What's New in PostgreSQL 11

Kathy Gibbs
Sr. Specialist Solutions Architect,
Databases

Agenda

- Why would I want to go to PostgreSQL 11
- Partition Enhancements
- Parallel Enhancements
- Stored Procedures (Transaction Control)
- Adding New Table Columns with Default Values
- Command Line improvements
- Improved Statistics

PostgreSQL

Robust feature sets and extensions

Multi-Version Concurrency Control (MVCC), point in time recovery, granular access controls, tablespaces, asynchronous replication, nested transactions, online/hot backups, a refined query planner/optimizer, and write ahead logging

Supports international character sets, multi-byte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting

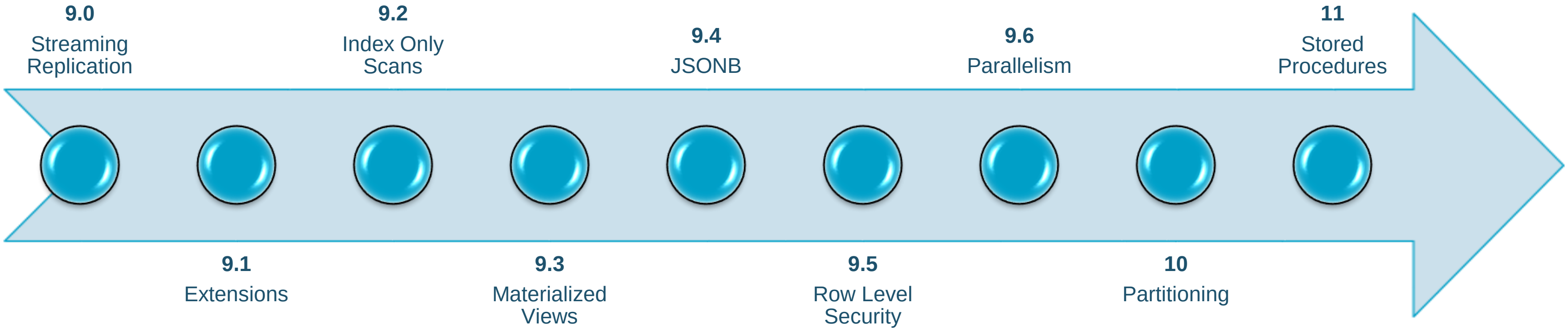
Reliable

High fault tolerance, ACID compliance, and full support for foreign keys, joins, views, triggers, and stored procedures

Standards-compliant

Includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. Supports storage of binary large objects, including pictures, sounds, or video

Evolution of PostgreSQL



9.0
Streaming Replication

9.1
Extensions

9.2
Index Only Scans

9.3
Materialized Views

9.4
JSONB

9.5
Row Level Security

9.6
Parallelism

10
Partitioning

11
Stored Procedures

Why go to PostgreSQL 11

PostgreSQL 11 provides users with improvements to overall performance of the database system, with specific enhancements associated with very large databases and high computational workloads.

"For PostgreSQL 11, our development community focused on adding features that improve PostgreSQL's ability to manage very large databases," said Bruce Momjian, a core team member of the PostgreSQL Global Development Group. "On top of PostgreSQL's proven performance for transactional workloads, PostgreSQL 11 makes it even easier for developers to run big data applications at scale."

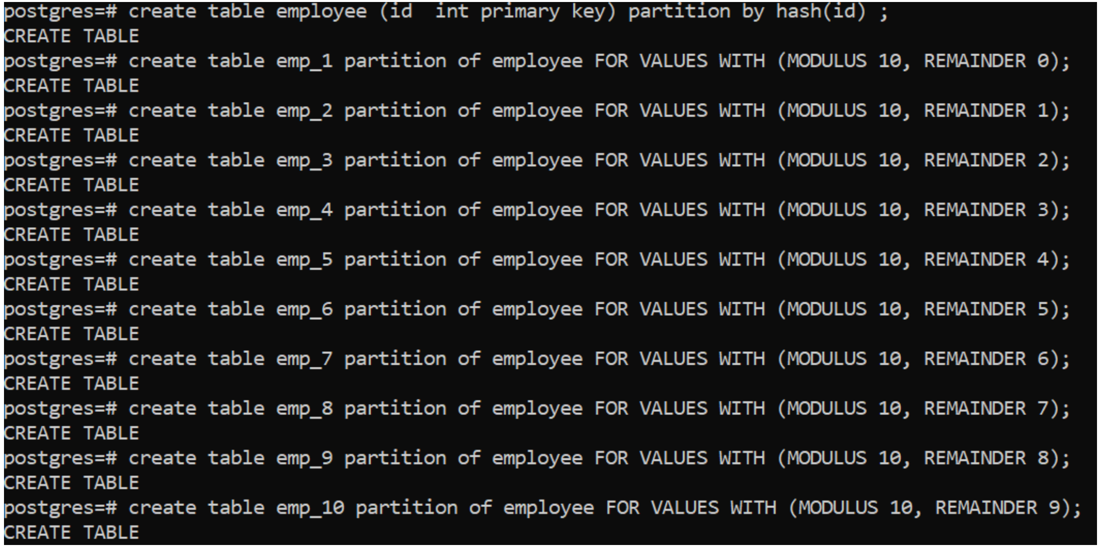
Partitioning Improvements

Partitioning Improvements

- Ability to partition by Hash Key
- Data Federation Improvements
- Partition Management Improvements
- Query Performance Improvements

