PostgreSQL中文社区

2021 PostgreSQL China Conference

主办 ： PostgreSQL 中文社区

# 第 11 届 PostgreSQL 中国技术大会

开源论道 × 数据驱动 × 共建数字化未来

# 华为集团IT PostgreSQL运用实践

陈华军

# 目录

- **流程IT去O与上云**
- 如何用好PostgreSQL
- 常见问题与对策
- 未来展望

# 流程IT-服务19万人的企业IT

**8500亿**RMB

**收入**

- 2B、2C、Cloud BU
- **xxx万**基站
- **1.4亿**部手机

**业务**

- **170**个国家
- **20多**个运营中心

**全球运营**

- **15**个研发中心
- **36**个联合创新中心

**全球研发**

- **19万**员工
- **6万**伙伴
- **900**个办公地点

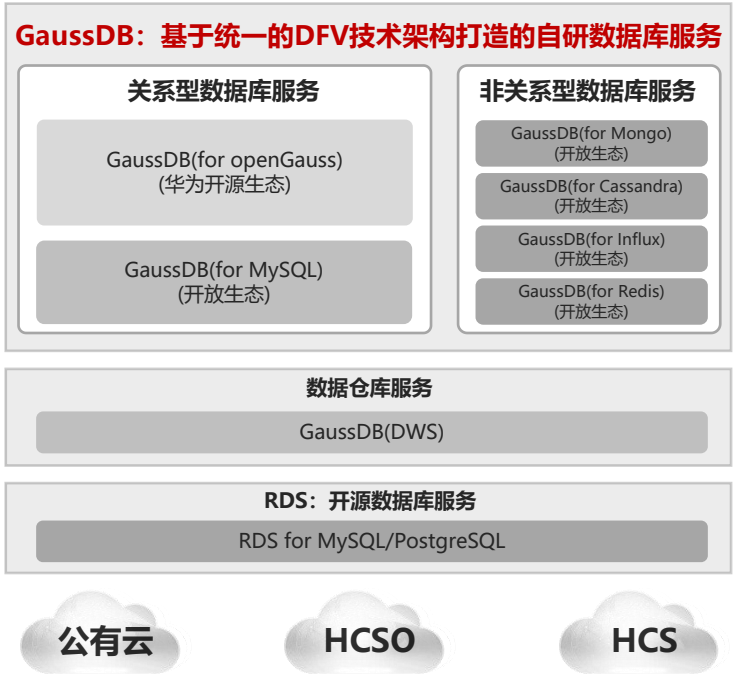
**全球协同**

提升客户业务能力,提升产品能力

要致富、先修路

| 投入 | 持续变革 | 数据 | IT | 办公 |
| --- | --- | --- | --- | --- |
| 变革与IT投入 **xxxx**名 流程IT员工 | IPD+、ISC+、CRM+、IFS **大平台支撑精兵作战** | 年度处理xxx万合同 **x.xx**亿行PO 年增长50%以上 | **600+**IT应用 分布全球的IT基础设施 | 每天发**200万**封邮件，**5万**次会议，每天跟外部第三方会议超过**400**次 |

# 流程IT数据库去O上云进展




2020年开始华为集团内部的数据库从线下自建库(Oracle,PG,MySQL)迁移到华为云RDS服务，并大规模采用PG去O，目前去O和上云工作已完成一半

**华为云MySQL：**

| 类别 | 华为云-MySQL |
|---|---|
| 实例数 | 8000+ |
| CPU | 10W U+ |
| 内存 | TB级 |
| 存储 | PB级 |

**华为云PostgreSQL：**

| 类别 | 华为云-PG |
|---|---|
| 实例数 | 4000+ |
| CPU | 7w U+ |
| 内存 | TB级 |
| 存储 | PB级 |

*上述为集团内部迁移到华为云的部分数据*

# 流程IT去O的困难与挑战

涉及ERP，制造，财经等关键的核心系统，对目标库的功能，性能和可靠性都有非常的要求

- 体量大
- 业务复杂
- 核心系统
- 时间紧无退路

# 交易库去O的目标库选型

- MySQL：华为商城等类互联网应用
- PostgreSQL/openGauss：ERP，制造，财经等企业应用

| 项目 | MySQL | PostgreSQL |
|---|---|---|
| **SQL 语法** | SQL特性支持36种，SQL语法支持比较弱 | SQL特性94种，SQL语法支持最完善 |
| **主从复制** | 逻辑复制 | 物理流复制、逻辑复制 |
| **复制安全性** | 5.7开始支持半同步 | 物理同步复制，强一致 |
| **分区表** | 支持 | 支持 |
| **物化视图** | 不支持 | 支持 |
| **索引类型** | 主要是btree 索引，不支持函数索引 | 多种索引类型，支持函数索引 |
| **并行查询** | 只支持主键并行 | 支持多种并行查询算法 |
| **业务场景倾向** | 类互联网应用（逻辑在业务层，最终一致性）场景 | 企业应用（财经、运营商、CRM、物联网等数据量较大，一致性要求高）场景 |

