Oracle 到 LightDB 开发指南

目录

简介

如果用户希望编写的 SQL 非常通用，不使用任何具体数据库特有的特性如并行执行、压缩、优化器提示、重做日志优化、数据异常恢复能力、可管理性，那么使用 PostgreSQL 社区版、**Percona** Distribution for **PostgreSQL**、**Postgres Pro Standard** 或 LightDB 并无本质性差别。所以，通常某个分支是有着特定的考虑，它们可能和高可用、SQL 语法兼容性、亦或性能相关。本文讨论的就是这些差别。

因为 LightDB 是直接基于 PostgreSQL 社区版，所以绝大多数服务器本身的特性是一样的（除了 Oracle 常用语法兼容性、企业级监控平台、PWR（对应于 Oracle AWR）、审计、优化器提示、top 命令行工具、物理备份工具、只读表优化等），所以除非特别说明，在本文中针对开源 PostgreSQL 说明的特性可以无缝适用于 LightDB，并不专门分开讨论它们。大部分测试以 LightDB 21.1-13.3 为例，但是为了沟通更方便，也可能直接表述为 PostgreSQL，Oracle 则使用 19c。

在此笔者还想补充一句，多年前，对于 B 端系统，阿里云就开始推荐 PostgreSQL 了，这确实是更适合的方案，但是这个推荐是在阿里云上线 postgresql 之后（2015 年,阿里云宣布正式推出 RDS for PostgreSQL 服务），目前主推 PolarDB-X，14 年笔者在公司论坛的帖子也是这么说的。存在即合理，应该根据实际情况选择尽可能最满足要求的那个。

限于篇幅，本文尽可能站在开发人员和 DBA 角度介绍笔者认为比较重要的那些特性，对于可能有差别但是个人认为并不会有很多实现成本差异的特性或并不太合适在数据库中实现的特性，文本并不做讨论；对于一些对开发或性能并没有那么重要的特性例如 PostgreSQL 嵌入式编程（类 Oracle ProC）、密钥管理、默认认证模式等亦如是；值得一提的是，对于 Oracle 和 LightDB 兼容特性在底层实现细节上的差异，例如 to_char 支持的格式、date 是否包含时间部分，默认 timestamp 精确到微秒还是毫秒，本文会尽可能描述清楚，LightDB 在开源版本和组件的基础上实现了更强的语义一致性，因此更符合用户预期行为。

为了最大程度的节省篇幅，很多内容维护在笔者的技术网站上，将以链接的方式指向。

PostgreSQL 发展及其分支简介

分支发展史

先来看一下 PostgreSQL 的发展史，如下图所示：

The History of PostgreSQL

The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

THE DESIGN OF POSTGRES

The implementation of POSTGRES began in 1986.

1986

1987

DEMOWARE

The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference.

The implementation of POSTGRES

Version 1, was released to a few external users in June 1989

1989

1990

A commentary on the POSTGRES rules system

Version 2 was released in June 1990 with the new rule system

rewritten rule system

Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rule system.

1991

1993

End of postgres

In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

Postgres95

In 1994, Andrew Yu and Jolly Chen added an SQL language interpreter to POSTGRES. Under a new name, Postgres95.

1994

1996

PostgreSQL

In 1996, Postgres95 has been renamed to PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, version numbering starting with 6.0.

Release 7.0

There are more improvements and fixes in 7.0 than in any previous release which includes Foreign Keys,updated psql, optimizer overhaul, join syntax

2000

2005

Windows service

Version 8.0: This is the first PostgreSQL release to run natively on Microsoft Windows® as a server. Point-in-time recovery is introdeced along with few more improvements

Introduced Replication

Version 9.0: his release of PostgreSQL adds features that have been requested for years, such as easy-to-use replication, a mass permission-changing facility, and anonymous code blocks.

2010

2017

Logical replication

Version 10.0: Major enhancements in PostgreSQL 10 include:

- Logical replication using publish/subscribe
- Declarative table partitioning
- Improved query parallelism.

Partitioning

Version 11.0: Major enhancements in PostgreSQL 11 include:

- Improvements to partitioning functionality
- Improvements on parallelism
- SQL stored procedures that support embedded transactions