Partitioning Improvements- Hash partitioning

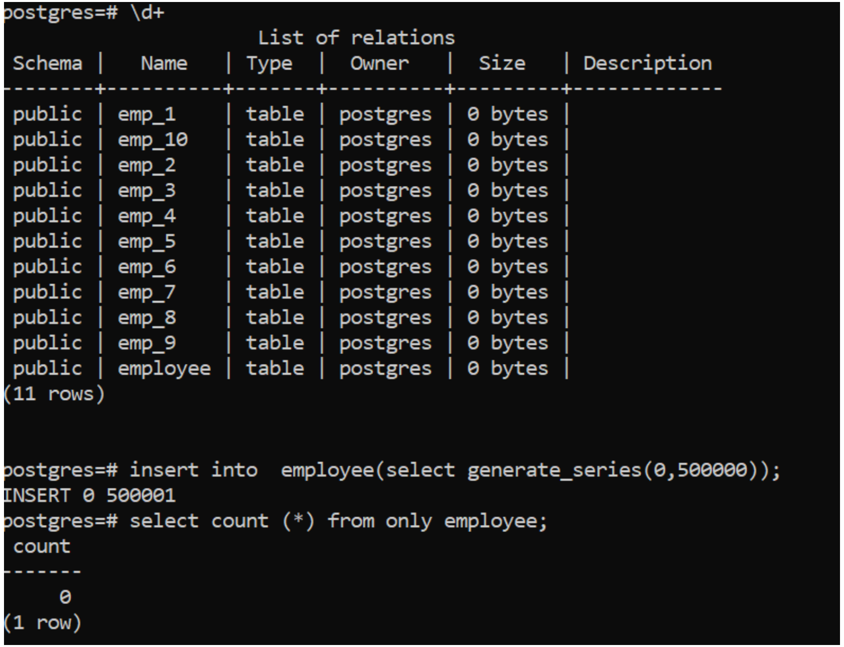
- Create partitions with MODULUS for the # of partitions



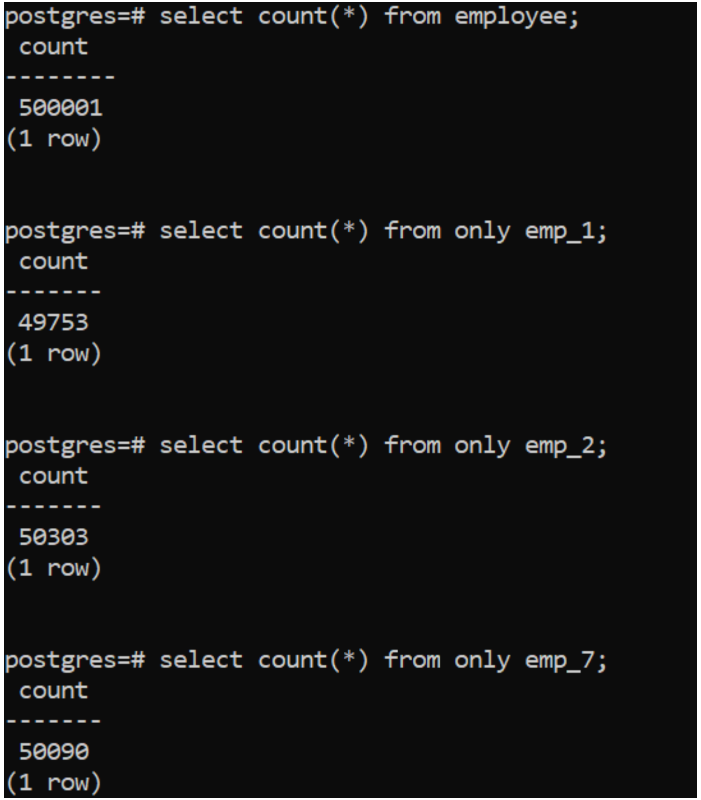
```
postgres=# create table employee (id  int primary key) partition by hash(id) ;
CREATE TABLE
postgres=# create table emp_1 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 0);
CREATE TABLE
postgres=# create table emp_2 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 1);
CREATE TABLE
postgres=# create table emp_3 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 2);
CREATE TABLE
postgres=# create table emp_4 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 3);
CREATE TABLE
postgres=# create table emp_5 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 4);
CREATE TABLE
postgres=# create table emp_6 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 5);
CREATE TABLE
postgres=# create table emp_7 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 6);
CREATE TABLE
postgres=# create table emp_8 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 7);
CREATE TABLE
postgres=# create table emp_9 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 8);
CREATE TABLE
postgres=# create table emp_10 partition of employee FOR VALUES WITH (MODULUS 10, REMAINDER 9);
CREATE TABLE
```