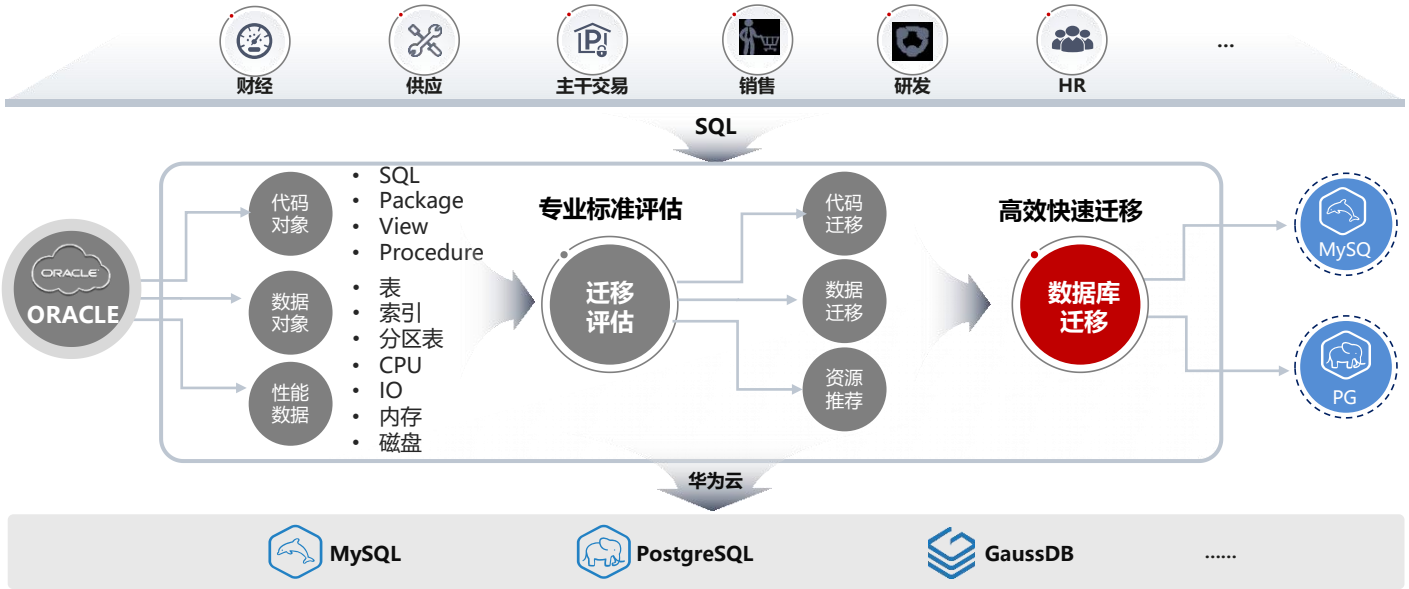
| | 数据库 | Oracle | MySQL | postgreSQL |
|---|---|---|---|---|
| 测试大类 | 用例名称 | 8C/32G | 4C/8G | 4C/8G |
| 单表访问 | 单表全表扫描 | 6.3s | 110s | 2s |
| | 单表索引等值 | 0.2s | 0.3s | 0.1s |
| | 单表索引范围 | 0.6s | 0.3s | 0.1s |
| 多表访问 | 多表走NL | 0.1s | 0.3s | 0.1s |
| | 多表走HASH | 7.6s | 无法跑出结果 | 4s |
| | 标量子查询 | 0.1s | 0.2s | 0.05s |
| | 内联视图 | 0.1s | 115s | 0.1s |
| DML | INSERT事务隔离 | 支持 | 支持 | 支持 |
| | INSERT行锁等待 | 支持 | 支持 | 支持 |
| | UPDATE事务 | 支持 | 支持 | 支持 |
| | UPDATE行锁等待 | 支持 | 支持 | 支持 |
| | DELETE事务 | 支持 | 支持 | 支持 |
| | DELETE行锁 | 支持 | 支持 | 支持 |
| 并发查询 | 并发测试 | 3分47秒 | 31分32秒 | 16分24秒 |
| | 平均每次耗时 | 12.7ms | 179.2ms | 88.4ms |
| 并发DML | 测试耗时 | 11分19秒 | 超过4个小时未完成 | 15分5秒 |
| 单线程写入 | 程序写(有程序开销) | 85秒 | 548秒 | 98秒 |
| | 存储过程写 | 0.8秒 | 20.3秒 | 0.2秒 |
| 数据库特性 | 唯一约束 | 支持 | 支持 | 支持 |
| | 分区表 | 范围、列表、哈希 | 范围、列表、哈希 | 范围、列表、哈希 |
| | 表长度限制 | 1GB | 65535字节 | 1GB |
| | 并行建索引 | 支持 | | 支持 |
| | 在线建索引 | 支持 | | 支持 |
| DDL | 添加字段 | 0.2秒 | 2分18秒 | 0.1秒 |
| | 新建索引 | 7秒 | 7分41秒 | 22秒 |

