible to use the partition_data_id() function in postgres as well, but that would partition all the data out in a single transaction which, for a live table, could cause serious contention & I/O issues. The python script allows commits to be done in batches and avoid that contention and you can add a pause in between batches to limit

I/O activity. The p_jobmon flag being set in the creation functions is done just to keep the spamming of the jobmon logs to a minimum for these test examples.

```
keith@keith=# \d partman_test.id_taptest_table
                  Table "partman_test.id_taptest_table"
 Column |           Type           | Collation | Nullable |    Default
--------+--------------------------+-----------+----------+----------------
 col1   | integer                  |           | not null |
 col2   | text                     |           | not null | 'stuff'::text
 col3   | timestamp with time zone |           |          | now()
Indexes:
    "id_taptest_table_pkey" PRIMARY KEY, btree (col1)


keith@keith=# SELECT count(*) FROM partman_test.id_taptest_table ;
 count
--------
 100000
(1 row)

keith@keith=# SELECT min(col1), max(col1) FROM partman_test.id_taptest_table ;
 min |  max
-----+--------
   1 | 100000
(1 row)
```

Since there is already data in the table, the child tables initially created will be based around the max value, two before it and two after it. As stated above for time, the trigger still covers for 4 partitions before & after most efficiently, so if you need to adjust that as well, see the `part_config` table.

```
keith@keith=# SELECT partman.create_parent('partman_test.id_taptest_table', 'col1', 'par
---------------
 t
(1 row)

keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' ORDER BY
        tablename
-------------------------
 id_taptest_table
 id_taptest_table_p100000
 id_taptest_table_p110000
 id_taptest_table_p120000
 id_taptest_table_p80000
 id_taptest_table_p90000
(6 rows)
```

However, the data still resides in the parent table at this time. To partition it out, use the python script as mentioned above. The options below will cause it to commit every 100 rows. If the interval option was not given, it would commit them at the configured interval of 10,000. Allowing a lower interval decreases the possible contention and allows the data to be more readily available in the newly created partitions:

```
$ python partition_data.py -c host=localhost -p partman_test.id_taptest_table -t id -i 1
Attempting to turn off autovacuum for partition set...
... Success!
Rows moved: 100
Rows moved: 100
...
Rows moved: 99
...
Rows moved: 100
Rows moved: 1
Total rows moved: 100000
Running vacuum analyze on parent table...
Attempting to reset autovacuum for old parent table and all child tables...
    ... Success!
```

Partitioning the data like this has also made the partitions that were needed to store the data

```
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' ORDER BY
        tablename
--------------------------
 id_taptest_table
 id_taptest_table_p0
 id_taptest_table_p10000
 id_taptest_table_p100000
 id_taptest_table_p110000
 id_taptest_table_p120000
 id_taptest_table_p20000
 id_taptest_table_p30000
 id_taptest_table_p40000
 id_taptest_table_p50000
 id_taptest_table_p60000
 id_taptest_table_p70000
 id_taptest_table_p80000
 id_taptest_table_p90000
(14 rows)
```

Now create the sub-partitions for 1000. As was noted above for time, we give the parent table who's children we want partitioned along with the properties to give those children:

```
keith@keith=# SELECT partman.create_sub_parent('partman_test.id_taptest_table', 'col1',
 create_sub_parent
-------------------
 t
(1 row)
```

All children tables get at least their minimum sub-partition made and the sub-partitions based around the current max value are also created.

```
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' ORDER BY
            tablename
-----------------------------------
 id_taptest_table
 id_taptest_table_p0
 id_taptest_table_p0_p0
 id_taptest_table_p10000
 id_taptest_table_p100000
 id_taptest_table_p100000_p100000
 id_taptest_table_p100000_p101000
 id_taptest_table_p100000_p102000
 id_taptest_table_p10000_p10000
 id_taptest_table_p110000
 id_taptest_table_p110000_p110000
 id_taptest_table_p120000
 id_taptest_table_p120000_p120000
 id_taptest_table_p20000
 id_taptest_table_p20000_p20000
 id_taptest_table_p30000
 id_taptest_table_p30000_p30000
 id_taptest_table_p40000
 id_taptest_table_p40000_p40000
 id_taptest_table_p50000
 id_taptest_table_p50000_p50000
 id_taptest_table_p60000
 id_taptest_table_p60000_p60000
 id_taptest_table_p70000
 id_taptest_table_p70000_p70000
 id_taptest_table_p80000
 id_taptest_table_p80000_p80000
 id_taptest_table_p90000
 id_taptest_table_p90000_p98000
 id_taptest_table_p90000_p99000
(30 rows)
```