Partitioning Improvements- Hash partitioning

- Load Data



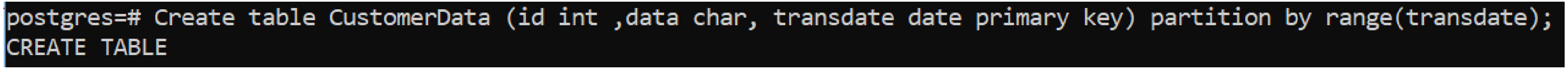
```
postgres=# \d+
                    List of relations
 Schema |   Name    | Type  |  Owner   |  Size   | Description
--------+-----------+-------+----------+---------+-------------
 public | emp_1     | table | postgres | 0 bytes |
 public | emp_10    | table | postgres | 0 bytes |
 public | emp_2     | table | postgres | 0 bytes |
 public | emp_3     | table | postgres | 0 bytes |
 public | emp_4     | table | postgres | 0 bytes |
 public | emp_5     | table | postgres | 0 bytes |
 public | emp_6     | table | postgres | 0 bytes |
 public | emp_7     | table | postgres | 0 bytes |
 public | emp_8     | table | postgres | 0 bytes |
 public | emp_9     | table | postgres | 0 bytes |
 public | employee  | table | postgres | 0 bytes |
(11 rows)


postgres=# insert into  employee(select generate_series(0,500000));
INSERT 0 500001
postgres=# select count (*) from only employee;
 count
-------
     0
(1 row)
```



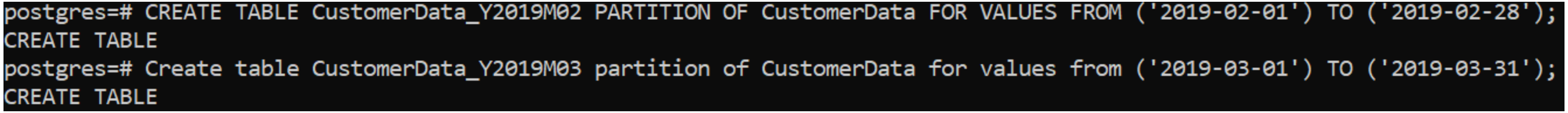
```
postgres=# select count(*) from employee;
 count
--------
 500001
(1 row)


postgres=# select count(*) from only emp_1;
 count
-------
 49753
(1 row)


postgres=# select count(*) from only emp_2;
 count
-------
 50303
(1 row)


postgres=# select count(*) from only emp_7;
 count
-------
 50090
(1 row)
```

Partitioning Improvements- Data Federation

- Native Sharding in 11 with postgres_fdw and partitions
- Create your partitioned table

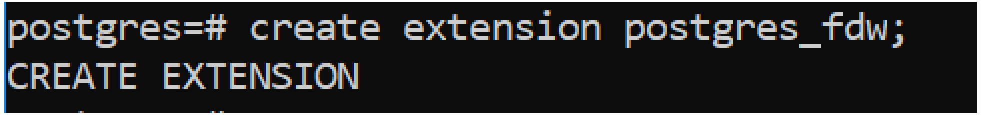


```
postgres=# Create table CustomerData (id int ,data char, transdate date primary key) partition by range(transdate);
CREATE TABLE
```



```
postgres=# CREATE TABLE CustomerData_Y2019M02 PARTITION OF CustomerData FOR VALUES FROM ('2019-02-01') TO ('2019-02-28');
CREATE TABLE
postgres=# Create table CustomerData_Y2019M03 partition of CustomerData for values from ('2019-03-01') TO ('2019-03-31');
CREATE TABLE
```

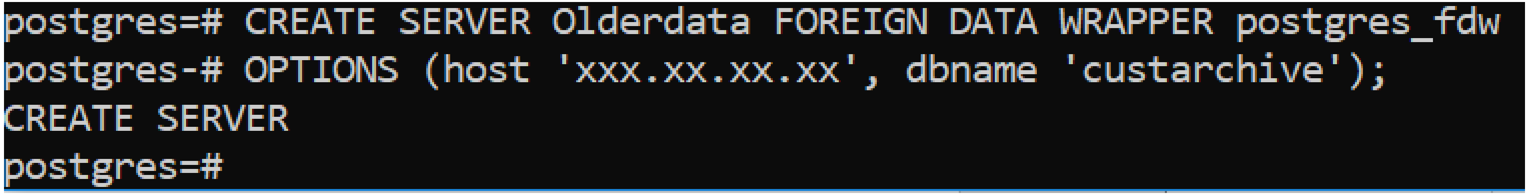
- Create the postgres_fdw in your instance



```
postgres=# create extension postgres_fdw;
CREATE EXTENSION
```

