<http://blog.csdn.net/zpf336/article/details/73809481>

# 典型使用场景

        随着使用时间的增加，数据库中的数据量也不断增加，因此数据库查询越来越慢。

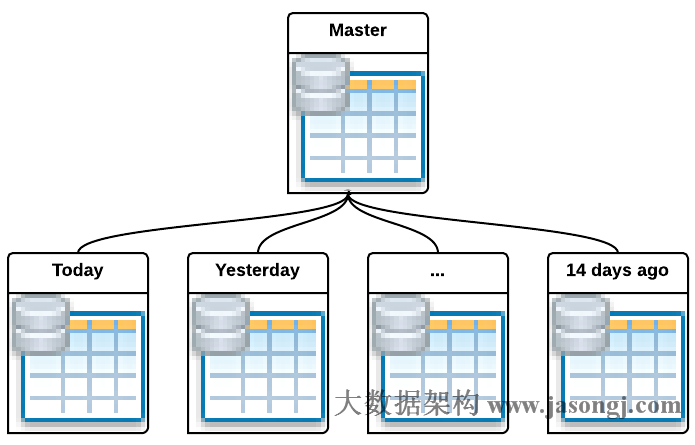
        加速数据库的方法很多，如添加特定的索引，将日志目录换到单独的磁盘分区，调整数据库引擎的参数等。这些方法都能将数据库的查询性能提高到一定程度。

        对于许多应用数据库来说，许多数据是历史数据并且随着时间的推移它们的重要性逐渐降低。如果能找到一个办法将这些可能不太重要的数据隐藏，数据库查询速度将会大幅提高。可以通过DELETE来达到此目的，但同时这些数据就永远不可用了。

        因此，需要一个高效的把历史数据从当前查询中隐藏起来并且不造成数据丢失的方法。本文即将介绍的数据库表分区即能达到此效果。

# 数据库表分区介绍

数据库表分区把一个大的物理表分成若干个小的物理表，并使得这些小物理表在逻辑上可以被当成一张表来使用。



## 数据库表分区术语介绍

* **主表/父表/Master Table**　该表是创建子表的模板。它是一个正常的普通表，但正常情况下它并不储存任何数据。
* **子表/分区表/Child Table/Partition Table**　这些表继承并属于一个主表。子表中存储所有的数据。主表与分区表属于一对多的关系，也就是说，一个主表包含多个分区表，而一个分区表只从属于一个主表

## 数据库表分区的优势

* 在特定场景下，查询性能极大提高，尤其是当大部分经常访问的数据记录在一个或少数几个分区表上时。表分区减小了索引的大小，并使得常访问的分区表的索引更容易保存于内存中。
* 当查询或者更新访问一个或少数几个分区表中的大部分数据时，可以通过顺序扫描该分区表而非使用大表索引来提高性能。
* 可通过添加或移除分区表来高效的批量增删数据。如可使用**ALTER TABLE NO INHERIT**可将特定分区从主逻辑表中移除（该表依然存在，并可单独使用，只是与主表不再有继承关系并无法再通过主表访问该分区表），或使用**DROP TABLE**直接将该分区表删除。这两种方式完全避免了使用**DELETE**时所需的**VACUUM**额外代价。
* 很少使用的数据可被迁移到便宜些的慢些的存储介质中

以上优势只有当表非常大的时候才能体现出来。一般来说，当表的大小超过数据库服务器的物理内存时以上优势才能体现出来

## PostgreSQL表分区

        现在PostgreSQL支持通过表继承来实现表的分区。父表是普通表并且正常情况下并不存储任何数据，它的存在只是为了代表整个数据集。PostgreSQL可实现如下两种表分区。

* **范围分区**　每个分区表包含一个或多个字段组合的一部分，并且每个分区表的范围互不重叠。比如可近日期范围分区
* **列表分区**　分区表显示列出其所包含的key值

