PG11新特性介绍

—— 研发一部 李梦怡

瀚高基础软件股份有限公司

www.highgo.com

目录

目录
CONTENT

PART 1

概　述

概述

并行执行

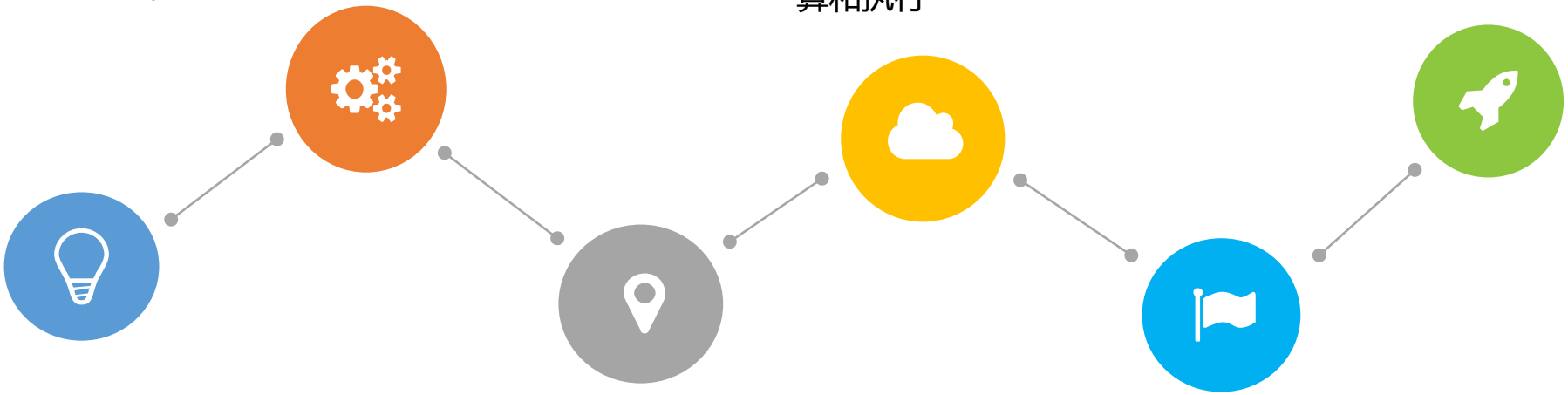
- 并行哈希连接
- 并行创建 B-tree 索引
- 并行执行的CREATE TABLE .. AS、CREATE MATERIALIZED VIEW以及使用UNION的特定查询

表达式的 (JIT) 编译

- 引入JIT编译来加速查询中的表达式的计算和执行

工具命令改进

- psql



分区增强

- 支持哈希分区
- 更新分区键值的UPDATE语句可以将受影响的行移动到相应的新分区中
- 增强了查询语句处理和执行时的分区消除，进而提高了SELECT查询语句的性能
- 支持分区表上的主键、外键、索引以及触发器

存储过程

- 支持嵌入事务

逻辑复制

PART 2

详 细 说 明

分区改进


```
-- 创建哈希分区表
CREATE TABLE part_test1 (x int, y text) PARTITION BY hash (x);

-- 创建子分区
CREATE TABLE part_test1_0 PARTITION OF part_test1 FOR VALUES
WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE part_test1_1 PARTITION OF part_test1 FOR VALUES
WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE part_test1_2 PARTITION OF part_test1 FOR VALUES
WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE part_test1_3 PARTITION OF part_test1 FOR VALUES
WITH (MODULUS 4, REMAINDER 3);
```

当记录中的分区键值字段被更新后，会自动将该记录移至新的正确的分区表中

MODULUS的取值为正整数。
REMAINDER的取值为小于MODULUS的非负整数。

分区改进

-- 插入1000行数据;
INSERT INTO part_test1 SELECT generate_series(0, 999), 'old' ;

```
postgres=# SELECT relname
FROM pg_class
WHERE oid = (SELECT tableoid FROM part_test1 WHERE x = 0);
    relname
--------------
 part_test1_0
(1 row)
```

-- 将0行更新为1003行;
UPDATE part_test1 SET x = 1003, y = 'new' WHERE x = 0;

```
postgres=# SELECT relname
postgres-# FROM pg_class
postgres-# WHERE oid = (SELECT tableoid FROM part_test1 WHERE x = 1003);
   relname
-------------
 part_test1_1
(1 row)
```

分区改进

范围分区和列表分区支持默认分区（当数据不满足所有分区规则时，放入默认分区中）

```
-- 范围分区
CREATE TABLE part_test2 (instant TIMESTAMP WITH TIME ZONE,
description TEXT)
PARTITION BY RANGE (instant);

CREATE TABLE part_test2_2017 PARTITION OF part_test2 FOR VALUES
FROM (‘2017-01-01’) TO (‘2018-01-01’);
CREATE TABLE part_test2_2018 PARTITION OF part_test2 FOR VALUES
FROM (‘2018-01-01’) TO (‘2019-01-01’);

-- 创建默认分区
CREATE TABLE part_test2_default PARTITION OF part_test2 DEFAULT;

-- 父表添加主键
ALTER TABLE part_test2 ADD PRIMARY KEY (instant);
```

分区改进

创建新的子分区时，无法创建与默认分区元组值相同的分区。


```
-- 插入两年的数据
INSERT INTO part_test2
SELECT generate_series( ‘2017-01-01’ ::timestamptz,
  ‘2018-12-31’ , ‘1 day’ ), ‘rain’ ;

-- 插入定义范围外的一条数据
INSERT INTO part_test2 VALUES ( ‘2019-02-20’ , ‘snow’ );

-- 查询
SELECT name, COUNT(*)
FROM part_test2, LATERAL (
SELECT relname
FROM pg_class
WHERE pg_class.oid = part_test2.tableoid) AS table_name
(name)
GROUP BY name
ORDER BY 1;
```

```
        name         | count
---------------------+-------
 part_test2_2017     |   365
 part_test2_2018     |   365
 part_test2_default  |     1
(3 rows)
```

分区改进

允许在查询执行期间消除不需要的分区访问，加速分区过滤，在分区数较多时，性能相比以前的版本有所提升。

PG 11增加了参数 enable_partition_pruning ，仅用于控制分区表（不用于控制inherit, union all等操作）的QUERY。
即以后使用创建分区表的语法创建的表，必须通过enable_partition_pruning参数来控制，是否要对select,update,delete操作过滤到目标分区。

示例：

```
create table pp_lp (a int, value int) partition by list (a);
--create table pp_lp1 partition of pp_lp for values in(1);
--create table pp_lp2 partition of pp_lp for values in(2);
```

分区改进

① enable_partition_pruning = off; constraint_exclusion = 'partition';

```
postgres=# explain (costs off) select * from pp_lp where a = 1;
          QUERY PLAN
--------------------------
 Append
   ->  Seq Scan on pp_lp1
         Filter: (a = 1)
   ->  Seq Scan on pp_lp2
         Filter: (a = 1)
(5 rows)
```

```
postgres=# explain (costs off) update pp_lp set value = 10 where a = 1;
         QUERY PLAN
------------------------
 Update on pp_lp
   Update on pp_lp1
   Update on pp_lp2
   ->  Seq Scan on pp_lp1
         Filter: (a = 1)
   ->  Seq Scan on pp_lp2
         Filter: (a = 1)
(7 rows)
```

```
postgres=# explain (costs off) delete from pp_lp where a = 1;
          QUERY PLAN
--------------------------
 Delete on pp_lp
   Delete on pp_lp1
   Delete on pp_lp2
   ->  Seq Scan on pp_lp1
         Filter: (a = 1)
   ->  Seq Scan on pp_lp2
         Filter: (a = 1)
(7 rows)
```