Partitioning Improvements- Data Federation

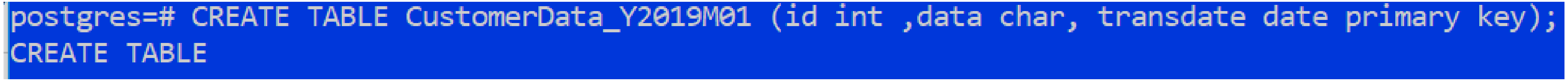
- Create the “foreign server” for your “Older data”



```
postgres=# CREATE SERVER Olderdata FOREIGN DATA WRAPPER postgres_fdw
postgres-# OPTIONS (host 'xxx.xx.xx.xx', dbname 'custarchive');
CREATE SERVER
postgres=#
```

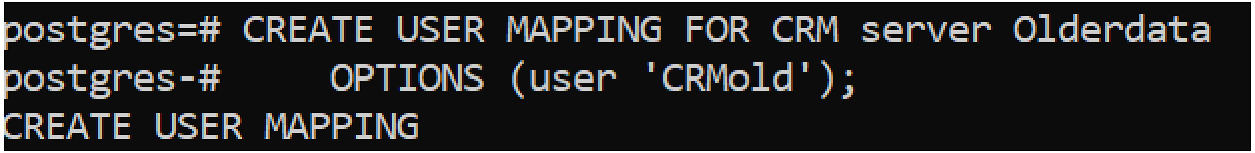
Partitioning Improvements- Data Federation

- On the “Olderdata” server, create the “partitioned” table



```
postgres=# CREATE TABLE CustomerData_Y2019M01 (id int ,data char, transdate date primary key);
CREATE TABLE
```

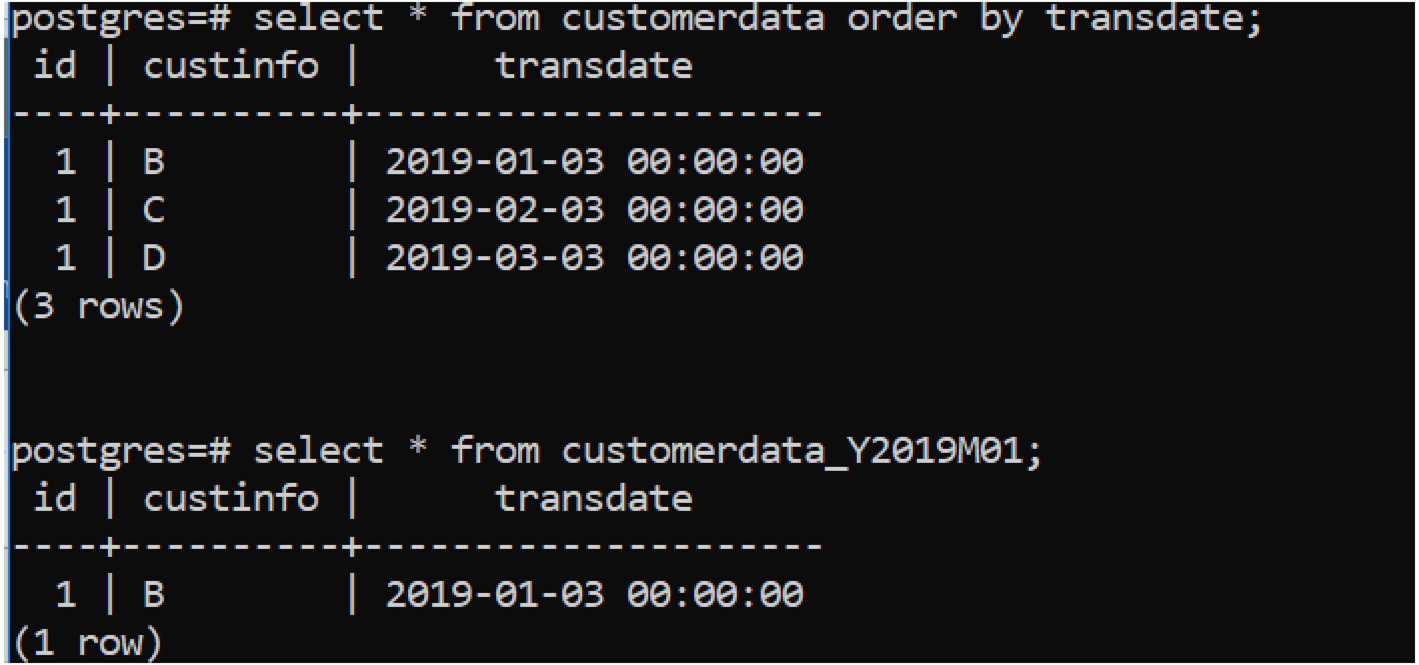
- Let’s also map our user “CRM” to the ”Olderdata” server user “CRMold”. This allows “CRM” to be “CRMold” when accessing remote tables



```
postgres=# CREATE USER MAPPING FOR CRM server Olderdata
postgres-#     OPTIONS (user 'CRMold');
CREATE USER MAPPING
```