If you're wondering why, even with data in them, the children didn't get all their sub-partitions created, it's for the same reason that the top partition only initially had the 2 previous and 2 after created: the data still exists in the sub-partition parents. You can see this by running the monitoring function built into pg_partman here:

```
keith@keith=# SELECT * FROM partman.check_parent() ORDER BY 1;
             parent_table              | count
---------------------------------------+-------
 partman_test.id_taptest_table_p0      |  9999
 partman_test.id_taptest_table_p10000  | 10000
 partman_test.id_taptest_table_p100000 |     1
 partman_test.id_taptest_table_p20000  | 10000
 partman_test.id_taptest_table_p30000  | 10000
 partman_test.id_taptest_table_p40000  | 10000
 partman_test.id_taptest_table_p50000  | 10000
 partman_test.id_taptest_table_p60000  | 10000
 partman_test.id_taptest_table_p70000  | 10000
 partman_test.id_taptest_table_p80000  | 10000
 partman_test.id_taptest_table_p90000  | 10000
(11 rows)
```

So, lets fix that:

```
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p0 -t id -i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p10000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p20000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p30000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p40000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p50000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p60000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p70000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p80000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p90000 -t i
python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p100000 -t
```

Now the monitoring function returns nothing (as should be the norm):

```
keith@keith=# SELECT * FROM partman.check_parent() ORDER BY 1;
 parent_table | count
--------------+-------
(0 rows)
```