# 数据库去O准备工作：数据清理、应用解耦、服务化

| 序号 | 挑战 | 解决方案 |
| --- | --- | --- |
| 1 | **数据容量大，不能直接切换** | 1. 数据瘦身：识别关键资产，清理/归档数据<br>2. 在完成瘦身的基础上，再考虑是否需要分库分表 |
| 2 | **耦合严重，共schema** | 1. 按应用拆分Schema，先完成重构、剥离、拆分<br>2. 完成整改后，按微服务为单元一次性切换数据库 |
| 3 | **大量使用存储过程** | 1. 整改SQL：使用标准SQL，不使用数据库特殊FEATURE<br>2. 数据库作为存储层，Java实现业务逻辑层，不在数据库中写复杂逻辑SQL |
| 4 | **结合服务化改造，在切换过程中存在新老系统并存** | 1. 根据业务需要，再按领域/区域逐个搬迁<br>2. 如果下游未切换，则需要保持老系统数据是全量 |

# 数据库迁移：通过平台服务快速迁移到基于华为云的数据库生态

**基于成熟的工具和专家经验，为产品团队提供数据库引擎推荐、资源评估和迁移难度分析，评估效率从月至天。实施阶段通过平台和工具辅助代码迁移和数据迁移。**

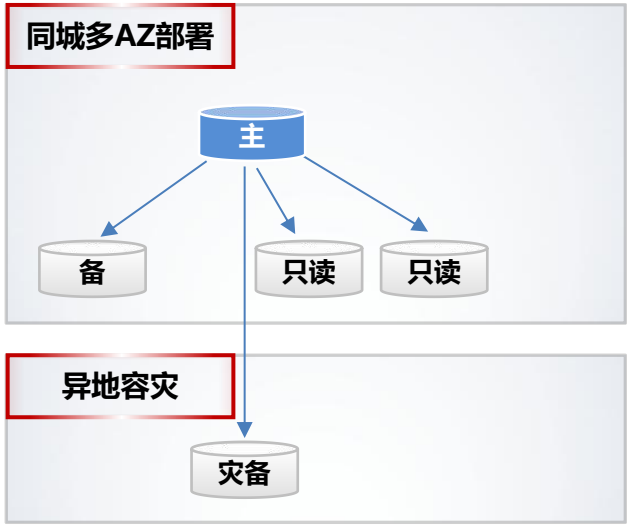


# 目录

- 流程IT去O与上云
- **用好PostgreSQL**
- 常见问题与对策
- 未来展望

# 部署架构

- 一主一备/一主两备 + 只读 + 异地容灾
- 同步复制+自动降级
- 基于JDBC多主机URL的读写分离




| 要点 | 描述 |
| --- | --- |
| 磁盘级高可用 | 自研分布式存储(EVS)采用3副本方式，单副本损坏，业务零感知。 |
| 反亲和性 | 同一实例的不同节点所在的VM创建在不同的物理机上，主机故障时只影响一个节点。 |
| 同城高可用 | 充分利用多AZ实现高可用方案，避免单个机房事故导致业务中断。 |
| 异地高可用 | 跨region灾备节点，满足容灾诉求。 |
| 自动故障切换 | 主节点故障，自动切备机，切VIP。 |
| 自动复制降级 | 配置为华为云PG最大可用模式时，备机故障，主库自动降级为异步复制，备机恢复时自动恢复同步复制。 |

# 参数调优

- 云上RDS的参数默认值都已经过调优，但个别参数仍需要用户根据场景定制。比如:

| 参数 | 说明 |
| --- | --- |
| lock_timeout | 当某个表上的长事务阻塞DDL时，这个表上的所有访问都会被阻塞，引发雪崩效应。<br>强烈建议设置合理的lock_timeout，比如1分钟或5分钟，及时中断阻塞。 |
| hot_standby_feedback | 可以在只读库上设置为on，避免只读库的查询和回放死元组清理WAL记录的后台任务发生冲突。<br>但是**必须同时做好只读库上长事务，未决2PC事务以及失效和滞后复制槽的监控**，并及时处理，否则会导致主库的表发生膨胀。 |

- 数据库初始化配置

```
CREATE DATABASE ${dbname}
    LC_COLLATE 'C'
    LC_CTYPE 'en_US.UTF-8'
    ENCODING 'UTF-8'
    TEMPLATE template0;
```

# 客户端配置（JDBC）

| 分类 | 配置 | 配置建议 | 备注 |
|---|---|---|---|
| 连接参数 | URL | jdbc:postgresql://${host}:5432/${dbname}?tcpKeepAlive=true&connectTimeout=5&defaultRowFetchSize=2000 | ■ 如遇到数据类型转换的兼容问题，可尝试设置stringtype=unspecified。设置stringtype=unspecified可使服务端更积极的做数据类型转换，兼容性更好。比如支持使用setString()对int字段赋值。<br>■ 测试环境调查问题，可临时设置“loggerLevel=DEBUG&loggerFile=/tmp/jdbc.log”打开详细日志 |
| 连接池 | 连接池组件 | 建议dbcp2，HikariCP。不建议使用druid | druid使用select 1对池中的空闲PG连接进行检活，容易产生空闲长事务；并且不支持连接生命周期控制 |
| | 最小连接数 | 建议设置为1，最大不超过5 | dbcp2：minIdle，maxIdle<br>HikariCP：minimumIdle |
| | 最大连接数 | ■ 所有应用连接池最大连接数合计不超过数据库最大连接数<br>■ 能满足业务需求的前提下，尽量设小 | dbcp2：maxTotal<br>HikariCP：maximumPoolSize |
| | 连接生命周期 | 务必设置以便及时释放会话级缓存和临时表（临时表会影响事务冻结），比如1小时 | dbcp2：maxConnLifetimeMillis<br>HikariCP：maxLifetime |

