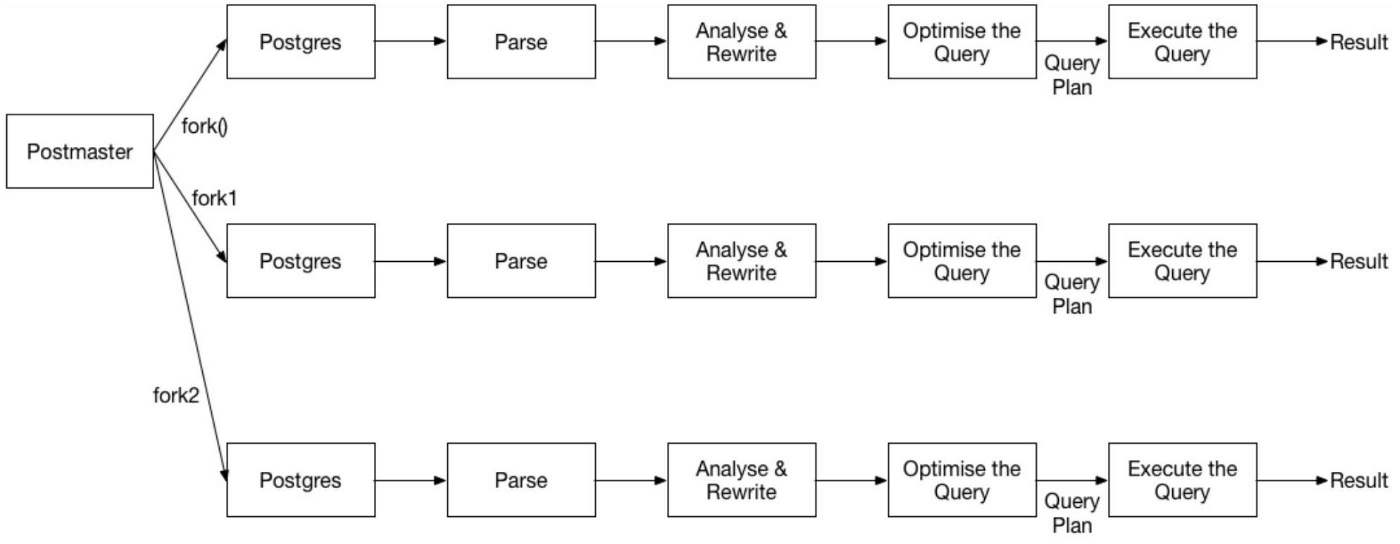
Postgres Indexes

How does Postgres work



Postgres Query Plans

两种基本查询方式

- Scan
- Join

Scan

- Seq Scan
- Index Scan
- Bitmap heap / index scan

Seq Scan: scan sequentially

```
sso=> explain select * from django_migrations;
                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on django_migrations  (cost=0.00..7.92 rows=392 width=43)
(1 row)

sso=> explain analyze select * from django_migrations;
                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Seq Scan on django_migrations  (cost=0.00..7.92 rows=392 width=43) (actual time=0.509..1.075 rows=402 loops=1)
 Planning time: 0.019 ms
 Execution time: 1.163 ms
(3 rows)
```

Side note: explain vs explain analyze

Explain: run the query planner, but do NOT run the query.

Explain analyze: run the actual query.

Side note: cost and actual cost

(**cost=0.00..7.92** rows=392 width=43)
(**actual time=0.509..1.075** rows=402 loops=1)

0.00: estimated startup cost (多久可以开始输出?)

7.92: estimated total cost (多久可以完成查询?)

```
sso=> explain (format json) select * from django_migrations;
                 QUERY PLAN
---------------------------------------------
 [                                          +
   {                                        +
     "Plan": {                              +
       "Node Type": "Seq Scan",             +
       "Parallel Aware": false,             +
       "Relation Name": "django_migrations",+
       "Alias": "django_migrations",        +
       "Startup Cost": 0.00,                +
       "Total Cost": 7.92,                  +
       "Plan Rows": 392,                    +
       "Plan Width": 43                     +
     }                                      +
   }                                        +
 ]
(1 row)
```

Side note: row and width

(cost=0.00..7.92 **rows=392 width=43**)

Rows: 估算出的输出行数

Width: 估算出的平均输出宽度（美行有多少字节?）

Index

Postgres: btree, hash, GIN, GIST index

- Btree: Most common, supports most operations
- Hash: Less common, only support "=" (hash comparison, duh!)
- GiST: Framework to implement many index types, most commonly GIS related (PostGIS)
- GIN: Inverted indexes, commonly used to index on JSON (see Postgres JSONB Field 对于 index, query 的支持)

Homework: read the Wiki page.