# 表分区在PostgreSQL上的实现

## 在PostgreSQL中实现表分区的步骤

**1. 创建主表**  不用为该表定义任何检查限制，除非需要将该限制应用到所有的分区表中。同样也无需为该表创建任何索引和唯一限制。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** **TABLE** almart
2. (
3. date\_key **date**,
4. hour\_key **smallint**,
5. client\_key **integer**,
6. item\_key **integer**,
7. account **integer**,
8. expense **numeric**
9. );

**2.创建多个分区表**   每个分区表必须继承自主表，并且正常情况下都不要为这些分区表添加任何新的列。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** **TABLE** almart\_2015\_12\_10 () inherits (almart);
2. **CREATE** **TABLE** almart\_2015\_12\_11 () inherits (almart);
3. **CREATE** **TABLE** almart\_2015\_12\_12 () inherits (almart);
4. **CREATE** **TABLE** almart\_2015\_12\_13 () inherits (almart);

**3. 为分区表添加限制**。这些限制决定了该表所能允许保存的数据集范围。这里必须保证各个分区表之间的限制不能有重叠。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **ALTER** **TABLE** almart\_2015\_12\_10
2. **ADD** **CONSTRAINT** almart\_2015\_12\_10\_check\_date\_key
3. **CHECK** (date\_Key = '2015-12-10'::**date**);
4. **ALTER** **TABLE** almart\_2015\_12\_11
5. **ADD** **CONSTRAINT** almart\_2015\_12\_10\_check\_date\_key
6. **CHECK** (date\_Key = '2015-12-11'::**date**);
7. **ALTER** **TABLE** almart\_2015\_12\_12
8. **ADD** **CONSTRAINT** almart\_2015\_12\_10\_check\_date\_key
9. **CHECK** (date\_Key = '2015-12-12'::**date**);
10. **ALTER** **TABLE** almart\_2015\_12\_13
11. **ADD** **CONSTRAINT** almart\_2015\_12\_10\_check\_date\_key
12. **CHECK** (date\_Key = '2015-12-13'::**date**);

      **4. 为每一个分区表，在主要的列上创建索引**。该索引并不是严格必须创建的，但在大部分场景下，它都非常有用。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** **INDEX** almart\_date\_key\_2015\_12\_10
2. **ON** almart\_2015\_12\_10 (date\_key);
3. **CREATE** **INDEX** almart\_date\_key\_2015\_12\_11
4. **ON** almart\_2015\_12\_11 (date\_key);
5. **CREATE** **INDEX** almart\_date\_key\_2015\_12\_12
6. **ON** almart\_2015\_12\_12 (date\_key);
7. **CREATE** **INDEX** almart\_date\_key\_2015\_12\_13
8. **ON** almart\_2015\_12\_13 (date\_key);

     **5. 定义一个trigger或者rule把对主表的数据插入操作重定向到对应的分区表**。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. --创建分区函数
2. **CREATE** OR REPLACE **FUNCTION** almart\_partition\_trigger()
3. **RETURNS** **TRIGGER** **AS** $$
4. **BEGIN**
5. IF NEW.date\_key = **DATE** '2015-12-10'
6. **THEN**
7. **INSERT** **INTO** almart\_2015\_12\_10 **VALUES** (NEW.\*);
8. ELSIF NEW.date\_key = **DATE** '2015-12-11'
9. **THEN**
10. **INSERT** **INTO** almart\_2015\_12\_11 **VALUES** (NEW.\*);
11. ELSIF NEW.date\_key = **DATE** '2015-12-12'
12. **THEN**
13. **INSERT** **INTO** almart\_2015\_12\_12 **VALUES** (NEW.\*);
14. ELSIF NEW.date\_key = **DATE** '2015-12-13'
15. **THEN**
16. **INSERT** **INTO** almart\_2015\_12\_13 **VALUES** (NEW.\*);
17. ELSIF NEW.date\_key = **DATE** '2015-12-14'
18. **THEN**
19. **INSERT** **INTO** almart\_2015\_12\_14 **VALUES** (NEW.\*);
20. **END** IF;
21. **RETURN** NULL;
22. **END**;
23. $$
24. LANGUAGE plpgsql;
25. --挂载分区Trigger
26. **CREATE** **TRIGGER** insert\_almart\_partition\_trigger
27. BEFORE **INSERT** **ON** almart
28. **FOR** EACH ROW **EXECUTE** **PROCEDURE** almart\_partition\_trigger();

