The State of Partitioning

Where Partitioning Has Been
Where It Is
Where It's Going

Keith Fiske
http://www.keithf4.com
http://www.crunchydata.com

crunchy data

Who Am I

- Senior Database Engineer at Crunchy Data
- Working with PostgreSQL since 8.3
- Author of several popular third party PostgreSQL extensions including
  - pg_partman - https://github.com/pgpartman/pg_partman
  - pgMonitor - https://github.com/CrunchyData/pgmonitor
  - pg_jobmon - https://github.com/omniti-labs/pg_jobmon
- Provide PostgreSQL training and develop solutions to make PostgreSQL easier to use

What is Partitioning?

- Organization of data into logical "chunks" or partitions
- Each partition is generally its own table
- Rules dictate where data goes and constrain data within a partition

Why Partition Tables?

- **Easier to manage data and space**

  Deletion of large amounts of data in PostgreSQL can be expensive and often does not return disk space to the OS. Dropping a table is quick and almost immediately returns disk space. Data retention is the primary reason for partitioning in PostgreSQL

- **Improves table maintenance**

  The VACUUM process in PostgreSQL grows in expense as table size grows. Smaller tables are easier for VACUUM to manage and can potentially be skipped

- **Query Performance**

  As tables grow in size, read and write performance may be impacted. On extremely large tables, partition pruning in the query plan can be a noticeable benefit. Avoids larger index & tables scans.

The Old Way

- Table Inheritance
  - Child tables that inherit their properties from a parent table
- Triggers
  - Triggers on the parent that route the data to the proper child
- Constraints
  - Constraints on the child tables that limit data that can exist inside them
- All this had to be manually managed (or custom automation written) and was extremely inefficient outside of retention management.
- May still be needed in some very narrow use-cases

The New Way

- Declarative Partitioning (aka native)
- SQL syntax commands
- Range, List, & Hash
- Internal tuple routing and partition pruning are far more efficient than triggers and constraint exclusion

Range Partitioning

- Partitioned into ranges by one or more columns with no overlap between partitions. Ex: Time/Integer

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);

CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

CREATE TABLE measurement_y2006m03 PARTITION OF measurement
    FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

=# \d+ measurement
                                Table "public.measurement"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 city_id   | integer |           | not null |         | plain   |              |
 logdate   | date    |           | not null |         | plain   |              |
 peaktemp  | integer |           |          |         | plain   |              |
 unitsales | integer |           |          |         | plain   |              |
Partition key: RANGE (logdate)
Partitions: measurement_y2006m02 FOR VALUES FROM ('2006-02-01') TO ('2006-03-01'),
            measurement_y2006m03 FOR VALUES FROM ('2006-03-01') TO ('2006-04-01')
```

List Partitioning

- Partitioned by explicitly listing which key value(s) appear(s) in each partition

```
CREATE TABLE cities (
    city_id         bigserial not null,
    name         text not null,
    population   int
) PARTITION BY LIST (initcap(name));

CREATE TABLE cities_west
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('Los Angeles', 'San Francisco');

=# \d+ cities
                                                  Table "public.cities"
   Column    |  Type   | Collation | Nullable |                 Default                 | Storage  | Stats target |
Description
------------+---------+-----------+----------+-----------------------------------------+----------+--------------+---
----------
 city_id    | bigint  |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |              |
 name       | text    |           | not null |                                         | extended |              |
 population | integer |           |          |                                         | plain    |              |
Partition key: LIST (initcap(name))
Partitions: cities_west FOR VALUES IN ('Los Angeles', 'San Francisco')
```

Hash Partitioning

- Used when you want to partition a randomized, growing data set evenly or don't know data distribution in advance

```
CREATE TABLE users (
    username    text         not null,
    password    text,
    created_on  timestamptz  not null default now(),
    id_admin    bool         not null default false
) PARTITION BY HASH (username);
```

- MODULUS is the number of partitions, and REMAINDER is a number, 0 or more, but less than MODULUS.