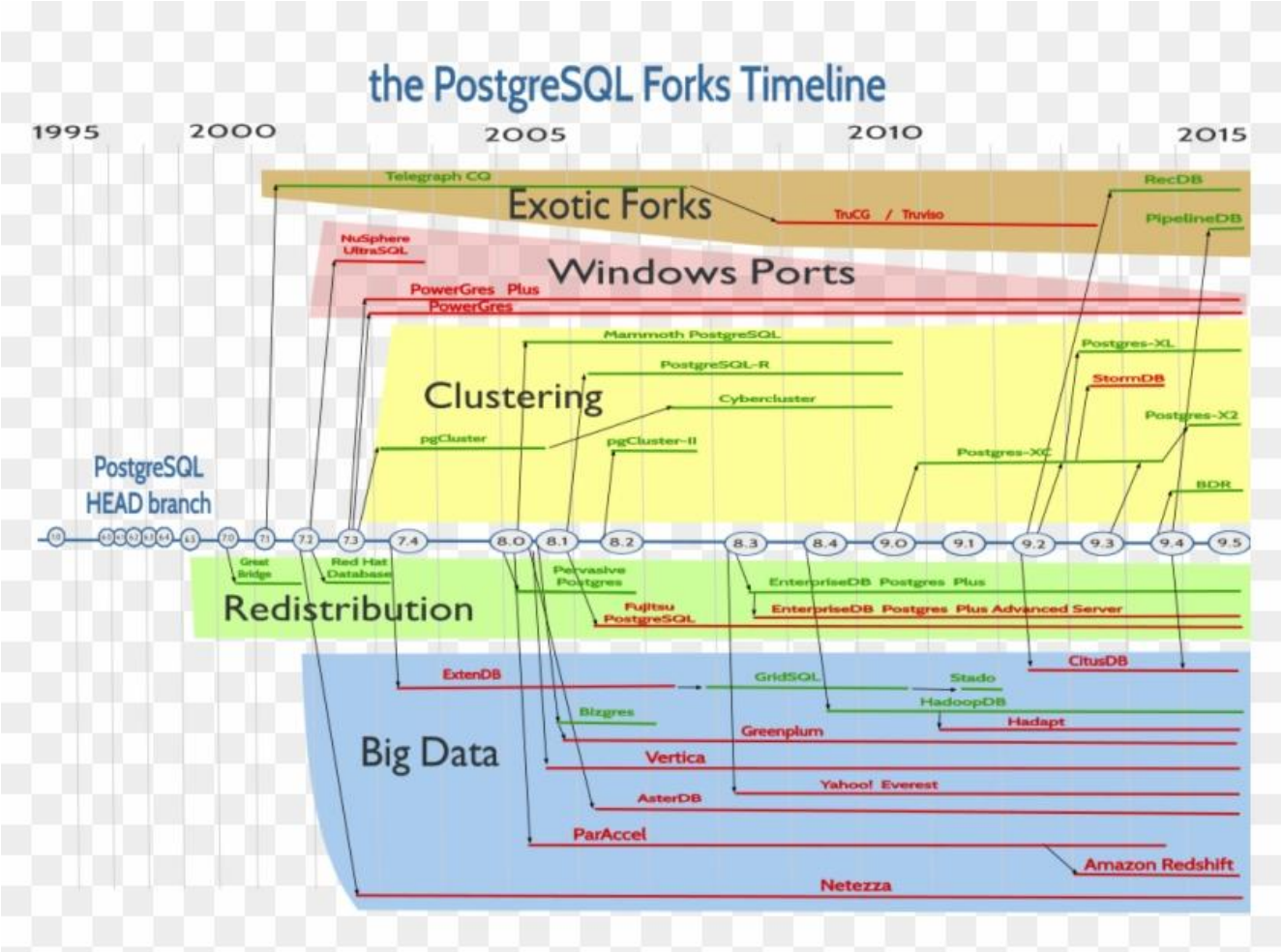
2018

postgreshelp.com

Follow us on
Facebook.com/postgreshelp
Twitter.com/postgreshelp

PostgreSQL 的前身 Postgres 起始于 1986 年（Oracle 差不多也在这一时代单身，而 MySQL AB 公司则成立于 1997 年），它可以认为是 PostgreSQL 内核原型，并不支持 SQL 接口，到 1994 年，伯克利由于支持压力问题，在 V4.2 版本时官宣终止。1995 年增加了 SQL 语言解析器，1996 年命名为 PostgreSQL，同时版本命名为 6.0，沿用至今，同年 PostgreSQL 全球开发组成立。2000 年 V8.0 版本第一次官方正式支持 Windows。

从 PostgreSQL 7.0 开始，延伸出了大量的 PostgreSQL 分支，这些分支通常都为三方商业公司所维护和支持，并且绝大部分不开源，包括著名的 EnterpriseDB、CYBERTEC PostgreSQL、Greenplum、PostgreSQL-XC、PostgreSQL-XL、Postgres Pro、FUJITSU Enterprise Postgres、CitusDB 及 Netezza，以及 PolarDB、openGauss、Tbase、人大金仓、达梦早期的版本。



由此可见，PostgreSQL 并不是新的数据库，并且全球内使用广泛，并且因为采用 BSD 协议，所以不存在版权风险问题。

主流维护版本

首先来看 PostgreSQL 当前的各自活跃版本。mariadb 目前主流的版本为 9.6 到 14，其中 14 当前还在 BETA 阶段，9.6 将于 2021 年 11 月 11 日终止维护。每个版本都包含了较多的新特性，例如 PostgreSQL 10 极大的增强了并行执行，PostgreSQL 12 支持多存储引擎架构。正常大约三个月发布一次。