Now we also see all child partitions were created for the data that exists:

```
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' ORDER B
            tablename
----------------------------------
 id_taptest_table
 id_taptest_table_p0
 id_taptest_table_p0_p0
 id_taptest_table_p0_p1000
 id_taptest_table_p0_p2000
```

```
id_taptest_table_p0_p3000
id_taptest_table_p0_p4000
id_taptest_table_p0_p5000
id_taptest_table_p0_p6000
id_taptest_table_p0_p7000
id_taptest_table_p0_p8000
id_taptest_table_p0_p9000
id_taptest_table_p10000
id_taptest_table_p100000
id_taptest_table_p100000_p100000
id_taptest_table_p100000_p101000
id_taptest_table_p100000_p102000
id_taptest_table_p10000_p10000
id_taptest_table_p10000_p11000
id_taptest_table_p10000_p12000
id_taptest_table_p10000_p13000
id_taptest_table_p10000_p14000
id_taptest_table_p10000_p15000
id_taptest_table_p10000_p16000
id_taptest_table_p10000_p17000
id_taptest_table_p10000_p18000
id_taptest_table_p10000_p19000
id_taptest_table_p110000
id_taptest_table_p110000_p110000
id_taptest_table_p120000
id_taptest_table_p120000_p120000
id_taptest_table_p20000
id_taptest_table_p20000_p20000
id_taptest_table_p20000_p21000
id_taptest_table_p20000_p22000
id_taptest_table_p20000_p23000
id_taptest_table_p20000_p24000
id_taptest_table_p20000_p25000
id_taptest_table_p20000_p26000
id_taptest_table_p20000_p27000
id_taptest_table_p20000_p28000
id_taptest_table_p20000_p29000
id_taptest_table_p30000
id_taptest_table_p30000_p30000
id_taptest_table_p30000_p31000
id_taptest_table_p30000_p32000
id_taptest_table_p30000_p33000
id_taptest_table_p30000_p34000
id_taptest_table_p30000_p35000
id_taptest_table_p30000_p36000
id_taptest_table_p30000_p37000
```

```
id_taptest_table_p30000_p38000
id_taptest_table_p30000_p39000
id_taptest_table_p40000
id_taptest_table_p40000_p40000
id_taptest_table_p40000_p41000
id_taptest_table_p40000_p42000
id_taptest_table_p40000_p43000
id_taptest_table_p40000_p44000
id_taptest_table_p40000_p45000
id_taptest_table_p40000_p46000
id_taptest_table_p40000_p47000
id_taptest_table_p40000_p48000
id_taptest_table_p40000_p49000
id_taptest_table_p50000
id_taptest_table_p50000_p50000
id_taptest_table_p50000_p51000
id_taptest_table_p50000_p52000
id_taptest_table_p50000_p53000
id_taptest_table_p50000_p54000
id_taptest_table_p50000_p55000
id_taptest_table_p50000_p56000
id_taptest_table_p50000_p57000
id_taptest_table_p50000_p58000
id_taptest_table_p50000_p59000
id_taptest_table_p60000
id_taptest_table_p60000_p60000
id_taptest_table_p60000_p61000
id_taptest_table_p60000_p62000
id_taptest_table_p60000_p63000
id_taptest_table_p60000_p64000
id_taptest_table_p60000_p65000
id_taptest_table_p60000_p66000
id_taptest_table_p60000_p67000
id_taptest_table_p60000_p68000
id_taptest_table_p60000_p69000
id_taptest_table_p70000
id_taptest_table_p70000_p70000
id_taptest_table_p70000_p71000
id_taptest_table_p70000_p72000
id_taptest_table_p70000_p73000
id_taptest_table_p70000_p74000
id_taptest_table_p70000_p75000
id_taptest_table_p70000_p76000
id_taptest_table_p70000_p77000
id_taptest_table_p70000_p78000
id_taptest_table_p70000_p79000
```

```
 id_taptest_table_p80000
 id_taptest_table_p80000_p80000
 id_taptest_table_p80000_p81000
 id_taptest_table_p80000_p82000
 id_taptest_table_p80000_p83000
 id_taptest_table_p80000_p84000
 id_taptest_table_p80000_p85000
 id_taptest_table_p80000_p86000
 id_taptest_table_p80000_p87000
 id_taptest_table_p80000_p88000
 id_taptest_table_p80000_p89000
 id_taptest_table_p90000
 id_taptest_table_p90000_p90000
 id_taptest_table_p90000_p91000
 id_taptest_table_p90000_p92000
 id_taptest_table_p90000_p93000
 id_taptest_table_p90000_p94000
 id_taptest_table_p90000_p95000
 id_taptest_table_p90000_p96000
 id_taptest_table_p90000_p97000
 id_taptest_table_p90000_p98000
 id_taptest_table_p90000_p99000
(119 rows)
```

We can still take this another level deeper as well. Normally with a large amount of data, it's not recommended to partition down to an interval this low since the benefit gained is minimal compared the management of such a large number of tables. But it's being done here as an example. Just as with the time example above, we now have to sub-partition each one of the sub-parent tables to say how we want their children sub-partitioned:

```
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p0', 'col1', 'partman',
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p10000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p20000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p30000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p40000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p50000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p60000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p70000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p80000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p90000', 'col1', 'partma
SELECT partman.create_sub_parent('partman_test.id_taptest_table_p100000', 'col1', 'partm
```

I won't show the full list here, but you can see how every child table of the above parents is now a parent table itself with the appropriate minimal child table created where needed as well as the child tables around the current max:

```
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' order by
```

```
                tablename
------------------------------------------
 id_taptest_table
 id_taptest_table_p0
 id_taptest_table_p0_p0
 id_taptest_table_p0_p0_p0
 id_taptest_table_p0_p1000
 id_taptest_table_p0_p1000_p1000
 id_taptest_table_p0_p2000
 id_taptest_table_p0_p2000_p2000
 ...
 id_taptest_table_p10000
 id_taptest_table_p100000
 id_taptest_table_p100000_p100000
 id_taptest_table_p100000_p100000_p100000
 id_taptest_table_p100000_p100000_p100100
 id_taptest_table_p100000_p100000_p100200
 id_taptest_table_p100000_p101000
 id_taptest_table_p100000_p101000_p101000
 id_taptest_table_p100000_p102000
 id_taptest_table_p100000_p102000_p102000
 id_taptest_table_p10000_p10000
 id_taptest_table_p10000_p10000_p10000
 id_taptest_table_p10000_p11000
 id_taptest_table_p10000_p11000_p11000
 ...
 id_taptest_table_p90000_p98000
 id_taptest_table_p90000_p98000_p98000
 id_taptest_table_p90000_p99000
 id_taptest_table_p90000_p99000_p99800
 id_taptest_table_p90000_p99000_p99900
(225 rows)
```