Index scan example

```
sso=> explain analyze select user_id from login_userprofile where id = 1;

 Index Scan using login_userprofile_pkey on login_userprofile
(cost=0.57..2.58 rows=1 width=4) (actual time=0.715..0.716 rows=1 loops=1)

   Index Cond: (id = 1)

 Planning time: 0.295 ms

 Execution time: 0.730 ms

(4 rows)
```

Index and memory cache

索引在内存中命中则可大幅度加速查询。

20k rows speed:

Cold query: Index scan ~ 10ms, Seq scan ~ 30ms

Hot query: Index scan ~ 0.5ms, Seq scan ~ 7ms

Jeff Dean's Rule of Thumb

L1 cache reference	0.5	ns					
Branch mispredict	5	ns					
L2 cache reference	7	ns					14x L1 cache
Mutex lock/unlock	25	ns					
Main memory reference	100	ns					20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000	ns	3	us			
Send 1K bytes over 1 Gbps network	10,000	ns	10	us			
Read 4K randomly from SSD*	150,000	ns	150	us			~1GB/sec SSD
Read 1 MB sequentially from memory	**250,000**	**ns**	**250**	**us**			
Round trip within same datacenter	500,000	ns	500	us			
Read 1 MB sequentially from SSD*	**1,000,000**	**ns**	**1,000**	**us**	**1**	**ms**	**~1GB/sec SSD, 4X memory**
Disk seek	10,000,000	ns	10,000	us	10	ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	**20,000,000**	**ns**	**20,000**	**us**	**20**	**ms**	**80x memory, 20X SSD**
Send packet CA->Netherlands->CA	150,000,000	ns	150,000	us	150	ms	

「差距似乎不大」

前提: 有足够的内存缓存所有的数据。

北京 OD RDS m4 系列 Postgres: 1700 RMB / (GB * 年) 内存

Name	API Name	Memory	Storage	vCPUs	PostgreSQL On Demand cost	PostgreSQL Reserved cost
Search	Search	Search	Search	Search	Search	Search
M5 Large	db.m5.large	8 GiB	0 GiB (EBS only)	2 vCPUs	$1559.280000 annually	$932.000000 annually
M5 Extra Large	db.m5.xlarge	16 GiB	0 GiB (EBS only)	4 vCPUs	$3118.560000 annually	$1864.000000 annually
M5 Double Extra Large	db.m5.2xlarge	32 GiB	0 GiB (EBS only)	8 vCPUs	$6237.120000 annually	$3729.000000 annually
M5 Quadruple Extra Large	db.m5.4xlarge	64 GiB	0 GiB (EBS only)	16 vCPUs	$12474.240000 annually	$7457.000000 annually
M5 12xlarge	db.m5.12xlarge	192 GiB	0 GiB (EBS only)	48 vCPUs	$37422.720000 annually	$22371.000000 annually
M5 24xlarge	db.m5.24xlarge	384 GiB	0 GiB (EBS only)	96 vCPUs	$74845.440000 annually	$44743.000000 annually

When index does not work...

```
sso=> explain select user_id from login_userprofile where id > 100;

                                  QUERY PLAN

------------------------------------------------------------------------------

 Seq Scan on login_userprofile  (cost=0.00..2450264.00 rows=78142304 width=4)

   Filter: (id > 100)

(2 rows)
```

Why???

Index 是另一种间接关系。

查询 Index 后仍需取出对应值，代价高于直接进行 sequential scan.

An working example

```
sso=> explain select user_id from login_userprofile where id < 100;
                                            QUERY PLAN
----------------------------------------------------------------------------------------------
 Index Scan using login_userprofile_pkey on login_userprofile  (cost=0.57..18.58 rows=96 width=4)
   Index Cond: (id < 100)
(2 rows)
```

Query planner settings

- set enable_bitmapscan=off
- set enable_indexscan=off
- set enable_indexonlyscan=off

	What?	Default cost
`seq_page_cost`	Read single database page from disk	**1.0**
`random_page_cost`	Read when rows involved are scattered randomly across disks	**4.0**
`cpu_tuple_cost`	Process a row of data	**0.01**
`cpu_index_tuple_cost`	Process an index entry during an index scan	**0.005**
`cpu_operator_cost`	Process an operator or function	**0.025**