        **6. 确保postgresql.conf中的constraint\_exclusion配置项没有被disable**。这一点非常重要，如果该参数项被disable，则基于分区表的查询性能无法得到优化，甚至比不使用分区表直接使用索引性能更低。

## 表分区如何加速查询优化

        当constraint\_exclusion为on或者partition时，查询计划器会根据分区表的检查限制将对主表的查询限制在符合检查限制条件的分区表上，直接避免了对不符合条件的分区表的扫描。

        为了验证分区表的优势，这里创建一个与上文创建的almart结构一样的表almart\_all，并为其date\_key创建索引，向almart和almart\_all中插入同样的9000万条数据（数据的时间跨度为**2015-12-01到2015-12-30**）。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** **TABLE** almart\_all
2. (
3. date\_key **date**,
4. hour\_key **smallint**,
5. client\_key **integer**,
6. item\_key **integer**,
7. account **integer**,
8. expense **numeric**
9. );

        **1. 插入随机测试数据到almart\_all**

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **INSERT** **INTO** almart\_all
2. **select**
3. (**select**
4. array\_agg(i::**date**)
5. **from** generate\_series('2015-12-01'::**date**, '2015-12-30'::**date**, '1 day'::interval) **as** t(i)
6. )[floor(random()\*4)+1] **as** date\_key,
7. floor(random()\*24) **as** hour\_key,
8. floor(random()\*1000000)+1 **as** client\_key,
9. floor(random()\*100000)+1 **as** item\_key,
10. floor(random()\*20)+1 **as** account,
11. floor(random()\*10000)+1 **as** expense
12. **from**
13. generate\_series(1,300000000,1);

        **2. 插入同样的测试数据到almart**

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **INSERT** **INTO** almart **SELECT** \* **FROM** almart\_all;