If you ran the check_parent() function, you'd see that now each one of these new parent tables now needs to have its data moved. Now's a good time show a trick for generating many individual statements based on values returned from a query:

```
SELECT 'python partition_data.py -c host=localhost -p '||parent_table||' -t id -i 100' F

                                                ?column?
----------------------------------------------------------------------------------------
 python partition_data.py -c host=localhost -p partman_test.id_taptest_table -t id -i 10
 python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p0 -t id -i
 python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p0_p0 -t id
 python partition_data.py -c host=localhost -p partman_test.id_taptest_table_p0_p1000 -t
...
```

This will generate the commands to partition out the data found in any parent table managed by pg_partman. Yes some are already empty, but that won't matter since they'll just do nothing and it makes the query to generate these commands easier. Recommend putting the output from this into an executable shell file vs just pasting it all into the shell directly. Now if you get a list of all the tables, you can see there's quite a lot now (the row count returned is the number of tables).

```
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' order by
                tablename
-----------------------------------------
 id_taptest_table
 id_taptest_table_p0
 id_taptest_table_p0_p0
 id_taptest_table_p0_p0_p0
 id_taptest_table_p0_p0_p100
 id_taptest_table_p0_p0_p200
 id_taptest_table_p0_p0_p300
 id_taptest_table_p0_p0_p400
 id_taptest_table_p0_p0_p500
 id_taptest_table_p0_p0_p600
 id_taptest_table_p0_p0_p700
 id_taptest_table_p0_p0_p800
 id_taptest_table_p0_p0_p900
 id_taptest_table_p0_p1000
 id_taptest_table_p0_p1000_p1000
 id_taptest_table_p0_p1000_p1100
 id_taptest_table_p0_p1000_p1200
 id_taptest_table_p0_p1000_p1300
 id_taptest_table_p0_p1000_p1400
 id_taptest_table_p0_p1000_p1500
 id_taptest_table_p0_p1000_p1600
 id_taptest_table_p0_p1000_p1700
 id_taptest_table_p0_p1000_p1800
 id_taptest_table_p0_p1000_p1900
 id_taptest_table_p0_p2000
 id_taptest_table_p0_p2000_p2000
 id_taptest_table_p0_p2000_p2100
 ...
 id_taptest_table_p90000_p98000_p98800
 id_taptest_table_p90000_p98000_p98900
 id_taptest_table_p90000_p99000
 id_taptest_table_p90000_p99000_p99000
 id_taptest_table_p90000_p99000_p99100
 id_taptest_table_p90000_p99000_p99200
 id_taptest_table_p90000_p99000_p99300
 id_taptest_table_p90000_p99000_p99400
 id_taptest_table_p90000_p99000_p99500
 id_taptest_table_p90000_p99000_p99600
 id_taptest_table_p90000_p99000_p99700
 id_taptest_table_p90000_p99000_p99800
 id_taptest_table_p90000_p99000_p99900
(1124 rows)
```

Now all 100,000 rows are properly partitioned where they should be and any new rows should go where they're supposed to.

Set run_maintenance() to run often enough

Using the above time-based partitions, run_maintenance() should be called at least twice a day to ensure it keeps up with the requirements of the smallest time partition interval (daily).

For serial based partitioning, you must know your data ingestion rate and call it often enough to keep new partitions created ahead of that rate.