分区改进

② enable_partition_pruning = on;

```
postgres=# explain (costs off) select * from pp_lp where a = 1;
         QUERY PLAN
-------------------------
 Append
   ->  Seq Scan on pp_lp1
         Filter: (a = 1)
(3 rows)
```

```
postgres=# explain (costs off) update pp_lp set value = 10 where a = 1;
         QUERY PLAN
-------------------------
 Update on pp_lp
   Update on pp_lp1
   ->  Seq Scan on pp_lp1
         Filter: (a = 1)
(4 rows)
```

```
postgres=# explain (costs off) delete from pp_lp where a = 1;
         QUERY PLAN
-------------------------
 Delete on pp_lp
   Delete on pp_lp1
   ->  Seq Scan on pp_lp1
         Filter: (a = 1)
(4 rows)
```

分区改进

增加了分区表全局索引管理

```
--以list分区为例:
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));

create table cities_ab partition of cities (constraint
city_id_nonzero check (city_id!=0))
for values in('a','b’   );

--创建全局索引
create index idx_cities_1 on cities(name);
```

分区改进

对于现有分区，自动添加索引

```
postgres=# \d+ cities
                                                  Table "public.cities"
   Column   |  Type  | Collation | Nullable |                 Default                  | Storage  |
------------+--------+-----------+----------+------------------------------------------+----------+
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |
 name       | text   |           | not null |                                          | extended |
 population | bigint |           |          |                                          | plain    |
Partition key: LIST ("left"(lower(name), 1))
Indexes:
    "idx_cities_1" btree (name)
Partitions: cities_ab FOR VALUES IN ('a', 'b')

postgres=# \d cities_ab
                          Table "public.cities_ab"
   Column   |  Type  | Collation | Nullable |                 Default
------------+--------+-----------+----------+------------------------------------------
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass)
 name       | text   |           | not null |
 population | bigint |           |          |
Partition of: cities FOR VALUES IN ('a', 'b')
Indexes:
    "cities_ab_name_idx" btree (name)
Check constraints:
    "city_id_nonzero" CHECK (city_id <> 0)
```

分区改进

元数据中，主表索引显示为'I'类型，分区上的索引显示为'i'类型。

```
postgres=# select relname,relkind from pg_class where relname ~ 'citi';
       relname       | relkind
---------------------+---------
 idx_cities_1        | I    # 分区表 - 主表上的索引
 cities_ab           | r    # 分区表 - 分区表
 cities_ab_name_idx  | i    # 分区表 - 分区表上的索引
 cities_city_id_seq  | S    # 序列
 cities              | p    # 分区表 - 主表
(5 rows)
```

分区改进

Attach新表时，会在其上自动创建索引。

--新增分区，自动创建索引
create table cities_cd partition of cities (constraint city_id_nonzero check (city_id!=0)) for values in('c','d');
如果这个分区上已经包含了同样定义的索引，那么会自动将这个索引attach到主表的索引中，而不会新建这个索引。

```
postgres=# \d+ cities_cd
                                                Table "public.cities_cd"
   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 | bigint |           |          |                                       | plain    |              |
Partition of: cities FOR VALUES IN ('c', 'd')
Partition constraint: (("left"(lower(name), 1) IS NOT NULL) AND ("left"(lower(name), 1) = ANY (ARRAY['c'::text, 'd'::text])))
Indexes:
    "cities_cd_name_idx" btree (name)
Check constraints:
    "city_id_nonzero" CHECK (city_id <> 0)
```

分区改进

主表全局壳子索引（“ only”）

create index idx_cities_2 on only cities (population); --INVALID索引

此索引不会在分区上构建

```
postgres=# \d+ cities
                                                  Table "public.cities"
   Column   |  Type  | Collation | Nullable |                Default                | Storage  |
------------+--------+-----------+----------+---------------------------------------+----------+
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |
 name       | text   |           | not null |                                         | extended |
 population | bigint |           |          |                                         | plain    |
Partition key: LIST ("left"(lower(name), 1))
Indexes:
    "idx_cities_1" btree (name)
    "idx_cities_2" btree (population) INVALID
Partitions: cities_ab FOR VALUES IN ('a', 'b'),
            cities_cd FOR VALUES IN ('c', 'd')
```

分区改进

将分区表分区索引绑定到主表全局索引

比如cities_ab分区想构建population字段的索引，而其他分区却不想构建，如下：
create index idx_cities_ab_2 on cities_ab (population);

将这个分区上的索引，绑定到INVALID的全局壳子索引下面：
alter index idx_cities_2 **attach partition** idx_cities_ab_2;

```
postgres=# \d+ cities_ab
                                                  Table "public.cities_ab"
   Column   |  Type  | Collation | Nullable |                 Default                 | Storage  |
------------+--------+-----------+----------+-----------------------------------------+----------+
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |
 name       | text   |           | not null |                                         | extended |
 population | bigint |           |          |                                         | plain    |
Partition of: cities FOR VALUES IN ('a', 'b')
Partition constraint: (("left"(lower(name), 1) IS NOT NULL) AND ("left"(lower(name), 1) = ANY (ARR
Indexes:
    "cities_ab_name_idx" btree (name)
    "idx_cities_ab_2" btree (population)
Check constraints:
    "city_id_nonzero" CHECK (city_id <> 0)
```

分区改进

删除全局索引时，自动删除已经attach在这个全局索引下面的所有索引

```
postgres=# drop index idx_cities_1;
DROP INDEX
postgres=# \d cities
                                  Table "public.cities"
   Column   |  Type  | Collation | Nullable |                  Default
------------+--------+-----------+----------+-------------------------------------
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass)
 name       | text   |           | not null |
 population | bigint |           |          |
Partition key: LIST ("left"(lower(name), 1))
Indexes:
    "idx_cities_2" btree (population) INVALID
Number of partitions: 2 (Use \d+ to list them.)
postgres=# \d cities_cd
                                 Table "public.cities_cd"
   Column   |  Type  | Collation | Nullable |                  Default
------------+--------+-----------+----------+-------------------------------------
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass)
 name       | text   |           | not null |
 population | bigint |           |          |
Partition of: cities FOR VALUES IN ('c', 'd')
Check constraints:
    "city_id_nonzero" CHECK (city_id <> 0)
```

分区改进

删除全局索引时，自动删除已经attach在这个全局索引下面的所有索引

```
postgres=# drop index idx_cities_1;
DROP INDEX
postgres=# \d cities
                                  Table "public.cities"
   Column   |  Type  | Collation | Nullable |                  Default
------------+--------+-----------+----------+-------------------------------------
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass)
 name       | text   |           | not null |
 population | bigint |           |          |
Partition key: LIST ("left"(lower(name), 1))
Indexes:
    "idx_cities_2" btree (population) INVALID
Number of partitions: 2 (Use \d+ to list them.)
postgres=# \d cities_cd
                                 Table "public.cities_cd"
   Column   |  Type  | Collation | Nullable |                  Default
------------+--------+-----------+----------+-------------------------------------
 city_id    | bigint |           | not null | nextval('cities_city_id_seq'::regclass)
 name       | text   |           | not null |
 population | bigint |           |          |
Partition of: cities FOR VALUES IN ('c', 'd')
Check constraints:
    "city_id_nonzero" CHECK (city_id <> 0)
```

分区改进

当两个分区表的分区定义一致时，在分区字段上JOIN就可以用到分区与分区之间直接并行JOIN，而不需要将数据追加后再JOIN。（enable_partitionwise_join）