JOIN: Nested Loop + Seq Scan

- 在循环中共进行 m*n 次查询，速度极慢。对双方 JOIN Key 建立索引**可能**可以解决。

```
SELECT "login_userprofile"."weixin_uid"  FROM   "weixin_wxuser"

       inner join "auth_user"

               ON ( "weixin_wxuser"."user_id" = "auth_user"."id" )

       left outer join "login_userprofile"

                    ON ( "auth_user"."id" = "login_userprofile"."user_id" )

WHERE  ( "weixin_wxuser"."enterprise_id" = %s

         AND "weixin_wxuser"."enterprise_associated" = %s )

ORDER  BY "weixin_wxuser"."user_id" DESC
```

JOIN: Nested Loop + Inner Index Scan

对 inner loop 的 cardinity 估算失误则可能会导致 nested loop + inner index scan 估算出错。

explain select auth_user.first_name from auth_user, login_userprofile where login_userprofile.id = 1 and auth_user.id = login_userprofile.user_id;

```
                                        QUERY PLAN
------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=1.14..5.18 rows=1 width=11)
   ->  Index Scan using login_userprofile_pkey on login_userprofile  (cost=0.57..2.58 rows=1 width=4)
         Index Cond: (id = 1)
   ->  Index Scan using auth_user_pkey on auth_user  (cost=0.57..2.58 rows=1 width=15)
         Index Cond: (id = login_userprofile.user_id)
(5 rows)
```

JOIN: Merge Join

仅限 = 的情况下使用merge join. 需要 = 的 key 上有 index.

（若无 index 就会是 seq scan）

```
sso=> explain select auth_user.first_name from auth_user, login_userprofile where auth_user.id = login_userprofile.user_id;
                                                        QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=138.79..6239912.38 rows=78142400 width=11)
   Merge Cond: (auth_user.id = login_userprofile.user_id)
   ->  Index Scan using auth_user_pkey on auth_user  (cost=0.57..3620243.81 rows=78249816 width=15)
   ->  Index Only Scan using login_userprofile_user_id_key on login_userprofile  (cost=0.57..1484600.57 rows=78142400 width=4)
(4 rows)
```

Just index everything?

NO.

https://raw.githubusercontent.com/pgexperts/pgx_scripts/master/indexes/needed_indexes.sql

https://raw.githubusercontent.com/pgexperts/pgx_scripts/master/indexes/unused_indexes.sql

- Memory constraint
- Computational cost (on write)

两个实际案例

Case: Cardinity, Index, and Limit

explain select * from hydrogen_cloudfunctionjob where cloud_function_id = 4831 order by created_at desc limit 1;

- cloud_function_id cardinity 不平衡（某些函数执行次数极高，某些函数执行次数极低）
- 表极大，写入速度快

后果

- Analyze 得出的结果不能反应实际情况
- Index 失效

Index 为什么失效?

- Query planner 数据来源: reltuples, relpages

```
sso=> SELECT relname, relkind, reltuples, relpages
sso-> FROM pg_class
sso-> WHERE relname LIKE 'hydrogen_cloudfunctionjob%';
                       relname                      | relkind |  reltuples   | relpages
----------------------------------------------------+---------+--------------+----------
 hydrogen_cloudfunctionjob                          | r       | 1.90963e+07 |   528940
 hydrogen_cloudfunctionjob_735f292e                 | i       | 1.91418e+07 |   154712
 hydrogen_cloudfunctionjob_cloud_function_id_created_at_idx | i       | 1.91419e+07 |    99036
 hydrogen_cloudfunctionjob_created_at               | i       | 1.91419e+07 |   137854
 hydrogen_cloudfunctionjob_id_seq                   | S       |            1 |        1
 hydrogen_cloudfunctionjob_pkey                     | i       | 1.91418e+07 |    88292
 hydrogen_cloudfunctionjob_ticketid                 | i       | 1.91418e+07 |   132389
(7 rows)
```

Query planer 更新数据的来源?