      **3. 在almart和slmart\_all上执行同样的query，查询2015-12-15日不同client\_key的平均消费额。**

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. \timing
2. explain analyze
3. **select**
4. avg(expense)
5. **from**
6. (**select**
7. client\_key,
8. sum(expense) **as** expense
9. **from**
10. almart
11. **where**
12. date\_key = **date** '2015-12-15'
13. **group** **by** 1
14. )；
15. QUERY PLAN
16. -----------------------------------------------------------------------------------------------------------------------------------------------------------------------
17. Aggregate  (cost=19449.05..19449.06 **rows**=1 width=32) (actual **time**=9474.203..9474.203 **rows**=1 loops=1)
18. ->  HashAggregate  (cost=19196.10..19308.52 **rows**=11242 width=36) (actual **time**=8632.592..9114.973 **rows**=949825 loops=1)
19. ->  Append  (cost=0.00..19139.89 **rows**=11242 width=36) (actual **time**=4594.262..6091.630 **rows**=2997704 loops=1)
20. ->  Seq Scan **on** almart  (cost=0.00..0.00 **rows**=1 width=9) (actual **time**=0.002..0.002 **rows**=0 loops=1)
21. Filter: (date\_key = '2015-12-15'::**date**)
22. ->  Bitmap Heap Scan **on** almart\_2015\_12\_15  (cost=299.55..19139.89 **rows**=11241 width=36) (actual **time**=4594.258..5842.708 **rows**=2997704 loops=1)
23. Recheck Cond: (date\_key = '2015-12-15'::**date**)
24. ->  Bitmap **Index** Scan **on** almart\_date\_key\_2015\_12\_15  (cost=0.00..296.74 **rows**=11241 width=0) (actual **time**=4587.582..4587.582 **rows**=2997704 loops=1)
25. **Index** Cond: (date\_key = '2015-12-15'::**date**)
26. Total runtime: 9506.507 ms
27. (10 **rows**)
28. **Time**: 9692.352 ms
29. explain analyze
30. **select**
31. avg(expense)
32. **from**
33. (**select**
34. client\_key,
35. sum(expense) **as** expense
36. **from**
37. almart\_all
38. **where**
39. date\_key = **date** '2015-12-15'
40. **group** **by** 1
41. ) foo；
42. QUERY PLAN
43. ------------------------------------------------------------------------------------------------------------------------------------------------------------------
44. Aggregate  (cost=770294.11..770294.12 **rows**=1 width=32) (actual **time**=62959.917..62959.917 **rows**=1 loops=1)
45. ->  HashAggregate  (cost=769549.54..769880.46 **rows**=33092 width=9) (actual **time**=61694.564..62574.385 **rows**=949825 loops=1)
46. ->  Bitmap Heap Scan **on** almart\_all  (cost=55704.56..754669.55 **rows**=2975999 width=9) (actual **time**=919.941..56291.128 **rows**=2997704 loops=1)
47. Recheck Cond: (date\_key = '2015-12-15'::**date**)
48. ->  Bitmap **Index** Scan **on** almart\_all\_date\_key\_index  (cost=0.00..54960.56 **rows**=2975999 width=0) (actual **time**=677.741..677.741 **rows**=2997704 loops=1)
49. **Index** Cond: (date\_key = '2015-12-15'::**date**)
50. Total runtime: 62960.228 ms
51. (7 **rows**)
52. **Time**: 62970.269 ms

        由上可见，使用分区表时，所需时间为9.5秒，而不使用分区表时，耗时63秒。

        使用分区表，PostgreSQL跳过了除2015-12-15日分区表以外的分区表，只扫描2015-12-15的分区表。而不使用分区表只使用索引时，数据库要使用索引扫描整个数据库。另一方面，使用分区表时，每个表的索引是独立的，即每个分区表的索引都只针对一个小的分区表。而不使用分区表时，索引是建立在整个大表上的。数据量越大，索引的速度相对越慢。

## 管理分区

        从上文分区表的创建过程可以看出，分区表必须在相关数据插入之前创建好。在生产环境中，很难保证所需的分区表都已经被提前创建好。同时为了不让分区表过多，影响数据库性能，不能创建过多无用的分区表。

### 周期性创建分区表

        在生产环境中，经常需要周期性删除和创建一些分区表。一个经典的做法是使用定时任务。比如使用cronjob每天运行一次，将1年前的分区表删除，并创建第二天分区表（该表按天分区）。有时为了容错，会将之后一周的分区表全部创建出来。

### 动态创建分区表

        上述周期性创建分区表的方法在绝大部分情况下有效，但也只能在一定程度上容错。另外，上文所使用的分区函数，使用IF语句对date\_key进行判断，需要为每一个分区表准备一个IF语句。

        如插入date\_key分别为2015-12-10到2015-12-14的5条记录，前面4条均可插入成功，因为相应的分区表已经存在，但最后一条数据因为相应的分区表不存在而插入失败。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **INSERT** **INTO** almart(date\_key) **VALUES** ('2015-12-10');
2. **INSERT** 0 0
3. **INSERT** **INTO** almart(date\_key) **VALUES** ('2015-12-11');
4. **INSERT** 0 0
5. **INSERT** **INTO** almart(date\_key) **VALUES** ('2015-12-12');
6. **INSERT** 0 0
7. **INSERT** **INTO** almart(date\_key) **VALUES** ('2015-12-13');
8. **INSERT** 0 0
9. **INSERT** **INTO** almart(date\_key) **VALUES** ('2015-12-14');
10. ERROR:  relation "almart\_2015\_12\_14" does not exist
11. LINE 1: **INSERT** **INTO** almart\_2015\_12\_14 **VALUES** (NEW.\*)
12. ^
13. QUERY:  **INSERT** **INTO** almart\_2015\_12\_14 **VALUES** (NEW.\*)
14. CONTEXT:  PL/pgSQL **function** almart\_partition\_trigger() line 17 **at** SQL statement
15. **SELECT** \* **FROM** almart;
16. date\_key  | hour\_key | client\_key | item\_key | account | expense
17. ------------+----------+------------+----------+---------+---------
18. 2015-12-10 |          |            |          |         |
19. 2015-12-11 |          |            |          |         |
20. 2015-12-12 |          |            |          |         |
21. 2015-12-13 |          |            |          |         |
22. (4 **rows**)