必须满足以下条件，优化器才会使用分区JOIN分区：

1、打开enable_partitionwise_join开关
2、分区表的模式一致（range, list, hash）
3、分区表的分区数目一致
4、分区表每个分区的定义一致
5、分区字段必须参与JOIN（但是可以含其他JOIN字段）
6、分区字段的类型必须一致
7、如果是表达式分区键，那么表达式必须一致

注意：由于判断是否使用智能分区并行JOIN需要耗费一定的优化器判断逻辑，会带来执行计划成本的提升，所以enable_partitionwise_join默认是关闭的。

分区改进

建立两个分区结构一样的分区表

```
create table a(id int, info text, crt_time timestamp) partition by range(id);
create table a0 partition of a for values from (0) to (10000);
create table a1 partition of a for values from (10000) to (20000);
create table a2 partition of a for values from (20000) to (30000);
create table a3 partition of a for values from (30000) to (40000);

create table b(bid int , info text, crt_time timestamp, c1 int, c2 int) partition by range(bid);
create table b0 partition of b for values from (0) to (10000);
create table b1 partition of b for values from (10000) to (20000);
create table b2 partition of b for values from (20000) to (30000);
create table b3 partition of b for values from (30000) to (40000);
```

分区改进


```
postgres=# set enable_partitionwise_join =off;
SET
postgres=# explain select a.* from a join b on (a.id=b.bid);
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Merge Join  (cost=728.10..2131.62 rows=92208 width=44)
   Merge Cond: (b0.bid = a0.id)
   ->  Sort  (cost=345.88..356.08 rows=4080 width=4)
         Sort Key: b0.bid
         ->  Append  (cost=0.00..101.20 rows=4080 width=4)
               ->  Seq Scan on b0  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b1  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b2  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b3  (cost=0.00..20.20 rows=1020 width=4)
   ->  Sort  (cost=382.21..393.51 rows=4520 width=44)
         Sort Key: a0.id
         ->  Append  (cost=0.00..107.80 rows=4520 width=44)
               ->  Seq Scan on a0  (cost=0.00..21.30 rows=1130 width=44)
               ->  Seq Scan on a1  (cost=0.00..21.30 rows=1130 width=44)
               ->  Seq Scan on a2  (cost=0.00..21.30 rows=1130 width=44)
               ->  Seq Scan on a3  (cost=0.00..21.30 rows=1130 width=44)
(16 rows)
```

分区改进


```
postgres=# set enable_partitionwise_join =on;
SET
postgres=# explain select a.* from a join b on (a.id=b.bid);
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Append  (cost=149.77..1080.54 rows=23052 width=44)
   ->  Merge Join  (cost=149.77..241.32 rows=5763 width=44)
         Merge Cond: (b0.bid = a0.id)
         ->  Sort  (cost=71.17..73.72 rows=1020 width=4)
               Sort Key: b0.bid
               ->  Seq Scan on b0  (cost=0.00..20.20 rows=1020 width=4)
         ->  Sort  (cost=78.60..81.43 rows=1130 width=44)
               Sort Key: a0.id
               ->  Seq Scan on a0  (cost=0.00..21.30 rows=1130 width=44)
   ->  Merge Join  (cost=149.77..241.32 rows=5763 width=44)
         Merge Cond: (b1.bid = a1.id)
         ->  Sort  (cost=71.17..73.72 rows=1020 width=4)
               Sort Key: b1.bid
               ->  Seq Scan on b1  (cost=0.00..20.20 rows=1020 width=4)
         ->  Sort  (cost=78.60..81.43 rows=1130 width=44)
               Sort Key: a1.id
               ->  Seq Scan on a1  (cost=0.00..21.30 rows=1130 width=44)
   ->  Merge Join  (cost=149.77..241.32 rows=5763 width=44)
         Merge Cond: (b2.bid = a2.id)
         ->  Sort  (cost=71.17..73.72 rows=1020 width=4)
               Sort Key: b2.bid
               ->  Seq Scan on b2  (cost=0.00..20.20 rows=1020 width=4)
         ->  Sort  (cost=78.60..81.43 rows=1130 width=44)
               Sort Key: a2.id
               ->  Seq Scan on a2  (cost=0.00..21.30 rows=1130 width=44)
   ->  Merge Join  (cost=149.77..241.32 rows=5763 width=44)
         Merge Cond: (b3.bid = a3.id)
```

分区改进

当分区结构不一样时，不会用到分区并行JOIN

```
postgres=# create table b4 partition of b for values from (40000) to (50000);
CREATE TABLE
postgres=# explain select a.* from a join b on (a.id=b.bid);
                                  QUERY PLAN
------------------------------------------------------------------------------
 Merge Join  (cost=822.78..2574.28 rows=115260 width=44)
   Merge Cond: (a0.id = b0.bid)
   ->  Sort  (cost=382.21..393.51 rows=4520 width=44)
         Sort Key: a0.id
         ->  Append  (cost=0.00..107.80 rows=4520 width=44)
               ->  Seq Scan on a0  (cost=0.00..21.30 rows=1130 width=44)
               ->  Seq Scan on a1  (cost=0.00..21.30 rows=1130 width=44)
               ->  Seq Scan on a2  (cost=0.00..21.30 rows=1130 width=44)
               ->  Seq Scan on a3  (cost=0.00..21.30 rows=1130 width=44)
   ->  Sort  (cost=440.57..453.32 rows=5100 width=4)
         Sort Key: b0.bid
         ->  Append  (cost=0.00..126.50 rows=5100 width=4)
               ->  Seq Scan on b0  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b1  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b2  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b3  (cost=0.00..20.20 rows=1020 width=4)
               ->  Seq Scan on b4  (cost=0.00..20.20 rows=1020 width=4)
(17 rows)
```

分区改进

允许对每个分区单独评估分区表上的聚合函数，随后合并结果。

```
postgres=# set enable_partitionwise_aggregate=on;
SET
```

```
postgres=# EXPLAIN ANALYZE SELECT a, count(*) FROM plt1 GROUP BY a;
                                                        QUERY
PLAN
-----------------------------------------------------------------------------------------------------------
 Append  (cost=5100.00..61518.90 rows=30 width=12) (actual
time=324.837..944.804 rows=30 loops=1)
   ->  Foreign Scan  (cost=5100.00..20506.30 rows=10 width=12) (actual
time=324.837..324.838 rows=10 loops=1)
         Relations: Aggregate on (public.fplt1_p1 plt1)
   ->  Foreign Scan  (cost=5100.00..20506.30 rows=10 width=12) (actual
time=309.954..309.956 rows=10 loops=1)
         Relations: Aggregate on (public.fplt1_p2 plt1)
   ->  Foreign Scan  (cost=5100.00..20506.30 rows=10 width=12) (actual
time=310.002..310.004 rows=10 loops=1)
         Relations: Aggregate on (public.fplt1_p3 plt1)
 Planning time: 0.370 ms
 Execution time: 945.384 ms
(9 rows)
```