Auto vacuum, auto analyze

- autovacuum_analyze_threshold
- autovacuum_analyze_scale_factor
- autovacuum_vacuum_threshold
- autovacuum_vacuum_scale_factor

Threshold: 超过 X 后则**允许**执行

Scale factor: 超过 X 比例后则**开始**执行

Per table settings?

alter table hydrogen_cloudfunctionjob set (autovacuum_analyze_scale_factor = 0.00005);

alter table hydrogen_cloudfunctionjob set (autovacuum_analyze_threshold = 1000);

- Q: 能不能设置 autovacuum_analyze_threshold 更小？

Case: 2019-07-18 Postgres CPU 跑满

```
SELECT COUNT(*) AS "__count" FROM
"wpdata_articlecomment" WHERE
("wpdata_articlecomment"."tree_id" = %s
AND "wpdata_articlecomment"."lft" >= %s
AND "wpdata_articlecomment"."lft" <= %s
AND
"wpdata_articlecomment"."publish_status"
= %s)
```

/wpdata.api_v5:IfanrArticleCommentResource.get_resource

ACTION

Query

```
SELECT COUNT(*) AS "__count" FROM "wpdata_articlecomment" WHERE (
"wpdata_articlecomment"."tree_id" = %s AND
"wpdata_articlecomment"."lft" >= %s AND "wpdata_articlecomment"."lft"
<= %s AND "wpdata_articlecomment"."publish_status" = %s)
```

Explain plan

1	Query plan	Finalize Aggregate (cost=79448.81..79448.82 rows=1 width=8)
2	Query plan	-> Gather (cost=79448.39..79448.80 rows=4 width=8)
3	Query plan	Workers Planned: ?
4	Query plan	-> Partial Aggregate (cost=78448.39..78448.40 rows=1 width=8)
5	Query plan	-> Parallel Seq Scan on wpdata_articlecomment (cost=0.00..78446.41 rows =790 width=0)
6	Query plan	Filter: ?

Database instance

`sso.cwm2ouezvr7f.rds.cn-north-1.amazonaws.com.cn:5432`

Analyze?

```
Aggregate  (cost=79421.83..79421.84 rows=1 width=8) (actual time=55.611..55.611 rows=1 loops=1)
  ->  Gather  (cost=1000.00..79421.83 rows=1 width=0) (actual time=55.607..57.137 rows=0 loops=1)
        Workers Planned: 4
        Workers Launched: 4
        ->  Parallel Seq Scan on wpdata_articlecomment  (cost=0.00..78421.73 rows=1 width=0) (actual
time=53.036..53.036 rows=0 loops=5)
              Filter: ((lft >= 1105) AND (lft <= 1105) AND (tree_id = 1105) AND ((publish_status)::text =
'approved'::text))
              Rows Removed by Filter: 163254
Planning time: 0.426 ms
Execution time: 57.183 ms
```

并不缺 Index

```
Indexes:
   "wpdata_articlecomment_pkey" PRIMARY KEY, btree (id)
   "wpdata_articlecomment_6be37982" btree (parent_id)
   "wpdata_articlecomment_a00c1b00" btree (article_id)
   "wpdata_articlecomment_comment_type_a1142160" btree (comment_type)
   "wpdata_articlecomment_comment_type_a1142160_like" btree (comment_type varchar_pattern_ops)
   "wpdata_articlecomment_e8701ad4" btree (user_id)
   "wpdata_articlecomment_publish_status_idx" btree (publish_status)
   "wpdata_articlecomment_root_688afa41_uniq" btree (root)
```

为什么没命中 Index?

因为 wpdata_articlecomment_publish_status_idx 的 cardinity 太低.

```
sso=> select distinct publish_status from wpdata_articlecomment;
 publish_status
----------------
 0
 approved
 deleted
 invalid
 pending
 post-trashed
 trash
 unapproved
(8 rows)
```

为什么没命中 Index?

Query planner 认为按照 publish_status filter 后 bitmap heap scan (recheck condition: tree_id, lft_id) 的代价远远高于 seq scan 的代价。

为什么很高的 CPU?

因为 QPS 很高。

Common Problems & Solutions

- Very high CPU usage
  - Missing index?
  - Really high QPS?
- Very high memory usage
  - Missing index?
- Slow query
  - Explain, create index
  - Partitioning

Slow Query Logs

Set RDS settings for slow query logs.
<https://console.amazonaws.cn/rds/home?region=cn-north-1#parameter-groups-detail:ids=sso-pg96;type=DbParameterGroup;editing=false>

Name: log_min_duration_statement

保存即可，不会导致数据库重启。

Analyze slow query logs

Pgbadger: https://github.com/darold/pgbadger

SSO setup:

```
./pgbadger --prefix '%t:%r:%u@%d:[%p]:' ~/Downloads/postgres.log
```