Partitioning Improvements- Data Federation

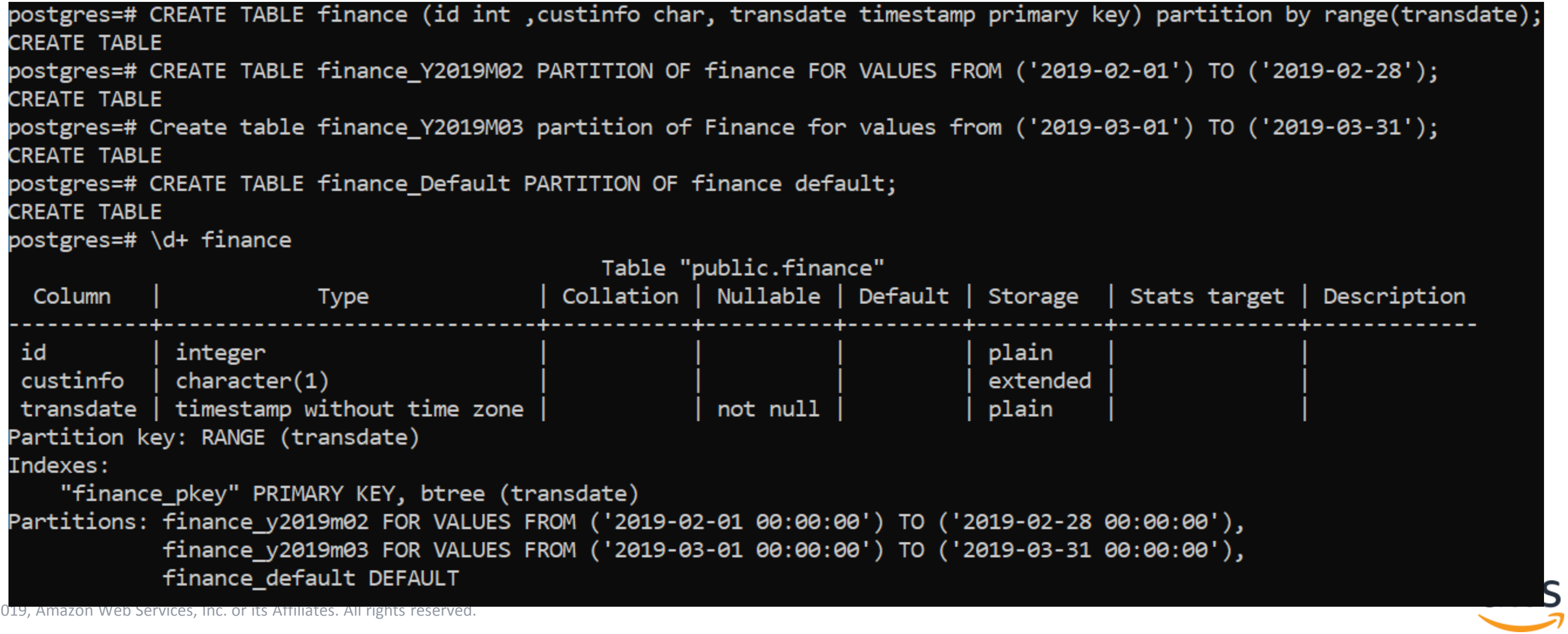
- Now you are ready to query



```
postgres=# select * from customerdata order by transdate;
 id | custinfo |      transdate
----+----------+----------------------
  1 | B        | 2019-01-03 00:00:00
  1 | C        | 2019-02-03 00:00:00
  1 | D        | 2019-03-03 00:00:00
(3 rows)


postgres=# select * from customerdata_Y2019M01;
 id | custinfo |      transdate
----+----------+----------------------
  1 | B        | 2019-01-03 00:00:00
(1 row)
```