If you're using the Background Worker (BGW), set the pg_partman_bgw.interval value in postgresql.conf. This example sets it to run every 12 hrs (43200 seconds). See the doc/pg_partman.md file for more information on the BGW settings.

```
pg_partman_bgw.interval = 43200
pg_partman_bgw.role = 'keith'
pg_partman_bgw.dbname = 'keith'
```

If you're not using the BGW, you must use a third-party scheduling tool like cron to schedule the calls to run_maintenance()

```
03 01,13 * * * psql -c "SELECT run_maintenance()"
```

Use Retention Policy

To drop partitions on the first example above that are older than 30 days, set the following:

```
UPDATE part_config SET retention = '30 days', retention_keep_table = false WHERE parent_
```

To drop partitions on the second example above that contain a value 100 less than the current max (max(col1) - 100), set the following:

```
UPDATE part_config SET retention = '100', retention_keep_table = false WHERE parent_tabl
```

For example, once the current id value of col1 reaches 1000, all partitions with values less than 900 will be dropped.

If you'd like to keep the old data stored offline in dump files, set the retention_schema column as well (the keep* config options will be overridden if this is set):

```
UPDATE part_config SET retention = ’30 days’, retention_schema = ’archive’ WHERE parent_
```

Then use the included python script **dump_partition.py** to dump out all tables contained in the archive schema:

```
$ python dump_partition.py -c "host=localhost username=postgres" -d mydatabase -n archiv
```

To implement any retention policy, just ensure run_maintenance() is called often enough for your needs. That function handles both partition creation and the retention policies.

Undo Partitioning: Simple Time Based

As with partitioning data out, it’s best to use the python script to undo partitioning as well to avoid contention and moving large amounts of data in a single transaction. Except for the final example, there’s no data in these partition sets, but the example would work either way. This also shows how you can give time-based partition sets a lower interval than what they are partitioned at. This set was daily above, but the batches are committed at the hourly marks (if there was data).

```
$ python undo_partition.py -p partman_test.time_taptest_table -c host=localhost -t time
Attempting to turn off autovacuum for partition set...
    ... Success!
Total rows moved: 0
Running vacuum analyze on parent table...
Attempting to reset autovacuum for old parent table...
    ... Success!
```

Undo Partitioning: Simple Serial ID

This just undoes the id partitions committing at the default partition interval of 10 given above.

```
$ python undo_partition.py -p partman_test.id_taptest_table -c host=localhost -t id
Attempting to turn off autovacuum for partition set...
    ... Success!
Total rows moved: 0
Running vacuum analyze on parent table...
Attempting to reset autovacuum for old parent table...
    ... Success!
```

Undo Partitioning: Sub-partition ID->ID->ID

Undoing sub-partitioning involves a little more work (or possibly a lot if it’s a large set). You have to start from the bottom up. Just as I did above for generating statements for partitioning the data out, I can do the same for the undo_partition.py script. Keep in mind this gets the undo statement for ALL the parents at once. You do have to go through and parse out the top level calls as well as the mid-level partition, but this at least saves you a lot of potential typing (and typos). The bottom partitions must all be done first and the top last. Also, in this case I have no intention of keeping the old, empty tables anymore, so the –droptable option is given. pg_partman tries to be as safe as possible, so it only uninherits tables by default when undoing partitioning. If you want something dropped, you have to be sure and tell it.

```
SELECT 'python undo_partition.py -c host=localhost -p '||parent_table||' -t id -i 100 --
```

First do the lowest level sub-partitions:

```
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p0_p0 -t id
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p0_p1000 -t
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p0_p2000 -t
...
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p100000_p10
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p100000_p10
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p100000_p10
```

Next do what were the mid level sub-partitions:

```
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p0 -t id -i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p10000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p100000 -t
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p110000 -t
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p120000 -t
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p20000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p30000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p40000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p50000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p60000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p70000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p80000 -t i
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table_p90000 -t i
```

And finally do the last, top level partition:

```
python undo_partition.py -c host=localhost -p partman_test.id_taptest_table -t id -i 10
```

Now there is only one table left with all the data

```
keith@keith=# SELECT tablename FROM pg_tables WHERE schemaname = 'partman_test' ORDER B
    tablename
-----------------
 id_taptest_table

keith@keith=# SELECT count(*) FROM partman_test.id_taptest_table ;
 count
--------
 100000
```

```
(1 row)
```

Undo Partitioning: Sub-partition Time->Time->Time

This is done in the same exact way as for ID->ID->ID except the undo_partition.py script would use the -t time setting and -i would use a time interval value.