        针对该问题，可使用动态SQL的方式进行数据路由，并通过获取将数据插入不存在的分区表产生的异常消息并动态创建分区表的方式保证分区表的可用性。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** OR REPLACE **FUNCTION** almart\_partition\_trigger()
2. **RETURNS** **TRIGGER** **AS** $$
3. **DECLARE** date\_text TEXT;
4. **DECLARE** insert\_statement TEXT;
5. **BEGIN**
6. **SELECT** to\_char(NEW.date\_key, 'YYYY\_MM\_DD') **INTO** date\_text;
7. insert\_statement := 'INSERT INTO almart\_'
8. || date\_text
9. ||' VALUES ($1.\*)';
10. **EXECUTE** insert\_statement USING NEW;
11. **RETURN** NULL;
12. EXCEPTION
13. **WHEN** UNDEFINED\_TABLE
14. **THEN**
15. **EXECUTE**
16. 'CREATE TABLE IF NOT EXISTS almart\_'
17. || date\_text
18. || '(CHECK (date\_key = '''
19. || date\_text
20. || ''')) INHERITS (almart)';
21. RAISE NOTICE 'CREATE NON-EXISTANT TABLE almart\_%', date\_text;
22. **EXECUTE**
23. 'CREATE INDEX almart\_date\_key\_'
24. || date\_text
25. || ' ON almart\_'
26. || date\_text
27. || '(date\_key)';
28. **EXECUTE** insert\_statement USING NEW;
29. **RETURN** NULL;
30. **END**;
31. $$
32. LANGUAGE plpgsql;

        使用该方法后，再次插入date\_key为2015-12-14的记录时，对应的分区表不存在，但会被自动创建。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **INSERT** **INTO** almart **VALUES**('2015-12-13'),('2015-12-14'),('2015-12-15');
2. NOTICE:  **CREATE** NON-EXISTANT **TABLE** almart\_2015\_12\_14
3. NOTICE:  **CREATE** NON-EXISTANT **TABLE** almart\_2015\_12\_15
4. **INSERT** 0 0
5. **SELECT** \* **FROM** almart;
6. date\_key  | hour\_key | client\_key | item\_key | account | expense
7. ------------+----------+------------+----------+---------+---------
8. 2015-12-10 |          |            |          |         |
9. 2015-12-11 |          |            |          |         |
10. 2015-12-12 |          |            |          |         |
11. 2015-12-13 |          |            |          |         |
12. 2015-12-13 |          |            |          |         |
13. 2015-12-14 |          |            |          |         |
14. 2015-12-15 |          |            |          |         |
15. (7 **rows**)

### 移除分区表

        虽然如上文所述，分区表的使用可以跳过扫描不必要的分区表从而提高查询速度。但由于服务器磁盘的限制，不可能无限制存储所有数据，经常需要周期性删除过期数据，如删除5年前的数据。如果使用传统的DELETE，删除速度慢，并且由于**DELETE**只是将相应数据标记为删除状态，不会将数据从磁盘删除，需要使用**VACUUM**释放磁盘，从而引入额外负载。

        而在使用分区表的条件下，可以通过直接DROP过期分区表的方式快速方便地移除过期数据。如

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **DROP** **TABLE** almart\_2014\_12\_15;