■ **TIPS:**

- 根据业务特征设置合理的查询超时（QueryTimeout）
- 大量数据更新，尽量使用JDBC批更新，合理控制单个批次更新的数据量，比如2000
- 尽量使用JDBC PraparedStatment，对简单SQL性能可提升1倍，并且可预防SQL注入
- PG缓存的执行计划消耗内存过大或不适用于所有的参数值时，在充分评估影响的前提下可考虑关闭服务端预编译(prepareThreshold=0)

# 监控告警

■ 除了常规的OS资源监控，数据库年龄，长事务等PG特有的监控也必不可少

**OS**

- CPU
- 内存
- IO
- 网络
- SWAP
- 存储空间

**DB**

- 生死(连通性)
- 连接数
- 活动连接数
- 数据库年龄
- 长事务存活时间
- 未决2PC事务存活时间
- 失效的复制槽
- 为复制槽保留的WAL大小
- 复制延迟

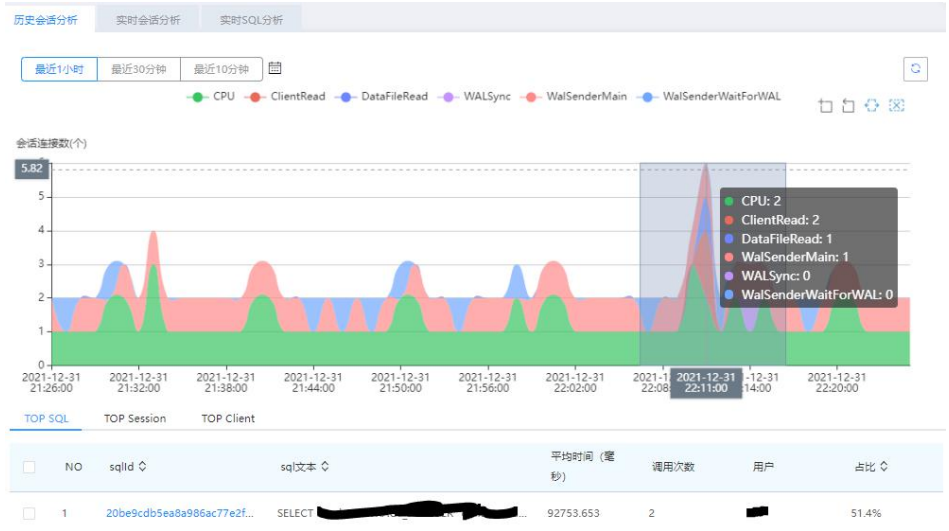
# 数据库性能诊断工具

**基于丰富的商业数据库运维经验，构建PG数据库性能诊断工具，支撑应用开发团队及时发现应用中的慢SQL、锁争用，提升应用健壮性和用户体验。**

| | | | |
|---|---|---|---|
| 锁分析 | 对象锁 | 阻塞锁 | KILL会话 |
| 会话分析 | 实时会话 | 历史会话 | TOP SQL |
| SQL分析 | SQL 查询 | 慢 SQL | SQL诊断 |