分区改进


```
postgres=# set enable_partitionwise_aggregate=off;
SET
```

```
postgres=# EXPLAIN ANALYZE SELECT a, count(*) FROM plt1 GROUP BY a;
                                                                QUERY
PLAN
-----------------------------------------------------------------------------------------
 HashAggregate  (cost=121518.01..121518.31 rows=30 width=12) (actual
time=6498.452..6498.459 rows=30 loops=1)
   Group Key: plt1.a
   ->  Append  (cost=0.00..106518.00 rows=3000001 width=4) (actual
time=0.595..5769.592 rows=3000000 loops=1)
         ->  Seq Scan on plt1  (cost=0.00..0.00 rows=1 width=4) (actual
time=0.007..0.007 rows=0 loops=1)
         ->  Foreign Scan on fplt1_p1  (cost=100.00..35506.00 rows=1000000
width=4) (actual time=0.587..1844.506 rows=1000000 loops=1)
         ->  Foreign Scan on fplt1_p2  (cost=100.00..35506.00 rows=1000000
width=4) (actual time=0.384..1839.633 rows=1000000 loops=1)
         ->  Foreign Scan on fplt1_p3  (cost=100.00..35506.00 rows=1000000
width=4) (actual time=0.402..1876.505 rows=1000000 loops=1)
 Planning time: 0.251 ms
 Execution time: 6499.018 ms
(9 rows)
```

分区改进

允许postgres_fdw的外部表作为分区，同时允许insert,update,copy数据路由到对应外部表分区。

支持postgres_fdw外部表作为分区，将聚合下推到对应的外部数据源执行。

分区改进

外部服务器：

```
create table a (id int,name varchar(20));

insert into a values(1,’a’);
```

```
CREATE EXTENSION postgres_fdw;  ##创建扩展：

create server server_remote foreign data wrapper postgres_fdw options(host '192.168.100.110',port '5866',dbname 'highgo');  ##创建外部服务器

create user mapping for lmy server server_remote options(user  ‘lmy',password  ‘highgo');  ##创建用户映射

CREATE FOREIGN TABLE a_remote(id int,name varchar(20)) server server_remote options (table_name  ‘a’);  ##创建外部表

CREATE TABLE plist2 (id int, name VARCHAR(20)) PARTITION BY LIST (id) ;

alter table plist2 attach partition a_remote for values in (1);

insert into plist2 values(1,'b');
```

分区改进

- 哈希分区
- 默认分区
- 分区过滤
- 全局索引
- 并行JOIN
- 并行聚合
- postgres_fdw
- **其他事项**

允许在分区键上创建UNIQUE约束和PRIMARY KEY约束。

分区表支持foreign key。

允许对分区表主表创建触发器，同时这些触发器自动建立到所有分区上，并且未来新增的分区，也会自动创建对应触发器。

支持使用UPDATE更新分区键值，更新的元组将被移动到符合条件的分区。

并行改进

并行创建索引

parallel append

parallel hash

PostgreSQL 11 对几种数据集的定义指令增加了并行处理功能，最显著的就是通过**CREATE INDEX**指令创建的**B-Tree**索引。

其他几种支持并行化操作的还有：
CREATE TABLE .. AS
SELECT INTO
CREATE MATERIALIZED VIEW

并行改进

备注：测试用例和结果来自Francs（4核 8GB内存虚拟机）

创建语句：

```
CREATE TABLE big(user_id int4,user_name text,ctime timestamp(6) without time
zone default clock_timestamp() );
INSERT INTO big(user_id,user_name) SELECT  n ,n || '_data' FROM
generate_series(1,30000000) n;;
```

设置并行度：

```
set max_parallel_maintenance_workers =4;
```

创建索引：

```
CREATE INDEX idx_big_ctime ON big USING BTREE(ctime);
```

并行改进

并行创建索引

parallel append

parallel hash

备注：测试用例和结果来自Francs（4核 8GB内存虚拟机）

```
top - 19:35:45 up 390 days, 10:48,  2 users,  load average: 0.00, 0.00, 0.00
Tasks: 185 total,   3 running, 182 sleeping,   0 stopped,   0 zombie
Cpu(s): 56.5%us, 12.5%sy,  0.0%ni, 30.3%id,  0.7%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8062340k total,  6978968k used,  1083372k free,    25040k buffers
Swap:        0k total,        0k used,        0k free,  5616096k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21164 pg11      20   0  572m 176m 8152 D 67.8  2.2   0:02.20 postgres: francs francs [local] CREATE INDEX
21541 pg11      20   0  555m 149m 3536 R 61.1  1.9   0:01.84 postgres: parallel worker for PID 21164
21543 pg11      20   0  531m 132m 3512 D 55.5  1.7   0:01.67 postgres: parallel worker for PID 21164
21540 pg11      20   0  515m 109m 3540 D 49.2  1.4   0:01.48 postgres: parallel worker for PID 21164
21542 pg11      20   0  611m 210m 3496 R 39.2  2.7   0:01.18 postgres: parallel worker for PID 21164
21533 pg11      20   0 15040 1292  940 R  0.3  0.0   0:00.02 top -c -U pg11
14365 pg11      20   0  105m 1656 1332 S  0.0  0.0   0:00.00 -bash
14731 pg11      20   0  105m 1680 1340 S  0.0  0.0   0:00.06 -bash
21163 pg11      20   0  114m 1896 1528 S  0.0  0.0   0:00.01 psql francs francs
26280 pg11      20   0  402m  27m  27m S  0.0  0.3   0:00.10 /opt/pgsql_11beta3/bin/postgres
26295 pg11      20   0  114m  956  500 S  0.0  0.0   0:00.00 postgres: logger
26300 pg11      20   0  402m 3640 3132 S  0.0  0.0   0:00.00 postgres: checkpointer
26301 pg11      20   0  402m 3348 2852 S  0.0  0.0   0:00.12 postgres: background writer
26302 pg11      20   0  402m  17m  16m S  0.0  0.2   0:00.85 postgres: walwriter
26303 pg11      20   0  402m 2032 1248 S  0.0  0.0   0:00.11 postgres: autovacuum launcher
26304 pg11      20   0  116m  908  428 S  0.0  0.0   0:00.01 postgres: archiver
26305 pg11      20   0  116m 1136  576 S  0.0  0.0   0:00.21 postgres: stats collector
26306 pg11      20   0  402m 1704 1008 S  0.0  0.0   0:00.00 postgres: logical replication launcher
```

并行改进

并行创建索引

parallel append

parallel hash

备注：测试用例和结果来自Francs（4核 8GB内存虚拟机）

max_parallel_maintenance_workers	索引创建时间(毫秒)
0	14938.738
2	10469.283
4	10439.237
6	11577.147
8	17020.216

并行改进

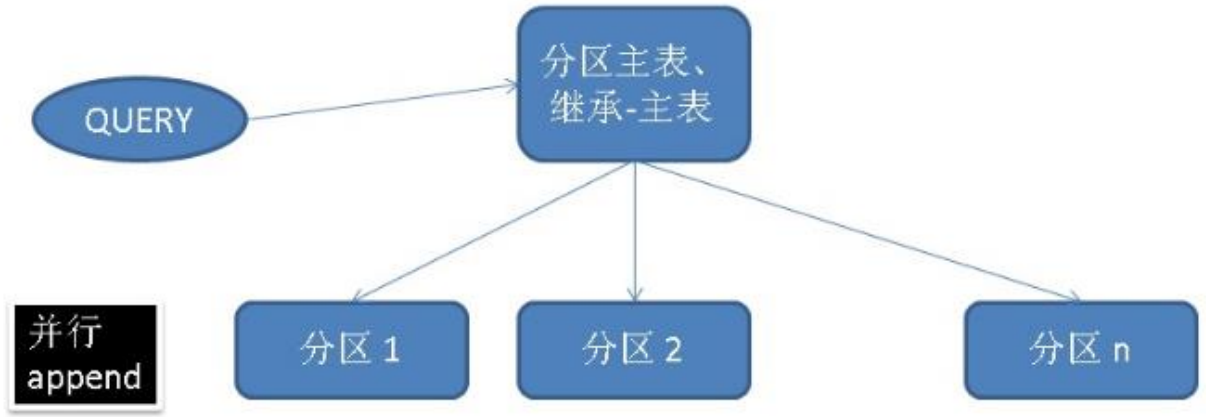
- 并行创建索引
- **parallel append**
- parallel hash