```
CREATE TABLE users_p0 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 0);
CREATE TABLE users_p1 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 1);
CREATE TABLE users_p2 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 2);
CREATE TABLE users_p3 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 3);
CREATE TABLE users_p4 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 4);
CREATE TABLE users_p5 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 5);
CREATE TABLE users_p6 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 6);
CREATE TABLE users_p7 PARTITION OF users ( primary key (username) ) FOR VALUES WITH (MODULUS 8, REMAINDER 7);
```

Hash Partitioning

```
\d+ users
                                          Table "public.users"
   Column   |           Type           | Collation | Nullable | Default | Storage  | Stats target | Description
------------+--------------------------+-----------+----------+---------+----------+--------------+-------------
 username   | text                     |           | not null |         | extended |              |
 password   | text                     |           |          |         | extended |              |
 created_on | timestamp with time zone |           | not null | now()   | plain    |              |
 id_admin   | boolean                  |           | not null | false   | plain    |              |
Partition key: HASH (username)
Partitions: users_p0 FOR VALUES WITH (modulus 8, remainder 0),
            users_p1 FOR VALUES WITH (modulus 8, remainder 1),
            users_p2 FOR VALUES WITH (modulus 8, remainder 2),
            users_p3 FOR VALUES WITH (modulus 8, remainder 3),
            users_p4 FOR VALUES WITH (modulus 8, remainder 4),
            users_p5 FOR VALUES WITH (modulus 8, remainder 5),
            users_p6 FOR VALUES WITH (modulus 8, remainder 6),
            users_p7 FOR VALUES WITH (modulus 8, remainder 7)
```

```
\d+ users_p1
                                        Table "public.users_p1"
   Column   |           Type           | Collation | Nullable | Default | Storage  | Stats target | Description
------------+--------------------------+-----------+----------+---------+----------+--------------+-------------
 username   | text                     |           | not null |         | extended |              |
 password   | text                     |           |          |         | extended |              |
 created_on | timestamp with time zone |           | not null | now()   | plain    |              |
 id_admin   | boolean                  |           | not null | false   | plain    |              |
Partition of: users FOR VALUES WITH (modulus 8, remainder 1)
Partition constraint: satisfies_hash_partition('1161847'::oid, 8, 1, username)
Indexes:
    "users_p1_pkey" PRIMARY KEY, btree (username)
```

Hash Partitioning

```
\copy users (username) from stdin;
proffers
babbles
cents
choose
chalked
redoubts
pitting
coddling
relieves
wooing
codgers
sinewy
separate
ferry
crusty
cursing
hawkers
deducted
gaseous
voyagers
\.
```

Hash Partitioning

```
SELECT tableoid::regclass as partition_name, count(*) FROM users GROUP BY 1 ORDER BY 1;

 partition_name | count
----------------+-------
 users_p0       |     2
 users_p1       |     5
 users_p2       |     1
 users_p3       |     3
 users_p4       |     2
 users_p5       |     3
 users_p6       |     3
 users_p7       |     1
(8 rows)
```

- If you can identify a column to partition data by, range or list are much better than hash long term
- Unable to add/remove child tables without recreating entire partition set
- Data often becomes unbalanced unless it is actually random.
- Even UUIDs can end up unbalanced. Look into UUID7/ULID (sortable, time-based UUID)

A Note About Identity

- SQL standard for managing table sequences
- Better handling of sequence permissions when tied to a table
- Better enforcement of only allowing sequence use for column values
- Easier to remove sequences from a table
- *Only supported properly with declarative partitioning*
- *Only works when entering data through the parent table*