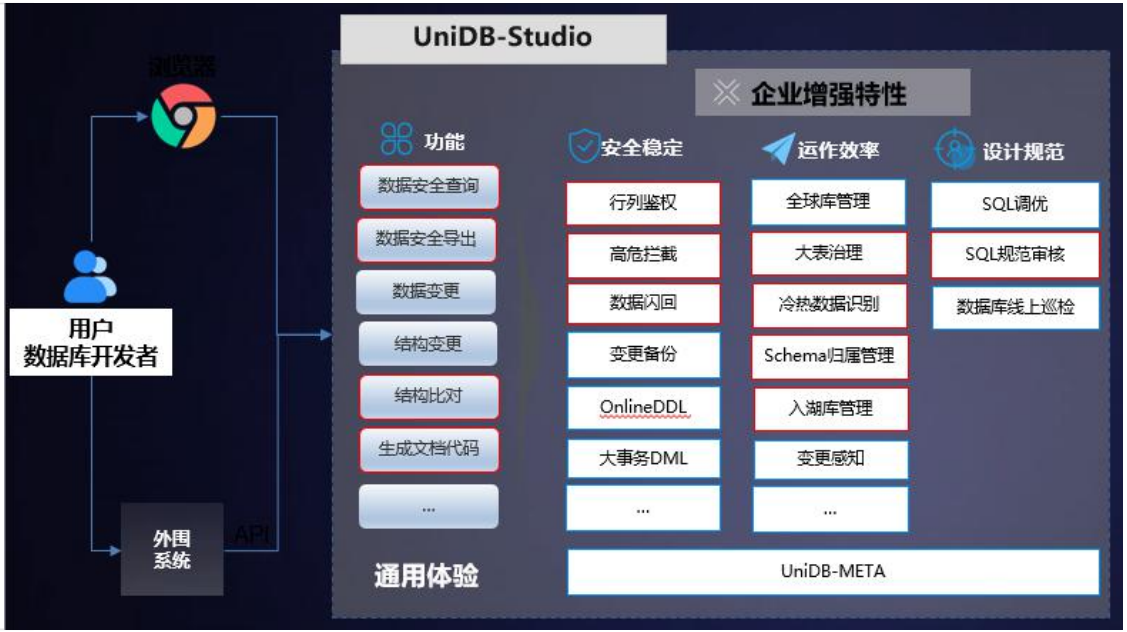
加工
采集
建模

| | |
|---|---|
| SQL及执行计划：<br>pg_stat_statements<br>auto_explain | 锁信息：pg_locks |
| | 会话信息：pg_stat_activity |



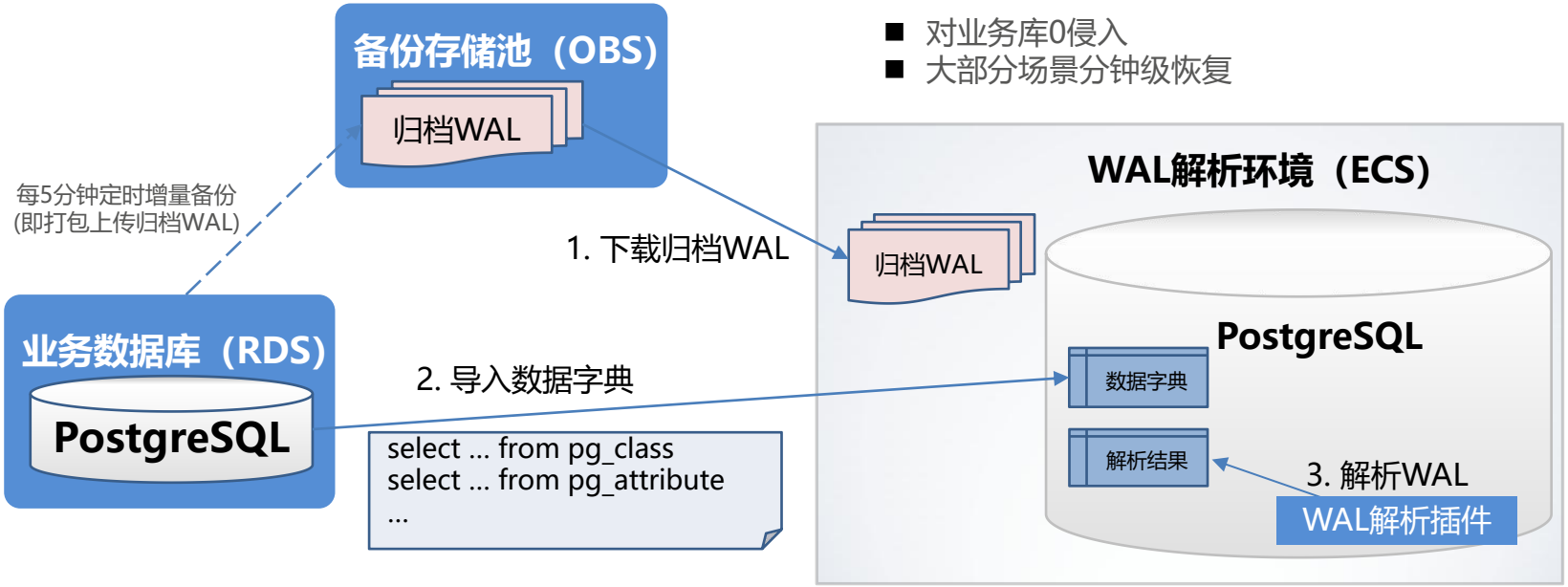
# 数据库开发者服务工具

面向企业数据库开发者，提供覆盖MySQL、PG、Gauss等10+数据库类型的查询、导出、变更等操作统一管理WEB IDE；融入了行列鉴权控制、防数据库高危操作、数据库设计规范、SQL规范、数据闪回、变更稳定性控制等企业安全稳定特性。



# PostgreSQL闪回实现

- 对业务库0侵入
- 大部分场景分钟级恢复



# 目录

- 流程IT去O与上云
- 用好PostgreSQL
- **常见问题与对策**
- 未来展望

# 问题1. 不合理的执行计划

- **大部分时候PG的优化器工作得很好，但某些特定场景下，容易产生不合理的执行计划。**

| 原因 | 对策 |
| --- | --- |
| **统计信息采集不及时** | ■ 手动执行analyze<br>■ 调整库级或表级参数autovacuum_analyze_scale_factor |
| **大表采样统计信息不准确** | ■ default_statistics_target参数调大<br>■ ALTER TABLE ... ALTER COLUMN ... SET STATISTICS<br>■ ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...) |
| **多字段条件组合的选择率估算不准确** | ■ CREATE STATISTICS ... |
| **缓存的执行计划不适用所有代入参数** | ■ 代码中不使用预编译语句<br>■ 数据库中设置plan_cache_mode为force_custom_plan<br>■ JDBC连接字符串中设置参数prepareThreshold=0 |
| **优化器算法缺陷** | ■ pg_hint_plan<br>■ SQL改写 |