Partitioning Improvements- Management

Default Partition



```
postgres=# CREATE TABLE finance (id int ,custinfo char, transdate timestamp primary key) partition by range(transdate);
CREATE TABLE
postgres=# CREATE TABLE finance_Y2019M02 PARTITION OF finance FOR VALUES FROM ('2019-02-01') TO ('2019-02-28');
CREATE TABLE
postgres=# Create table finance_Y2019M03 partition of Finance for values from ('2019-03-01') TO ('2019-03-31');
CREATE TABLE
postgres=# CREATE TABLE finance_Default PARTITION OF finance default;
CREATE TABLE
postgres=# \d+ finance
                                              Table "public.finance"
  Column   |            Type             | Collation | Nullable | Default | Storage  | Stats target | Description
-----------+-----------------------------+-----------+----------+---------+----------+--------------+-------------
 id        | integer                     |           |          |         | plain    |              |
 custinfo  | character(1)                |           |          |         | extended |              |
 transdate | timestamp without time zone |           | not null |         | plain    |              |
Partition key: RANGE (transdate)
Indexes:
    "finance_pkey" PRIMARY KEY, btree (transdate)
Partitions: finance_y2019m02 FOR VALUES FROM ('2019-02-01 00:00:00') TO ('2019-02-28 00:00:00'),
            finance_y2019m03 FOR VALUES FROM ('2019-03-01 00:00:00') TO ('2019-03-31 00:00:00'),
            finance_default DEFAULT
```

Partitioning Improvements- Management

- Ability to create primary keys, foreign keys, indexes, and AFTER triggers on partitioned tables that are passed down to all partitions
- Creating a row-level trigger will cause identical triggers in all existing partitions. Any new partitions too
- Check and Not Null constraints are always inherited in the partitions
- Using ONLY will result in an error when adding or dropping constraints on only the partitioned table

Partitioning Improvements- Query Performance

- 11 improves upon query performance when reading from partitions by using a new partition elimination strategy.
- Additionally, PostgreSQL 11 now supports the popular "upsert" feature on partitioned tables, which helps users to simplify application code and reduce network overhead when interacting with their data.
- When you want to perform bulk loads, you can load a table and then attach it as a new partition. You can do the same with detaching

Parallel Improvements

Parallel Improvements

- Parallelize BTREE index builds
- Parallelize Hash Joins
- General parallel Gains

Parallel Improvements - Parallelize BTREE index builds

- Generally a cost mode automatically determines how many worker processes should be requested
- Can set maintenance_work_mem which specifies the max memory that can be used by each index build operation
- Parallel index builds may benefit from increasing maintenance_work_mem where an equivalent serial index build will see little or no benefit

Parallel Improvements - Parallelize Hash Joins

- Currently the inner side is executed in full by every cooperating process to build identical copies of the hash table. This may be inefficient if the hash table is large or the plan is expensive.
- In a *parallel hash join*, the inner side is a *parallel hash* that divides the work of building a shared hash table over the cooperating processes.

Parallel Improvements- JIT Compilation

- The JIT expression compilation uses the LLVM project to boost the execution of expressions in WHERE clauses, target lists, aggregates, projections, as well as some other internal operations
- There are many claims of up to a 30% jump in performance. Long running queries that are CPU bound will benefit from JIT compilation
- PostgreSQL 11 it is off by default, for advanced users you can turn this parameter on.

Parallel Improvements – General Enhancements

- Parallel SELECT within Union even if the underlying queries can’t be parallelized
- Parallel Sequential Scan Gains
- Several data definition commands that either create tables or materialized views from queries are also parallel capable now, including the CREATE TABLE .. AS, SELECT INTO, and CREATE MATERIALIZED VIEW