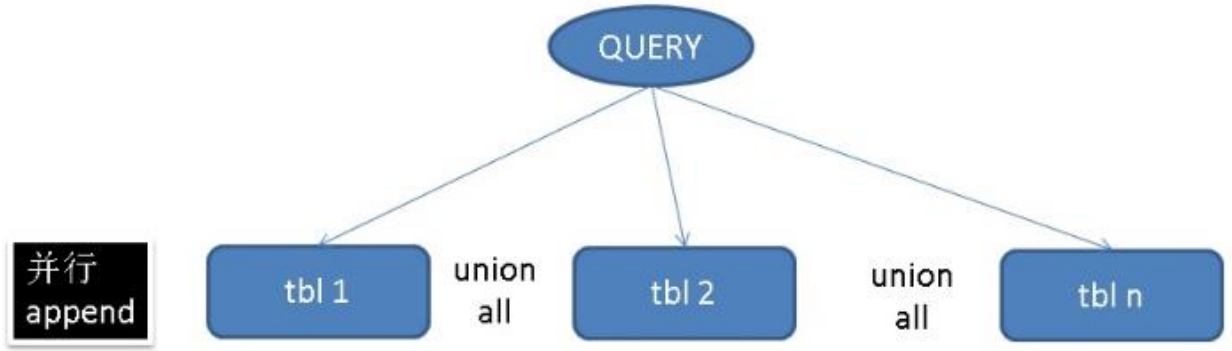
支持parallel append扫描多个子分区。

控制参数：
enable_parallel_append (boolean) ，默认为on。

并行改进

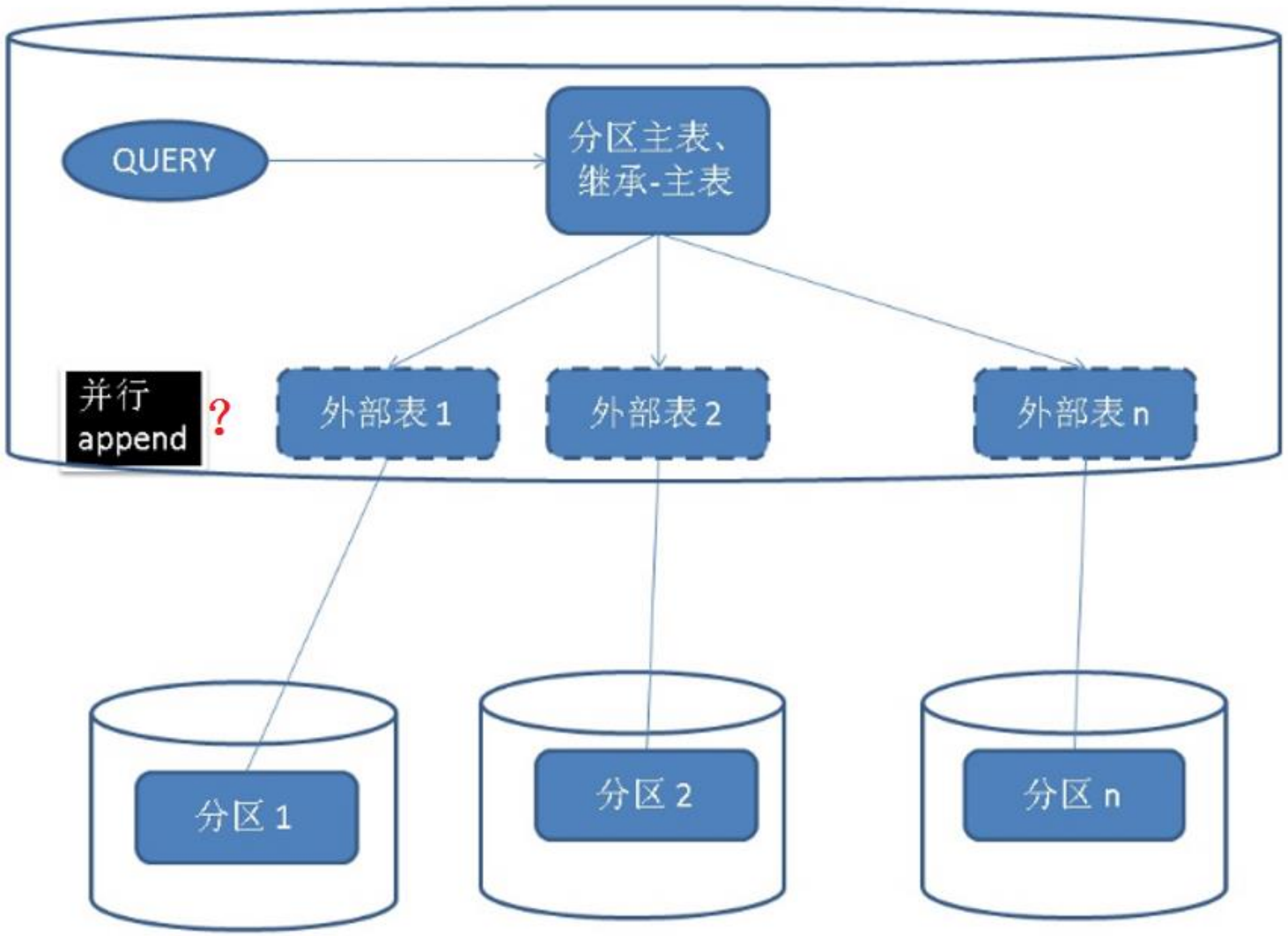
- 并行创建索引
- **parallel append**
- parallel hash





并行改进

- 并行创建索引
- **parallel append**
- parallel hash



并行改进

case	parallel append 耗时	串行 append 耗时	parallel append 性能提升	点评
1亿，4个并行分片，每个分片返回少量数据	2.37 秒	6.99 秒	2.95 倍	几乎线性提升
1亿，4个并行分片，每个分片返回大量数据（但是下一个NODE包括并行聚合）	6.46 秒	21.7 秒	3.36 倍	几乎线性提升
1亿，4个并行分片，每个分片返回大量数据（下一个NODE需要串行返回大量数据）	76.5 秒	18.3 秒	- 4.18 倍	append的下一个NODE需要返回大量数据时不适合使用并行append
2亿，64个并行分片，每个分片返回少量数据	0.655 秒	14.18 秒	21.65 倍	并行越大，提升越明显，这里还需要考虑内存带宽瓶颈（20多倍时，处理速度为12.9 GB/s）

并行改进

- 并行创建索引
- parallel append
- parallel hash

支持Parallel Hash Join。

控制参数：
enable_parallel_hash ， 默认为on。

并行改进

- 并行创建索引
- parallel append
- parallel hash

备注：测试用例和结果来自Francs

大表：5000万条数据
小表：800万条数据

开启并行	2738 ms
关闭并行	3496 ms

21.6%

PG10执行计划

```
des=> EXPLAIN SELECT t_small.name
  FROM t_big JOIN t_small ON (t_big.id = t_small.id)
       AND t_small.id < 100;
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Gather  (cost=151870.58..661385.28 rows=4143 width=13)
   Workers Planned: 4
   ->  Hash Join  (cost=150870.58..659970.98 rows=1036 width=13)
         Hash Cond: (t_big.id = t_small.id)
         ->  Parallel Seq Scan on t_big  (cost=0.00..470246.58 rows=10358258 width=4)
         ->  Hash  (cost=150860.58..150860.58 rows=800 width=17)
               ->  Seq Scan on t_small  (cost=0.00..150860.58 rows=800 width=17)
                     Filter: (id < 100)
(8 rows)
```