```
CREATE TABLE new_table (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    data text
);
```

Updating Partitioned Data

- Support added in PG11
- When an UPDATE causes a row to no longer match the partition constraint, PG will try to move it to a different partition where it does match the partition constraint
- Same as normal updates, behind the scenes does a DELETE/INSERT, but likely more expensive since it's between tables.
- Limited UPSERT support (INSERT … ON CONFLICT …)
    - DO UPDATE works if there's a matching unique constraint with the partition key

Default Partition

- Added in PG11
- Handle partition values that do not have a defined child
- Anti-constraint of all existing children, updated when child added or removed
- Cannot add a new child table if that child's constraint matches data in default. Must move data out first.
- Leaving data in DEFAULT can have massive performance penalties for both queries and DDL
    - Adding a new child causes scan of entire default to see if any data matches new constraint

```
ALTER TABLE [parent_table] ATTACH PARTITION [partition_name] DEFAULT;
```

Partition Pruning/Constraint Exclusion

- Running a query with a condition that does NOT include partition column

```
=# EXPLAIN ANALYZE SELECT * FROM measurement WHERE city_id < 5;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=8.21..223.75 rows=4184 width=24) (actual time=0.021..0.051 rows=4 loops=1)
   ->  Bitmap Heap Scan on measurement_20060201  (cost=8.21..24.74 rows=523 width=24) (actual time=0.020..0.021
rows=4 loops=1)
         Recheck Cond: (city_id < 5)
         Heap Blocks: exact=1
         ->  Bitmap Index Scan on measurement_20060201_pkey  (cost=0.00..8.07 rows=523 width=0) (actual
time=0.013..0.013 rows=4 loops=1)
               Index Cond: (city_id < 5)
   ->  Bitmap Heap Scan on measurement_20060202  (cost=8.21..24.74 rows=523 width=24) (actual time=0.003..0.003
rows=0 loops=1)
         Recheck Cond: (city_id < 5)
         ->  Bitmap Index Scan on measurement_20060202_pkey  (cost=0.00..8.07 rows=523 width=0) (actual
time=0.002..0.002 rows=0 loops=1)
               Index Cond: (city_id < 5)
[...]
         ->  Bitmap Index Scan on measurement_20060207_pkey  (cost=0.00..8.07 rows=523 width=0) (actual
time=0.001..0.001 rows=0 loops=1)
               Index Cond: (city_id < 5)
   ->  Seq Scan on measurement_default  (cost=0.00..29.62 rows=523 width=24) (actual time=0.007..0.007 rows=0
loops=1)
 Planning Time: 0.354 ms
 Execution Time: 0.168 ms
(34 rows)
```

Partition Pruning/Constraint Exclusion

- Running a query with a condition that DOES include partition column