        另外，无论使用DELETE还是DROP，都会将数据完全删除，即使有需要也无法再次使用。因此还有另外一种方式，即更改过期的分区表，解除其与主表的继承关系，如。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **ALTER** **TABLE** almart\_2015\_12\_15 **NO** INHERIT almart;

        但该方法并未释放磁盘。此时可通过更改该分区表，使其属于其它TABLESPACE，同时将该TABLESPACE的目录设置为其它磁盘分区上的目录，从而释放主表所在的磁盘。同时，如果之后还需要再次使用该“过期”数据，只需更改该分区表，使其再次与主表形成继承关系。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** TABLESPACE cheap\_table\_space LOCATION '/data/cheap\_disk';
2. **ALTER** **TABLE** almart\_2014\_12\_15 **SET** TABLESPACE cheap\_table\_space;

## PostgreSQL表分区的其它方式

        除了使用Trigger外，可以使用Rule将对主表的插入请求重定向到对应的子表。如

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** **RULE** almart\_rule\_2015\_12\_31 **AS**
2. **ON** **INSERT** **TO** almart
3. **WHERE**
4. date\_key = **DATE** '2015-12-31'
5. DO **INSTEAD**
6. **INSERT** **INTO** almart\_2015\_12\_31 **VALUES** (NEW.\*);

        与Trigger相比，Rule会带来更大的额外开销，但每个请求只造成一次开销而非每条数据都引入一次开销，所以该方法对大批量的数据插入操作更具优势。然而，实际上在绝大部分场景下，Trigger比Rule的效率更高。

**同时，COPY操作会忽略Rule，而可以正常触发Trigger。**

        另外，如果使用Rule方式，没有比较简单的方法处理没有被Rule覆盖到的插入操作。此时该数据会被插入到主表中而不会报错，从而无法有效利用表分区的优势。

        除了使用表继承外，还可使用UNION ALL的方式达到表分区的效果。

**[sql]** [view plain](http://blog.csdn.net/zpf336/article/details/73809481) [copy](http://blog.csdn.net/zpf336/article/details/73809481)

1. **CREATE** **VIEW** almart **AS**
2. **SELECT** \* **FROM** almart\_2015\_12\_10
3. **UNION** ALL
4. **SELECT** \* **FROM** almart\_2015\_12\_11
5. **UNION** ALL
6. **SELECT** \* **FROM** almart\_2015\_12\_12
7. ...
8. **UNION** ALL
9. **SELECT** \* **FROM** almart\_2015\_12\_30;

        当有新的分区表时，需要更新该View。实践中，与使用表继承相比，一般不推荐使用该方法。

# 总结

* 如果要充分使用分区表的查询优势，必须使用分区时的字段作为过滤条件
* 分区字段被用作过滤条件时，WHERE语句只能包含常量而不能使用参数化的表达式，因为这些表达式只有在运行时才能确定其值，而planner在真正执行query之前无法判定哪些分区表应该被使用
* 跳过不符合条件分区表是通过planner根据分区表的检查限制条件实现的，而非通过索引
* 必须将constraint\_exclusion设置为ON或Partition，否则planner将无法正常跳过不符合条件的分区表，也即无法发挥表分区的优势
* 除了在查询上的优势，分区表的使用，也可提高删除旧数据的性能
* 为了充分利用分区表的优势，应该保证各分区表的检查限制条件互斥，但目前并无自动化的方式来保证这一点。因此使用代码动态创建或者修改分区表比手工操作更安全
* 在更新数据集时，如果使得partition key column(s)变化到需要使某些数据移动到其它分区，则该更新操作会因为检查限制的存在而失败。如果一定要处理这种情景，可以使用更新Trigger，但这会使得结构变得复杂。
* 大量的分区表会极大地增加查询计划时间。表分区在多达几百个分区表时能很好地发挥优势，但不要使用多达几千个分区表。

**参考：**[SQL优化（三） PostgreSQL Table Partitioning](http://www.jasongj.com/2015/12/13/SQL3_partition/)