并行改进

- 并行创建索引
- parallel append
- parallel hash

备注：测试用例和结果来自Francs

大表：5000万条数据
小表：800万条数据

开启并行	2738 ms
关闭并行	3496 ms

21.6%

PG11执行计划

```
francs=> EXPLAIN SELECT t_small.name
  FROM t_big JOIN t_small ON (t_big.id = t_small.id)
       AND t_small.id < 100;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather  (cost=76862.42..615477.60 rows=800 width=13)
   Workers Planned: 4
   ->  Parallel Hash Join  (cost=75862.42..614397.60 rows=200 width=13)
         Hash Cond: (t_big.id = t_small.id)
         ->  Parallel Seq Scan on t_big  (cost=0.00..491660.86 rows=12499686 width=4)
         ->  Parallel Hash  (cost=75859.92..75859.92 rows=200 width=17)
               ->  Parallel Seq Scan on t_small  (cost=0.00..75859.92 rows=200 width=17)
                     Filter: (id < 100)
(8 rows)
```

并行改进

- 并行创建索引
- parallel append
- parallel hash

```
select count(*)
 from lineitem
 join orders on l_orderkey = o_orderkey
where o_totalprice > 5.00;
```

PG10：

```
Finalize Aggregate
 ->  Gather
  Workers Planned: 2
  ->  Partial Aggregate
   ->  Hash Join
     Hash Cond: (lineitem.l_orderkey
= orders.o_orderkey)
      ->  Parallel Seq Scan on lineitem
      ->  Hash
       ->  Seq Scan on orders
         Filter: (o_totalprice > 5.00)
```

PG11：

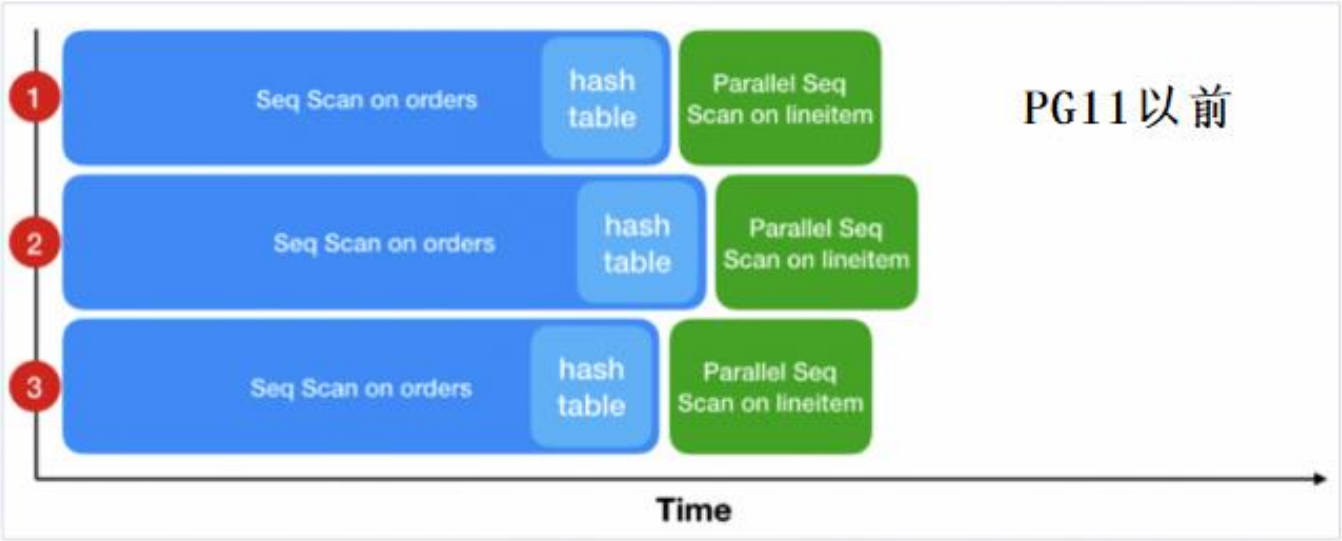
```
Finalize Aggregate
  ->  Gather
   Workers Planned: 2
   ->  Partial Aggregate
    ->  Parallel Hash Join
      Hash Cond: (lineitem.l_orderkey =
orders.o_orderkey)
      ->  Parallel Seq Scan on lineitem
      ->  Parallel Hash
       ->  Parallel Seq Scan on orders
         Filter: (o_totalprice > 5.00)
```

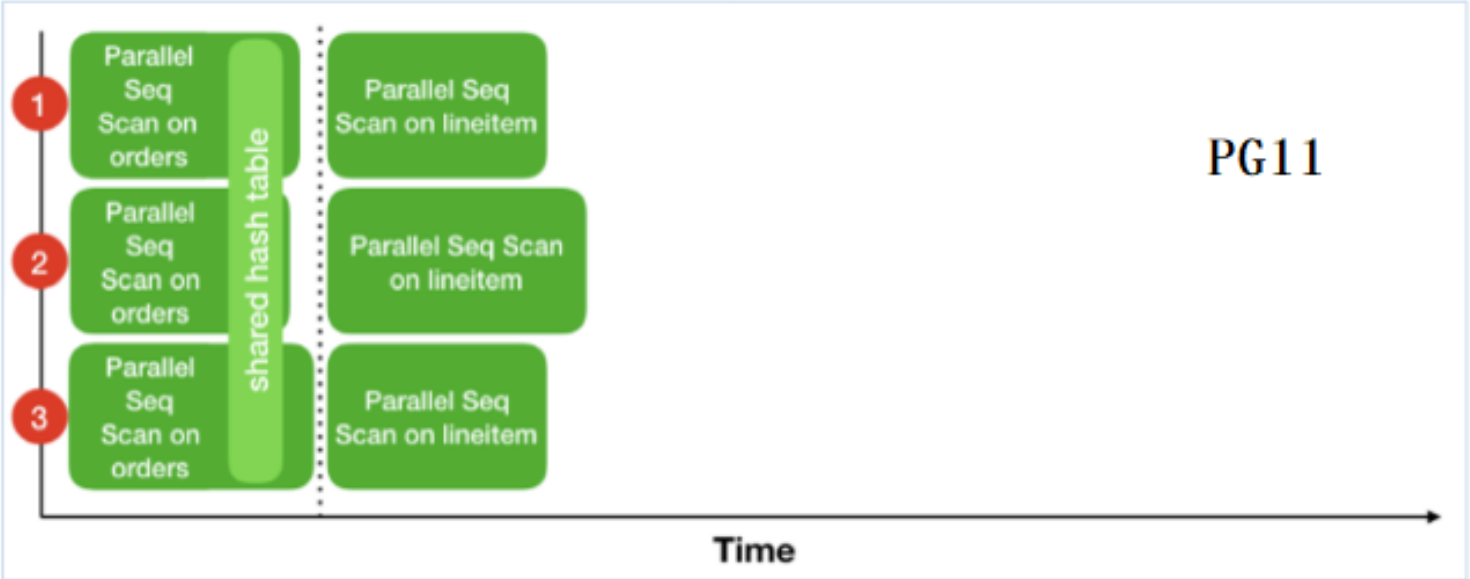
并行改进

并行创建索引

parallel append

parallel hash





存储过程

增加对存储过程的支持，支持嵌入式事务。

使用存储过程的好处：

与常规函数相比，过程不仅可以查询或修改数据，它们还具有在过程中开始/提交/回滚事务的能力。
特别是对于从Oracle迁移到PostgreSQL的人来说，新的过程功能可以节省大量时间。