```
=# EXPLAIN ANALYZE SELECT * FROM measurement WHERE logtime < '2006-02-04'::date;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..257.92 rows=4184 width=24) (actual time=0.018..0.053 rows=72 loops=1)
   Subplans Removed: 4
   ->  Seq Scan on measurement_20060201  (cost=0.00..29.62 rows=523 width=24) (actual time=0.018..0.027 rows=24 loops=1)
         Filter: (logtime < '2006-02-04'::date)
   ->  Seq Scan on measurement_20060202  (cost=0.00..29.62 rows=523 width=24) (actual time=0.006..0.010 rows=24 loops=1)
         Filter: (logtime < '2006-02-04'::date)
   ->  Seq Scan on measurement_20060203  (cost=0.00..29.62 rows=523 width=24) (actual time=0.004..0.008 rows=24 loops=1)
         Filter: (logtime < '2006-02-04'::date)
   ->  Seq Scan on measurement_default  (cost=0.00..29.62 rows=523 width=24) (actual time=0.002..0.002 rows=0 loops=1)
         Filter: (logtime < '2006-02-04'::date)
 Planning Time: 2.748 ms
 Execution Time: 0.118 ms
(12 rows)
```

Coming Soon™

- Improved query performance for partition sets with many tables.
  - Patch in current commitfest
- Global Indexes
  - Work has slowly been ongoing for a while, even before partitioning
  - Many discussions on hackers list about it

PostgreSQL Partition Manager (pg_partman)

- Originally created to better manage "the old way" when 9.1 introduced the extension system
- Declarative now manages triggers, constraints, & inheritance
- So is partman still needed?
- Many other things to manage and consider outside of child table creation

https://github.com/pgpartman/pg_partman

Still need pg_partman?

- Easily installed as an Extension
- Pre-creates child tables to avoid contention
  - Declarative does not automatically create child tables
  - Creating on demand can cause transaction backlog
- Currently used for time & integer/id based partitioning
  - New child tables needed indefinitely
  - Most other situations are a one-time setup
  - Version 5.1 will support LIST partitioning for single id values
- Retention management
  - Automatically detach/drop old tables based on configured intervals
  - Convenience script to help retain old tables as dump files
- Automatically creates default table (if desired)

Additional partman features

- Many options can be overwhelming. Likely only need a few.
- Background Worker to handle maintenance without third-party scheduler
- More easily partition existing table
  - Online & offline partitioning options depending on situation
- Handle naming length limits
  - 63 byte limit on all object names. PG truncates longer names
  - Partition suffix often indicates child property. Truncation could cut that off.
  - partman truncates the base table name and then adds suffix
  - Tip: Keep partition names as short as possible, especially with ID-based partitioning
- Non-partition column constraint exclusion
  - If old data is unchanging, creates a constraint based on existing data
  - Allows query performance optimizations outside the partition column

Additional partman features

- Sub-partitioning support
  - Negligible performance gains outside of VERY large tables (multi-terabyte)
  - May even cause performance degradation
  - Data always lives at lowest level
  - Some business logic requires additional separation of data
- Monitoring
  - pg_jobmon extension
    - Create alerting based around errors encountered during maintenance
    - Can be used to provide step-based logging inside any function without rolled back transactions undoing the logging within the database
  - Version 5.1 adds config column with last successful runtime per partition set
- Version 5 dropped trigger-based partitioning support

Current Issues In Core

- No Global Index
  - Cannot create a unique index on the parent that does not include the partition column(s)
- Unlogged is not properly inherited
  - Running ALTER TABLE to set OR unset unlogged property on parent does nothing in catalog and therefore inherits nothing to children
  - Because it changes nothing in catalog, you cannot change unlogged status of parent
- Dropping child tables with foreign keys TO the partition set
  - If DROP … CASCADE is run on a child table, drops the entire FK relation for the entire set
  - Must clean out all FK related data first before non-cascade drop can be done
- Relation options not inherited from parent (privileges, autovac, etc)
- Replica Identity not inherited from parent

Workarounds w/ partman!

- Apply property to partman's template table
  - Non-partition column primary keys, unique indexes & unique index tablespaces
    - Only enforced on at individual child table level
  - Relation-specific options (autovac, storage, etc)
  - Unlogged status
- Privileges from parent
  - Non-inheritance likely intentional
  - Flag in partman can do this to allow direct access to child tables
  - Direct access bypasses tuple routing and partition pruning bottlenecks
- Replica Identity from parent (upcoming version 5.1)

Partitioning in PostgreSQL

- Partitioning now a first-class feature in PostgreSQL
  - Versions 10 to 16 saw vast improvements following PG's iterative development process
- Primary reason to partition is data retention
- Recommend attempting query tuning before going straight to partitioning
  - You may see query performance reduced with partitioning vs examining the query plan and tuning the database or your queries
- Would prefer that pg_partman be made obsolete!

Thank you!

- These slides - http://slides.keithf4.com/state_of_partitioning.pdf
- PostgreSQL Home Page - postgresql.org
- Crunchy Data Solutions, Inc - crunchydata.com
- Planet PostgreSQL Community News Feed - planet.postgresql.org
- PostgreSQL Extension Network - pgxn.org