Hopefully these working examples can help you get started. Again, please see the `pg_partman.md` doc for the full details on all the functions and features of this extension. If you have any issues or questions, feel free to open an issue on the github page: https://github.com/pgpartman/pg_partman

This document is an aid for migrating an existing partitioned table set to using pg_partman. Please note that at this time, this guide is only for non-native, trigger-baed partitioning. Documentation for native partitioning is in the works, but it will be mostly focused on PostgreSQL 11 since 10 was very limited in its partitioning support. For now, the easiest way to migrate to a natively partitioned table is to create a brand new table and copy/move the data.

pg_partman does not support having child table names that do not match its naming convention. I've tried to implement that several times, but it's too difficult to support in a general manner and just ends up hindering development or breaking a feature. Your situation likely isn't exactly like the ones below, but this should at least provide guidance on what is required.

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 I will be using:

```
                                  Table "tracking.hits"
 Column |            Type             | Modifiers | Storage | Stats target | Description
--------+-----------------------------+-----------+---------+--------------+-------------
 id     | integer                     | not null  | plain   |              |
 start  | timestamp without time zone | not null  | plain   |              |
Child tables: tracking.hits20160103,
              tracking.hits20160110,
              tracking.hits20160117

insert into tracking.hits20160103 values (1, generate_series(’2016-01-03 01:00:00’::timestam
insert into tracking.hits20160110 values (1, generate_series(’2016-01-10 01:00:00’::timestam
insert into tracking.hits20160117 values (1, generate_series(’2016-01-17 01:00:00’::timestam

                                  Table "tracking.hits"
 Column |            Type             | Modifiers | Storage | Stats target | Description
--------+-----------------------------+-----------+---------+--------------+-------------
```

```
 id     | integer                     | not null  | plain   |              |
 start  | timestamp without time zone | not null  | plain   |              |
Child tables: tracking.hits1000,
              tracking.hits2000,
              tracking.hits3000

insert into tracking.hits1000 values (generate_series(1000,1999), now());
insert into tracking.hits2000 values (generate_series(2000,2999), now());
insert into tracking.hits3000 values (generate_series(3000,3999), now());

                                  Table "tracking.hits"
 Column |            Type             | Modifiers | Storage | Stats target | Description
--------+-----------------------------+-----------+---------+--------------+-------------
 id     | integer                     | not null  | plain   |              |
 start  | timestamp without time zone | not null  | plain   |              |
Child tables: tracking.hits_aa,
              tracking.hits_bb,
              tracking.hits_cc

Data depends on partitioning type. See below.
```

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 and old & new triggers existing at the same time.

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". Even the custom time intervals use the same patterns. All of them are listed here for your reference.

```
_pYYYY               - Yearly (and any custom time greater than this)
_pYYYY"q"Q           - Quarterly (double quotes required to add another string value inside
_pYYYY_MM            - Monthly (and all custom time intervals between yearly and monthly)
_pIYYY"w"IW          - Weekly (ISO Year and ISO Week)
_pYYYY_MM_DD         - Daily (and all custom time intervals between monthly and daily)
_pYYYY_MM_DD_HH24MI  - Hourly, Half-Hourly, Quarter-Hourly (and all custom time intervals b
_pYYYY_MM_DD_HH24MISS - Only used with custom time if interval is less than 1 minute (cannot
_p#####              - Serial/ID partition has a suffix that is the value of the lowest pos
```

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". Looking at the list above, you can see the new weekly pattern that pg_partman uses.

A note about quarterly partitioning… the to_timestamp() function does not recognize the "Q" format string like to_char() does. Why? You can look in the postgres source and see the reasoning, but it makes no sense to me. I handle this inside the show_partition_info() function if you need a way to do this.