SQL存储过程可以使用**CREATE PROCEDURE命令创建**，可以**执行CALL命令进行调用。**

存储过程

说明

示例

```
postgres=# CREATE PROCEDURE my_table_task() LANGUAGE plpgsql AS $$
postgres$# DECLARE
postgres$# BEGIN
postgres$#   CREATE TABLE table_committed (id int);
postgres$#   COMMIT;
postgres$#   CREATE TABLE table_rolled_back (id int);
postgres$#   ROLLBACK;
postgres$# END $$;
CREATE PROCEDURE
```

备注：1.事务性DDL。PG中可以在事务中运行DDL，而不被隐式提交。
DROP DATABASE、CREATE TABLESPACE/DROP TABLESPACE等少量DDL除外。

存储过程

说明

示例

```
postgres=# CALL my_table_task();
CALL
Time: 22.256 ms
```

```
postgres=# select * from table_committed;
 id
....
(0 rows)

Time: 1.000 ms
postgres=# select * from table_rolled_back;
2018-10-29 17:24:04.712 CST [15542] ERROR:  relation "table_rolled_back" does not exist at character 15
```

procedure与函数不同的地方，没有返回值的部分，同时调用方法使用CALL而不是select procedure_name。

JIT编译

PostgreSQL 11 引入了对 JIT （Just-In-Time ）编译的支持，以加速查询执行期间某些表达式的执行，主要是两个部分：表达式评估和元组变形。

表达式评估用于评估WHERE子句、聚合等。可以通过生成特定于每种情况的代码来加速它。

元组变形是将磁盘上的元组转换为其内存中表示的过程。可以通过创建特定于表布局的函数和要提取的列数来加速它。

JIT编译

说明

使用

JIT表达式的编译使用LLVM项目编译器的架构来提升在WHERE条件、指定列表、聚合以及一些内部操作的表达式的编译执行。
要使用JIT 编译，用户需要在configure时指定--with-llvm参数，并在系统中启用JIT编译，可通过在PostgreSQL的配置文件中设置jit = on，或是在PostgreSQL 当前会话中执行SET jit = on 。

测试数据（from Francs）

Jit=off	5千万	2385 ms
Jit=on	5千万	2154 ms

↑ **9.7%**

JIT编译

控制参数：

1. jit (boolean)
默认ON，表示开启JIT。
2. jit_above_cost (floating point)
默认100000，当planner发现COST大于这个值时，优化器会启用JIT动态编译。
3. jit_optimize_above_cost (floating point)
默认500000，当planner发现COST大于这个值时，优化器会启用JIT动态编译优化。
4. jit_inline_above_cost (floating point)
默认500000，当planner发现COST大于这个值时，优化器会对用户自定义函数、操作符(目前仅支持C, internal类型的函数)启用JIT优化。
5. jit_provider (string)
为了让JIT支持更多的编译器，PG设计时对编译器的支持也是模块化的，通过jit_provider可以指定使用哪个编译器，当然这个需要实现对应的provider接口才行。

目前PG默认选择的是LLVM编译器，原因是LLVM友好的许可协议与PG的开源许可协议无冲突。第二方面是LLVM后面有很强大的公司在支撑，比如苹果。

添加非空默认值不需要重写表

原理

举例

PostgreSQL11之前，在给表加列时，如果不设置列的默认值，不需要rewrite table。但是若设置新增列的默认值，则必须rewrite table。

ALTER语句会长时间保持对表的写锁定，如果需要将新列添加到具有默认值和NOT NULL约束的大表中，由于表重写，它还可能涉及过多的IO。

添加非空默认值不需要重写表

原理

举例

```
-- Postgres 10
CREATE TABLE alter_test (id SERIAL, name TEXT);
INSERT INTO alter_test (name) SELECT repeat('x', 100);
SELECT relfilenode FROM pg_class WHERE relname = 'alter_test';
 relfilenode
-------------
       16439

ALTER TABLE alter_test ADD COLUMN col1 INTEGER;

SELECT relfilenode FROM pg_class WHERE relname = 'alter_test';
 relfilenode
-------------
       16439

ALTER TABLE alter_test ADD COLUMN col2 INTEGER DEFAULT 1;

SELECT relfilenode FROM pg_class WHERE relname = 'alter_test';
 relfilenode
-------------
       16447
```

添加非空默认值不需要重写表


```
-- Postgres 11
CREATE TABLE alter_test (id SERIAL, name TEXT);
INSERT INTO alter_test (name) SELECT repeat('x', 100);
SELECT relfilenode FROM pg_class WHERE relname = 'alter_test';
 relfilenode
-------------
       16388

ALTER TABLE alter_test ADD COLUMN col1 INTEGER;

SELECT relfilenode FROM pg_class WHERE relname = 'alter_test';
 relfilenode
-------------
       16388

ALTER TABLE alter_test ADD COLUMN col2 INTEGER DEFAULT 1;

SELECT relfilenode FROM pg_class WHERE relname = 'alter_test';
 relfilenode
-------------
       16388
```

逻辑复制

说明

- 支持TRUNCATE语句

- 增加pg_replication_slot_advance函数
  当PostgreSQL 10中的逻辑复制环境发生冲突时，需要通过在备机上执行pg_replication_origin_advance函数来指定逻辑复制的启动LSN以解决冲突。在PostgreSQL 11中，可以在PUBLICATION实例中执行pg_replication_slot_advance函数达到同样的目的。

工具命令改进

说明

① 允许通过initdb 或 使用pg_resetwal通过参数--wal-segsize = <wal_segment_size>重置WAL时更改WAL文件的大小。
。之前16MB的缺省值只能在编译时更改。
废弃configure命令中的--with-wal-segsize选项。

② psql增加\ gdesc命令，显示最近执行的查询的列名和数据类型。

③ psql中支持exit和quit命令用于退出。

④pg_basebackup命令增加--no-verify-checksum选项跳过校验和验证过程；增加--create-slot选项创建复制槽，此选项与--slot选项一起使用。

可靠性提高

01 一致性检查

提供块一致性检查工具pg_verify_checksums命令。该命令必须在停止实例后执行。

02 备份块校验

pg_basebackup命令支持检查备份块的校验和。

03 索引一致性检查

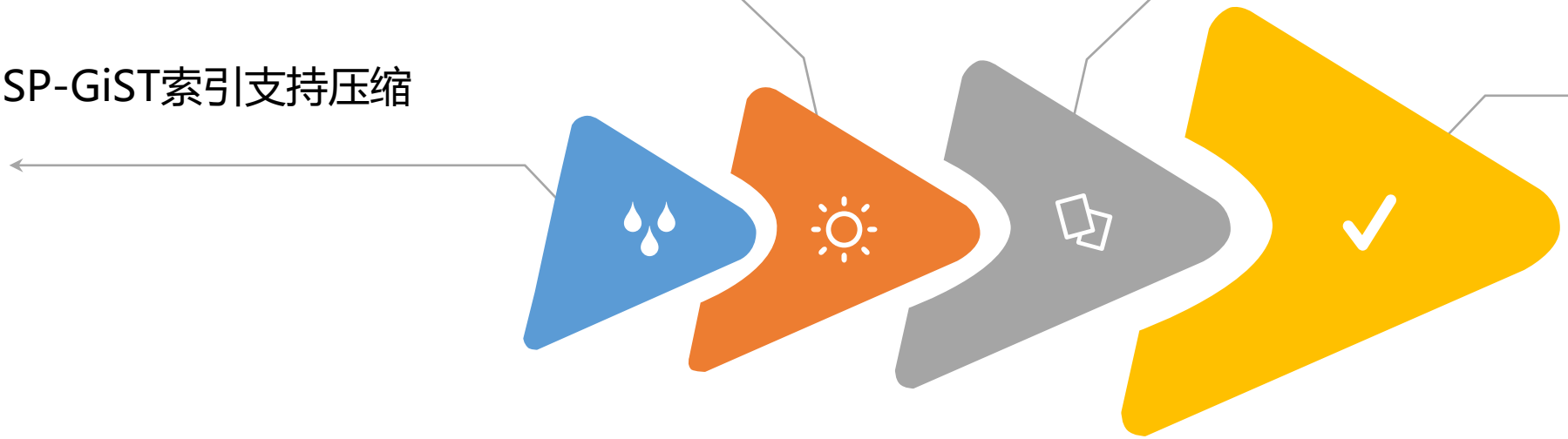
提供amcheck模块。该模块可以检查B-Tree索引的一致性。

索引改进

SP-GiST索引支持压缩

允许polygon类型使用SP-GiST索引 (空间分区索引)