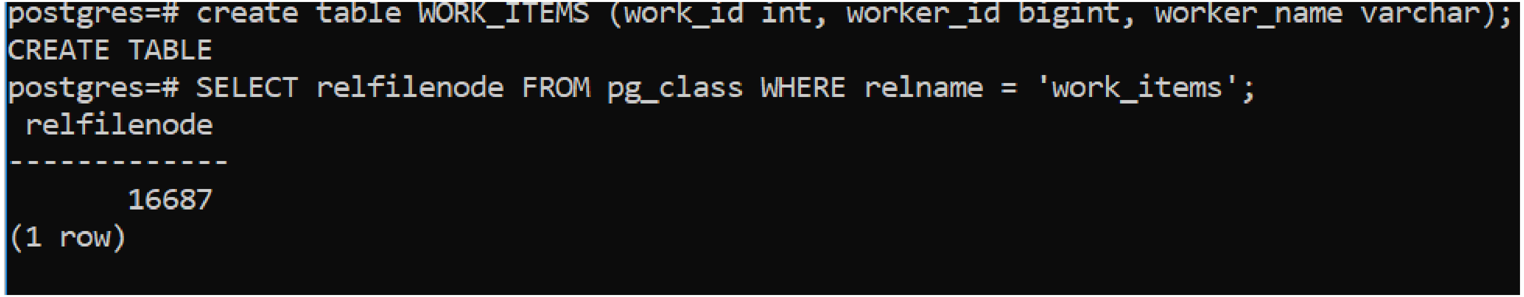
Additional Imrovements

Stored Procedure

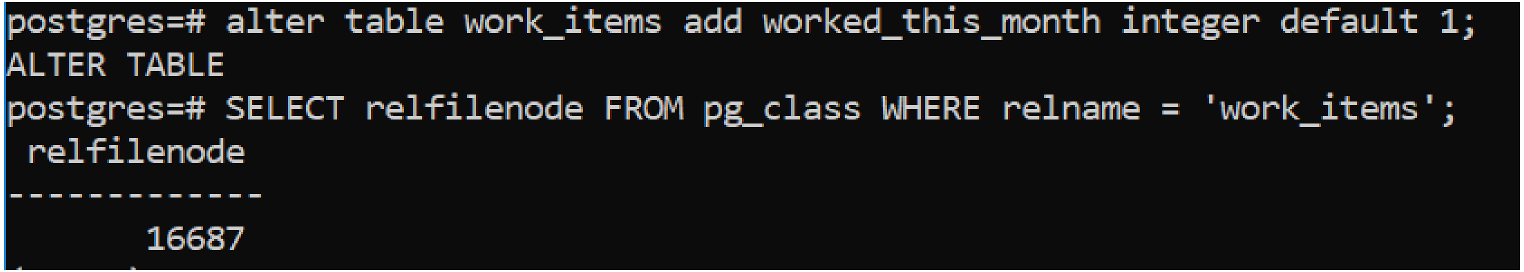
- Allows Transaction commit and abort inside of the procedure
- Can return no values
- Inner transactions cannot be committed independently of outer transactions, i.e., no autonomous transactions
- SQL procedures can be created using the CREATE PROCEDURE command, executed using the CALL command, and are supported by the server-side procedural languages PL/pgSQL, PL/Perl, PL/Python, and PL/Tcl

Adding New Table Columns with Default Values

- No longer have a table re-write when you ALTER TABLE ADD COLUMN with a non-null default



```
postgres=# create table WORK_ITEMS (work_id int, worker_id bigint, worker_name varchar);
CREATE TABLE
postgres=# SELECT relfilenode FROM pg_class WHERE relname = 'work_items';
 relfilenode
-------------
       16687
(1 row)
```



```
postgres=# alter table work_items add worked_this_month integer default 1;
ALTER TABLE
postgres=# SELECT relfilenode FROM pg_class WHERE relname = 'work_items';
 relfilenode
-------------
       16687
```

Command Line Improvements

- New users have found it difficult to do simple things like Quit from psql.
- Starting in PostgreSQL 11, you can use ‘quit’ or ‘exit’ to leave psql (not just the \q or ctrl + D

Improved Statistics

- Previously, while collecting optimizer statistics, most-common-values (MCV) were chosen based on their significance compared to all columns. But now, MCVs are chosen based on their significance *compared to non-MCV values*
- Improve selectivity estimates for >= and <=
- Improve optimizer's row count estimates for EXISTS and NOT EXISTS queries

Why run managed PostgreSQL on Amazon RDS?

A Brief History of Amazon RDS

- 2006: Amazon launches Amazon Web Services (AWS) with Simple Storage Service (S3)
- 2009: Amazon Relational Database Service (RDS) launches with support for MySQL
- 2012: Amazon RDS adds support for Oracle and SQL Server
- 2013: Amazon RDS adds support for PostgreSQL
- 2014: Amazon RDS announces Amazon Aurora
- July 2015: Amazon Aurora with MySQL compatibility goes GA
- October 2017: Amazon Aurora with PostgreSQL compatibility goes GA

Amazon Relational Database Service (RDS) is . . .

Cloud native engine

Amazon Aurora

Open source engines

MySQL

MariaDB

PostgreSQL

Commercial engines

Microsoft SQL Server

ORACLE

RDS platform

- Automatic fail-over
- Backup & recovery
- X-region replication

- Isolation & security
- Industry compliance
- Automated patching

- Advanced monitoring
- Routine maintenance
- Push-button scaling

Image credit: By Mackphillips - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=55946550

Amazon RDS for open source PostgreSQL engines are . . .

Cloud native engine

Amazon Aurora

Open source engines

MySQL

MariaDB

PostgreSQL

Commercial engines

Microsoft SQL Server

ORACLE

RDS platform

- Automatic fail-over
- Backup & recovery
- X-region replication

- Isolation & security
- Industry compliance
- Automated patching

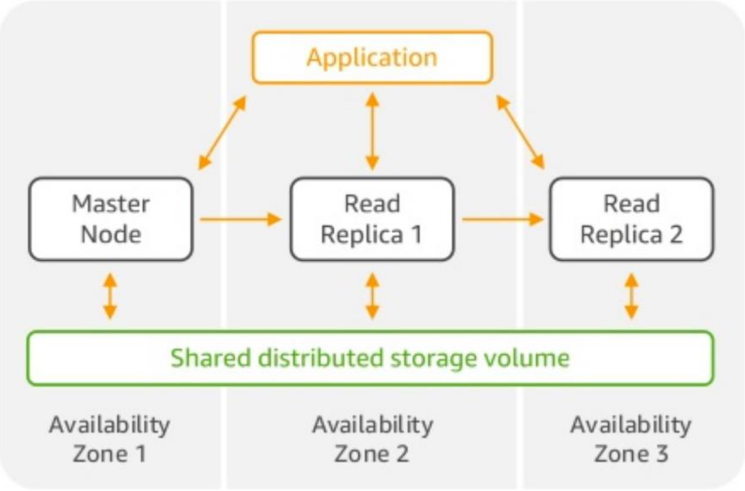
- Advanced monitoring
- Routine maintenance
- Push-button scaling