# 例1：LIMIT子句代价估算偏差

- **表定义&数据分布**

```
create table tb1(id int primary key,c1 int);
create index on tb1(c1);
insert into tb1 select id, id/10000 from generate_series(1,10000000)id;
```

- **SQL&执行计划**

```
postgres=# explain analyze select * from tb1 where c1=999 order by id limit 10;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
Limit (cost=0.43..332.29 rows=10 width=8) (actual time=1571.315..1571.319 rows=10 loops=1)
  ->  Index Scan using tb1_pkey on tb1  (cost=0.43..328935.03 rows=9912 width=8) (actual time=1571.314..1571.316 rows=10
loops=1)
        Filter: (c1 = 999)
        Rows Removed by Filter: 9989999
 Planning Time: 0.112 ms
 Execution Time: 1571.337 ms
(6 rows)
```

**上面Index Scan估算的行数和cost都比较准确，但评估LIMIT子句时，优化器假设数据分布是均匀的， 只需扫描主键索引的10/9912即可找到10条匹配的记录，最终的估算代价也被LIMIT降到10/9921。但实际上满足条件的记录都集中在索引的尾部。**

# SQL改写优化

- **SQL改写方法1：破坏索引排序**

```
select * from tb1 where c1=999 order by id+0 limit 10;
```

- **SQL改写方法2：物化子查询**

```
WITH t AS MATERIALIZED(
  select * from tb1 where c1=999
)
select * from t order by id limit 10
```

**改写后，本例中的SQL执行时间从1571ms减少到4ms**

# 例2：多个Join条件组合

- **表定义&数据分布**

```
create table tb1(c1 int,c2 int,d int);
create table tb2(c1 int,c2 int);
create table tb3(d int);
create index on tb3(d);
insert into tb1 select id, id, id/100 from generate_series(1,100000)id;
insert into tb2 select id, id from generate_series(1,100000)id;
insert into tb3 select id from generate_series(1,1000)id;
```

- **查询SQL：**

```
select count(*) from tb1,tb2,tb3
  where tb1.c1=tb2.c1 and tb1.c2=tb2.c2
        and tb1.d=tb3.d
```

## 执行计划：

```
Aggregate  (cost=7021.51..7021.52 rows=1 width=8) (actual time=18448.825..18448.828 rows=1 loops=1)
  ->  Nested Loop  (cost=3334.00..7021.51 rows=1 width=0) (actual time=38.280..18436.881 rows=99901 loops=1)
        Join Filter: (tb1.d = tb3.d)
        Rows Removed by Join Filter: 99900099
        ->  Hash Join  (cost=3334.00..6994.01 rows=1 width=4) (actual time=29.513..155.524 rows=100000 loops=1)
              Hash Cond: ((tb1.c1 = tb2.c1) AND (tb1.c2 = tb2.c2))
              ->  Seq Scan on tb1  (cost=0.00..1541.00 rows=100000 width=12) (actual time=0.006..10.360 rows=100000 loops=1)
              ->  Hash  (cost=1443.00..1443.00 rows=100000 width=8) (actual time=29.408..29.409 rows=100000 loops=1)
                    Buckets: 131072  Batches: 2  Memory Usage: 2982kB
                    ->  Seq Scan on tb2  (cost=0.00..1443.00 rows=100000 width=8) (actual time=0.005..7.975 rows=100000 loops=1)
        ->  Seq Scan on tb3  (cost=0.00..15.00 rows=1000 width=4) (actual time=0.003..0.077 rows=1000 loops=100000)
 Planning Time: 0.166 ms
 Execution Time: 18448.880 ms
(13 rows)
```