允许SP-GiST索引支持前缀模糊查询 (^@ 与 LIKE 'word%' 效果一致) ，并且效率比btree索引更高



允许唯一约束索引中，使用INCLUDE包含非唯一约束的字段的内容，但仅用于索引扫描

pg_prewarm

说明

pg_prewarm，可以将磁盘上的数据预加载到Postgres缓冲区缓存中。

系统将运行后台进程（postgres: autoprewarm master）定期将共享缓冲区的内容记录在名为autoprewarm.blocks的文件中，数据库重启后将重新加载这些块。
如果参数max_worker_processes为0，则后台工作进程将不会启动。

pg_prewarm.autoprewarm
默认值“on”，可启用自动prewarm功能

pg_prewarm.autoprewarm_interval
定期保存共享内存块信息的最小间隔（以秒为单位）。 默认值为300秒（5分钟）。

JSONB转换

添加扩展以将JSONB数据转换为PL / Perl和PL / Python

增加了插件jsonb_plpython transform，可以将SQL的jsonb类型映射到python编程语言的内置类型中。

增加了插件jsonb_plperl transform，可以将SQL的jsonb类型映射到perl编程语言的内置类型中。

监控

log_statement_stats,log_parser_stats,log_planner_stats, log_executor_stats中显示内存使用情况

01

02

增加pg_stat_activity.backend_type来显示后台工作进程的类型，这种类型在ps输出中也可见

备份和流复制

将已准备好的事务信息传递给逻辑复制订阅者

pg_basebackup不再备份unlogged表、临时表和pg_internal.init文件

将时间线信息添加到backup_label文件中。添加检查机制，确保wal时间线与backup_label文件的时间线匹配

将主机和端口连接信息添加到pg_stat_wal_receiver系统视图中

与PG10的不兼容性

pg_dump

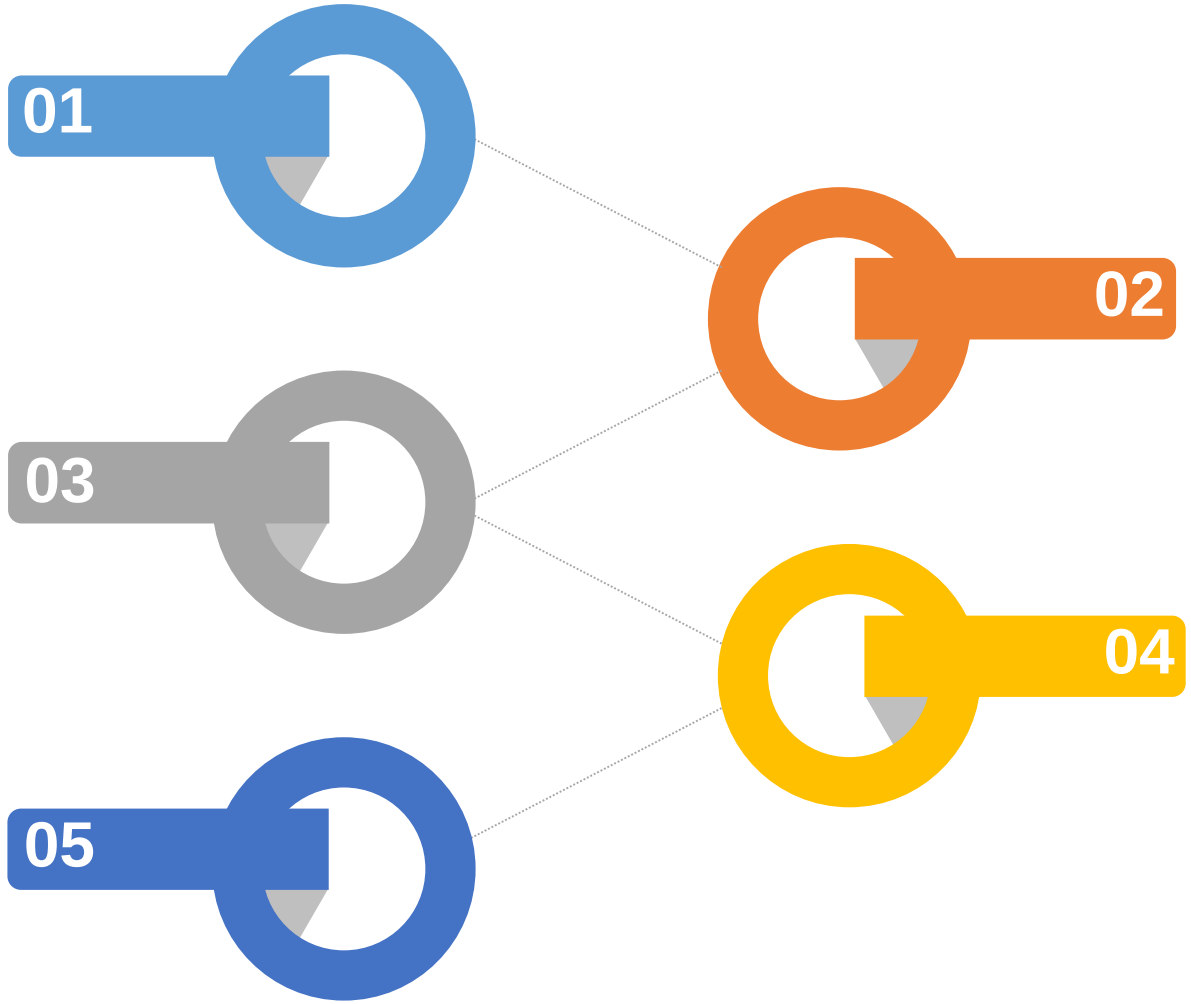
Pg_dump增加--load-via-partition-root、--no-comments参数
Pg_dumpall增加—encoding、--load-via-partition-root参数

弃用的子句

CREATE FUNCTION的WITH子句

系统表更改

删除了pg_class和pg_proc中的某些列



Libpq更改

libpq连接字符串中sslcompression的默认值已更改

不兼容函数

TO_NUMBER函数忽略模板中的分隔符。
TO_DATE / TO_NUMBER / TO_TIMESTAMP会跳过模板中的多字节字符

培训后补充

1. PG11之前，增加带有默认值的列需要重写表，此时建议为重写的表和索引预留约两倍的存储空间。
   详见PG10手册：https://www.postgresql.org/docs/10/sql-altertable.html
   Table and/or index rebuilds may take a significant amount of time for a large table; and will temporarily require as much as double the disk space.

THANKS!

2019

挑 战 一 切 不 可 能 ！

400-708-8006

www.highgo.com