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.

```
select 'ALTER TABLE '||n.nspname||'.'||c.relname||' RENAME TO '||substring(c.relname from 1
        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::regclass = 'tracking.hits'::regclass
        order by c.relname;
```

Which outputs:

```
ALTER TABLE tracking.hits20160103 RENAME TO hits_p2015w53;
ALTER TABLE tracking.hits20160110 RENAME TO hits_p2016w01;
ALTER TABLE tracking.hits20160117 RENAME TO hits_p2016w02;
```

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

```
   tablename   | schemaname
---------------+------------
 hits          | tracking
 hits_p2015w53 | tracking
 hits_p2016w01 | tracking
 hits_p2016w02 | tracking
```

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. Since the number value can vary, if you didn’t have that as the final suffix of the partition name, that could make pattern matching for a rename more challanging. Using my second example table above, it would be something like this.

```
select ’ALTER TABLE ’||n.nspname||’.’||c.relname||’ RENAME TO ’||substring(c.relname from 1
    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::regclass = ’tracking.hits’::regclass
    order by c.relname;

ALTER TABLE tracking.hits1000 RENAME TO hits_p1000;
ALTER TABLE tracking.hits2000 RENAME TO hits_p2000;
ALTER TABLE tracking.hits3000 RENAME TO hits_p3000;

 tablename | schemaname
-----------+------------
 hits      | tracking
 hits1000  | tracking
 hits2000  | tracking
 hits3000  | tracking
```

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.

If this is partitioned by time, assume the following data exists in the child tables:

```
insert into tracking.hits_aa values (1, generate_series(’2016-01-03 01:00:00’::timestamptz,
insert into tracking.hits_bb values (2, generate_series(’2016-01-10 01:00:00’::timestamptz,
insert into tracking.hits_cc values (3, generate_series(’2016-01-17 01:00:00’::timestamptz,
```

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.

```
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::regclass = ’tracking.hits’::regclass
        ORDER BY c.relname
LOOP
    -- 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.
    v_min_val := date_trunc(’week’, v_min_val);

    -- Build the sql statement to rename the child table
    -- Use the appropriate date/time string from the list above for your interval
    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 4)||’_p’||to_char(v_min_val, ’IYYY"w"IW

    -- 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_aa RENAME TO hits_p2015w53
NOTICE:  ALTER TABLE tracking.hits_bb RENAME TO hits_p2016w01
NOTICE:  ALTER TABLE tracking.hits_cc RENAME TO hits_p2016w02
```

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 same example partition set used before:

```
insert into tracking.hits_aa values (generate_series(1100,1294), now());
insert into tracking.hits_bb values (generate_series(2400,2991), now());
insert into tracking.hits_cc 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::regclass = ’tracking.hits’::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 4)||’_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_aa RENAME TO hits_p1000
NOTICE:  ALTER TABLE tracking.hits_bb RENAME TO hits_p2000
NOTICE:  ALTER TABLE tracking.hits_cc RENAME TO hits_p3000
```

Step 4

Actual setup and trigger swap

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 highly recommend doing steps 4a and 4b in a single transaction just to avoid weird trigger conflicts.

Step 4a

Drop your current partitioning trigger after starting a transaction

```
BEGIN;
DROP TRIGGER myoldtrigger ON tracking.hits;
```

Step 4b

Setup pg_partman to manage your partition set.

```
SELECT partman.create_parent(’tracking.hits’, ’start’, ’partman’, ’weekly’);
COMMIT;
```

This single function call will add your old partition set into pg_partman’s configuration, create a new trigger 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
                                  Table "tracking.hits"
 Column |            Type             | Modifiers | Storage | Stats target | Description
--------+-----------------------------+-----------+---------+--------------+-------------
 id     | integer                     | not null  | plain   |              |
 start  | timestamp without time zone | not null  | plain   |              |
Triggers:
    hits_part_trig BEFORE INSERT ON tracking.hits FOR EACH ROW EXECUTE PROCEDURE tracking.hi
Child tables: tracking.hits_p2015w53,
              tracking.hits_p2016w01,
              tracking.hits_p2016w02,
              tracking.hits_p2016w03,
              tracking.hits_p2016w04,
              tracking.hits_p2016w05,
              tracking.hits_p2016w06,
              tracking.hits_p2016w07,
              tracking.hits_p2016w08,
              tracking.hits_p2016w09
```

Note that I ran this create_parent() function on Feb 6th, 2016. This is the 5th week of the year. 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. Some of those tables already existed and, since their naming pattern matched pg_partman's, it handled that just fine.

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.

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