**上面对JOIN结果集行数的估算出现巨大偏差，估算1行，实际10w行，并导致对tb3执行了10w次全表扫描。**
**出现偏差的原因是：PG优化器假设2个Join条件对结果的过滤效果是互相独立的。**

Join结果集行数 = 2表的笛卡尔积 * Join条件1的选择率 * Join条件2的选择率

## SQL改写优化：

**把Join条件改成某些逻辑上的等价形式，影响行估算，进而改变执行计划。比如**

tb1.c2=tb2.c2
==>
tb1.c2+0=tb2.c2+0

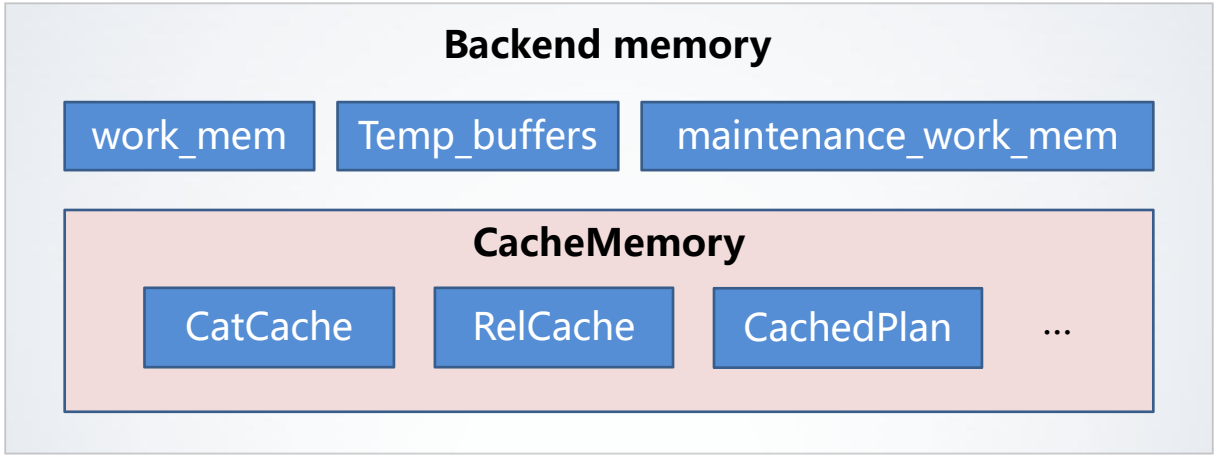
**修改后的SQL如下。SQL改写后使用hash join，执行时间从18秒缩短到159ms**

```
Aggregate  (cost=7284.62..7284.64 rows=1 width=8) (actual time=158.687..158.690 rows=1 loops=1)
  ->  Hash Join  (cost=3361.50..7283.38 rows=500 width=0) (actual time=43.258..151.051 rows=99901 loops=1)
      Hash Cond: (tb1.d = tb3.d)
      ->  Hash Join  (cost=3334.00..7249.00 rows=500 width=4) (actual time=42.365..127.613 rows=100000 loops=1)
          Hash Cond: ((tb1.c1 = tb2.c1) AND ((tb1.c2 + 0) = (tb2.c2 + 0)))
          ->  Seq Scan on tb1  (cost=0.00..1541.00 rows=100000 width=12) (actual time=0.022..12.525 rows=100000 loops=1)
          ->  Hash  (cost=1443.00..1443.00 rows=100000 width=8) (actual time=41.936..41.937 rows=100000 loops=1)
              Buckets: 131072  Batches: 2  Memory Usage: 2982kB
              ->  Seq Scan on tb2  (cost=0.00..1443.00 rows=100000 width=8) (actual time=0.046..11.682 rows=100000 loops=1)
      ->  Hash  (cost=15.00..15.00 rows=1000 width=4) (actual time=0.538..0.538 rows=1000 loops=1)
          Buckets: 1024  Batches: 1  Memory Usage: 44kB
          ->  Seq Scan on tb3  (cost=0.00..15.00 rows=1000 width=4) (actual time=0.047..0.248 rows=1000 loops=1)
Planning Time: 0.306 ms
Execution Time: 159.037 ms
```

本例中tb1.c2=tb2.c2基于统计信息估算出的选择率是1/100000；tb1.c2+0=tb2.c2+0无法利用统计信息，使用固定的选择率1/200
PG代价估算算法可参考：http://www.interdb.jp/pg/pgsql03.html

# 问题2. OOM

- PostgreSQL每个会话对应一个进程，且每个后端进程缓存表定义和执行计划，并发连接多时容易占用大量内存甚至触发OOM。



- PostgreSQL每个会话对应一个进程，且每个后端进程缓存表定义和执行计划，并发连接多时容易占用大量内存甚至触发OOM。

| 内存占用原因 | 对策 |
| --- | --- |
| 连接数过多 | ■ **调小应用端连接池大小**<br>■ **设置最大连接生命周期**<br>■ 拆分负载（读写分离，分库分表等）<br>■ 部署pgbouncer连接池 |
| 系统表缓存太大<br>(通常pg_statistic和pg_attribute占用内存最大) | ■ 避免不太分区字段的分区表访问SQL<br>■ 减少分区数<br>■ 连接隔离（不同业务单元使用不同连接池）<br>■ 通过ALTER TABLE SET STATISTICS对不需要通过频繁值和柱状图评估选择性的字段，减少收集的统计信息<br>■ 数据库端通过LRU算法限制缓存大小(华为云内核定制功能) |
| 执行计划缓存太大 | ■ 避免过多，过长的无法复用的预编译语句,比如in($1,$2... 参数多且个数不确定)<br>■ 客户端限制每个连接缓存的预编译语句数量(preparedStatementCacheQueries)<br>■ 关闭预编译语句(prepareThreshold=0)<br>■ 连接隔离（不同业务单元使用不同连接池）<br>■ 数据库端通过LRU算法限制缓存大小(华为云内核定制功能) |
| 页表项占用内存太大 | ■ 对大内存环境开启Huge Page |

# 问题3. 子事务的潜在风险

- **子事务对应的功能是SAVEPOINT，即使不显式使用SAVEPOINT，存储过程中的BEGIN ... EXCEPTION块会隐式的引入子事务。**

```
FOR i IN 1..1000 LOOP
  BEGIN
    INSERT INTO t VALUES (i);
  EXCEPTION  -- 每次循环产生一个子事务
    WHEN OTHERS THEN
      NULL;
  END;
END LOOP;
```

- **子事务的危害**
  - 事务ID快速增长，增加事务ID回卷风险
  - 使用逻辑订阅时，可能产生大量临时小文件
  - 单个会话中子事务数超过64后，进入**suboverflow**状态，高并发场景下可能导致数据库整体性能大幅下降