Image credit: By Mackphillips - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=55946550

Aurora PostgreSQL

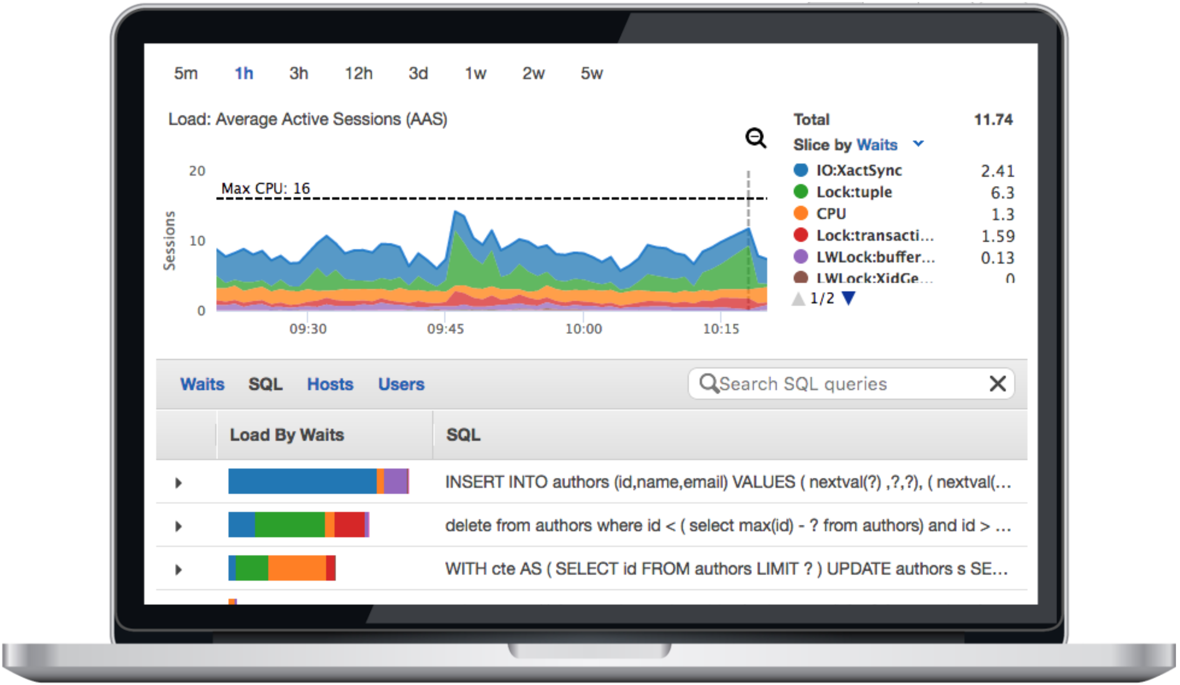
PostgreSQL + Amazon Aurora cloud-optimized storage

- Performance: Up to 3x+ better performance than PostgreSQL alone
- Availability: Failover time of <30 seconds
- Durability: 6 copies across 3 Availability Zones
- Read Replicas: Single-digit millisecond lag times on up to 15 replicas
- Targeting to add support for PG 11 to Aurora PostgreSQL this year



Performance Insights

- Measures DB Load
- Identifies bottlenecks (top SQL, wait events)
- Adjustable time frame (hour, day, week, longer)



Total	11.74
Slice by Waits	
● IO:XactSync	2.41
● Lock:tuple	6.3
● CPU	1.3
● Lock:transacti...	1.59
● LWLock:buffer...	0.13
● LWLock:XidGe...	0

Load By Waits	SQL
	INSERT INTO authors (id,name,email) VALUES (nextval(?) ,?,?), (nextval(...
	delete from authors where id < (select max(id) - ? from authors) and id > ...
	WITH cte AS (SELECT id FROM authors LIMIT ?) UPDATE authors s SE...

Conclusion

- Partition Enhancements
- Parallel Enhancement
- Stored Procedures (Transaction Control)
- Adding New Table Columns with Default Values
- Command Line improvements
- Improved Statistics
- **Amazon RDS for PostgreSQL 11 released earlier this month**
- **ALL the features talked about here are available on RDS PostgreSQL 11**
- **Targeting to add support for PG 11 to Aurora PostgreSQL this year**

- https://www.postgresql.org/docs/11/release-11.html

Thank you!