# 案例1. 子事务 + 逻辑订阅导致walsender卡死

**开启逻辑订阅时，walsender进程会在内存中保存每个活动事务的变更记录，但是当一个事务修改的记录数超过4096时，序列化变更记录到临时文件，同时顺带序列化该事务的所有子事务的变更记录。**

**数据准备：**

```
CREATE TABLE t(id int primary key);
CREATE TABLE t2(id int primary key);

CREATE OR REPLACE FUNCTION insert_data()
RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE i int;
BEGIN
   FOR i in 1..10000000 LOOP
   BEGIN
      INSERT INTO t(c1) VALUES(i);
   EXCEPTION
     WHEN UNIQUE_VIOLATION THEN
       NULL;
   END;
   END LOOP;
END
$$;

SELECT pg_create_logical_replication_slot
('test_slot','test_decoding');
```

```
pg_recvlogical --start -S test_slot -d postgres -f
/tmp/test_logical
```

**故障模拟：**

```
BEGIN;
SELECT insert_data(); -- walsender中缓存了1000w个子事务的变更记录
INSERT INTO t2 SELECT generate_series(1,5000); --子事务的变更记录落盘产生1000w个临时文件
```

1000w个子事务的变更记录落盘导致walsender进程长时间CPU打满，甚至可能撑爆磁盘，耗尽Inode。

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
68358 postgres  20   0 7075328   4.3g    924 R  97.0 56.2   2:39.36 postgres: walsender postgres [local] idle
```

每个子事务产生一个小临时文件，每个文件只有100多字节，但会占用4KB存储空间。

```
[postgres@[redacted] ~]$ ll /data01/data/pg_replslot/test_slot/|more
total 6876188
-rw------- 1 postgres postgres 176 Jan  3 22:27 state
-rw------- 1 postgres postgres 128 Jan  3 22:27 xid-38980405-lsn-7-F0000000.spill
-rw------- 1 postgres postgres 128 Jan  3 22:27 xid-38980406-lsn-7-F0000000.spill
-rw------- 1 postgres postgres 128 Jan  3 22:27 xid-38980407-lsn-7-F0000000.spill
```

# 案例2：子事务+高并发DML导致整库hang

**现象：** 数据库性能下降，且出现大量subtrans和SubtransControlLock锁(PG13以后为SubtransBuffer和SubtransSLRU锁)

**原因：** 1. 每个会话最多可以缓存64个子事务XID，超出64的部分将溢出到子事务文件(pg_subxact)

2. 只要有一个会话发生子事务溢出(pgxact->overflowed)，此期间采集的事务快照（suboverflowed）在判断元组可见性时，可能需要读取子事务文件。

3. 大量会话高并发频繁访问子事务文件，产生锁争用，严重时导致整库性能急剧下降

注：读写分离场景下，主库上子事务即使不发生溢出，备库上的查询也可能需要读取子事务文件。

**子事务溢出时的元组可见性判断**

- 读取元组的版本标识
- 获得subtrans轻量级锁
- 根据版本标识，获得其父事务ID
  - 如果涉及的xmin的子事务SLRU未在共享内存buffer中
    - 获得SubtransControlLock锁
    - 从subtrans文件中读取相应的页
    - 释放SubtransControlLock锁
  - 在共享内存buffer中
    - 读取相应的父事务ID值，读取父事务ID值，
- 释放subtrans锁

**建议：**谨慎使用子事务，特别是在大事务，长事务中

参考：

- https://www.cybertec-postgresql.com/en/subtransactions-and-performance-in-postgresql/
- https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/20
- https://www.postgresql.org/message-id/003201d79d7b$189141f0$49b3c5d0$@tju.edu.cn

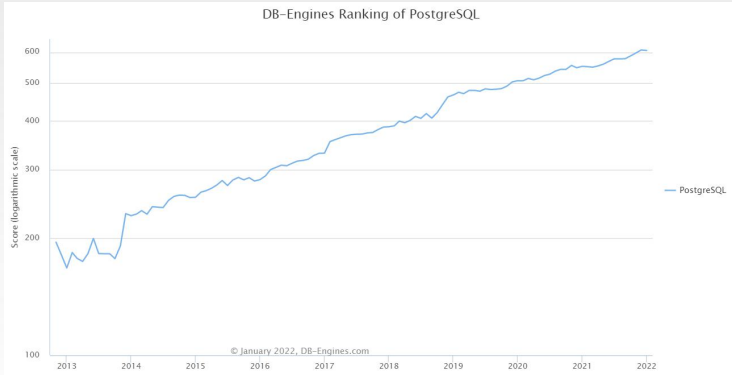
# 目录

- 流程IT去O与上云
- 用好PostgreSQL
- 常见问题与对策
- **未来展望**

# 未来展望

虽然PostgreSQL还不完美，但只要正确使用，完全可以应用到企业的关键系统。
随着PostgreSQL用户的不断增加，功能和生态也将越来越完善。

参考：
https://zhuanlan.zhihu.com/p/126601825



# THANKS

欢迎关注GaussDB数据库公众号