LightDB 数据库本身基本上沿用大约 3 个月左右发布一次的策略，在合并社区新版本上，会两个版本合并一次，例如当前 LightDB 是基于 13.3，那么在 PostgreSQL 社区发布到 13.5

时，1 个月内将会进行底层的合并；在并行维护的版本上，会支持三个并行版本，例如 13-15，到 16.2 发布时，13 将不再发布新版本；周边配套如 LightDB-EM 则会更加频繁。

一般来说，太高的发布频率既有利也有弊。从好的方面来说，用户可以更及时地收到功能和错误修复。从不好的方面来说，为了让 LightDB 保持最新的状态，很可能引入更多的 bug。

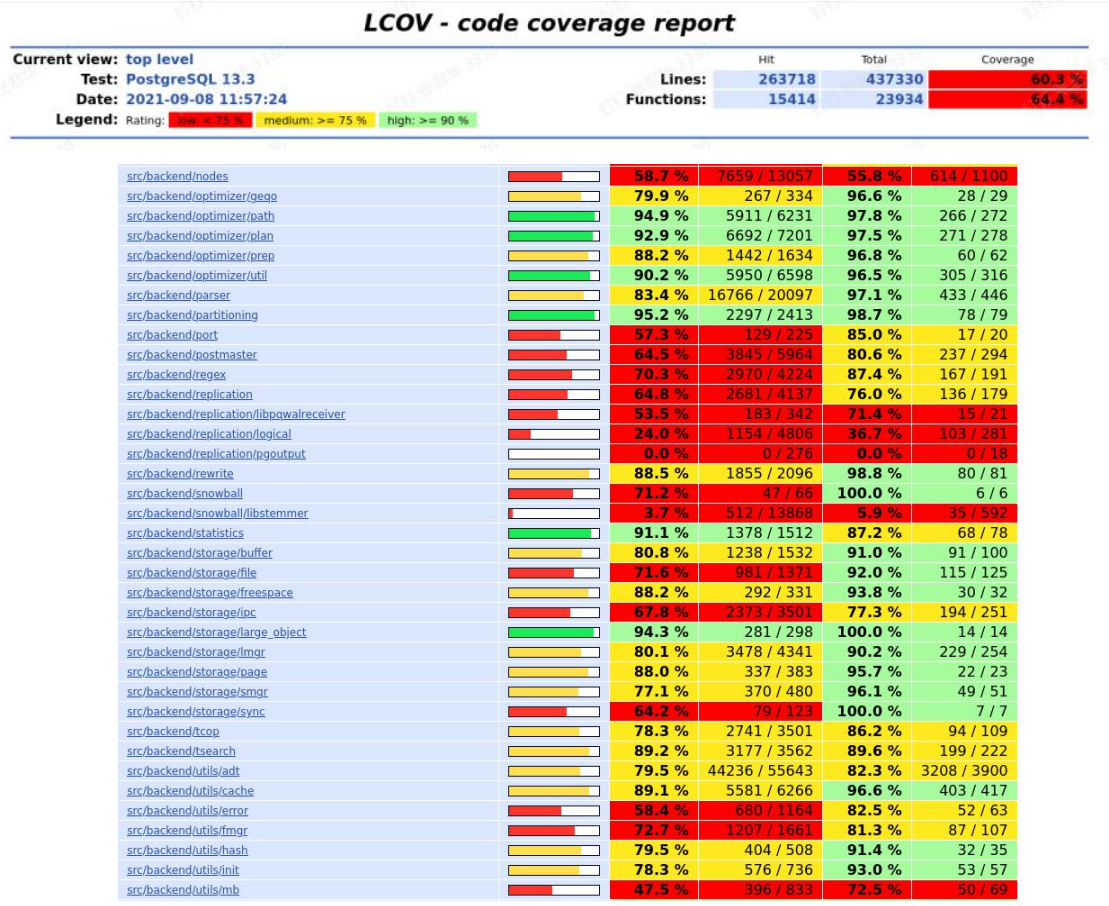
注 1：openGauss 和 PolarDB 都是绑定在具体版本上，openGauss 为 9.2，PolarDB 基于 11.9。

注 2：PolarDB-X 并不是基于 PostgreSQL，计算层使用 Java 编写，存储层使用 MySQL。

质量保证体系

PostgreSQL 包含了完整的自测体系，俗称回归测试（https://www.postgresql.org/docs/current/regress.html）。包括核心测试、插件测试、ECPG 测试、认证测试、并发压力测试（用于测试 ACID 隔离性）、存储过程测试、崩溃恢复和流复制测试、逻辑复制测试，以及针对客户端的测试。以及测试覆盖率检查。

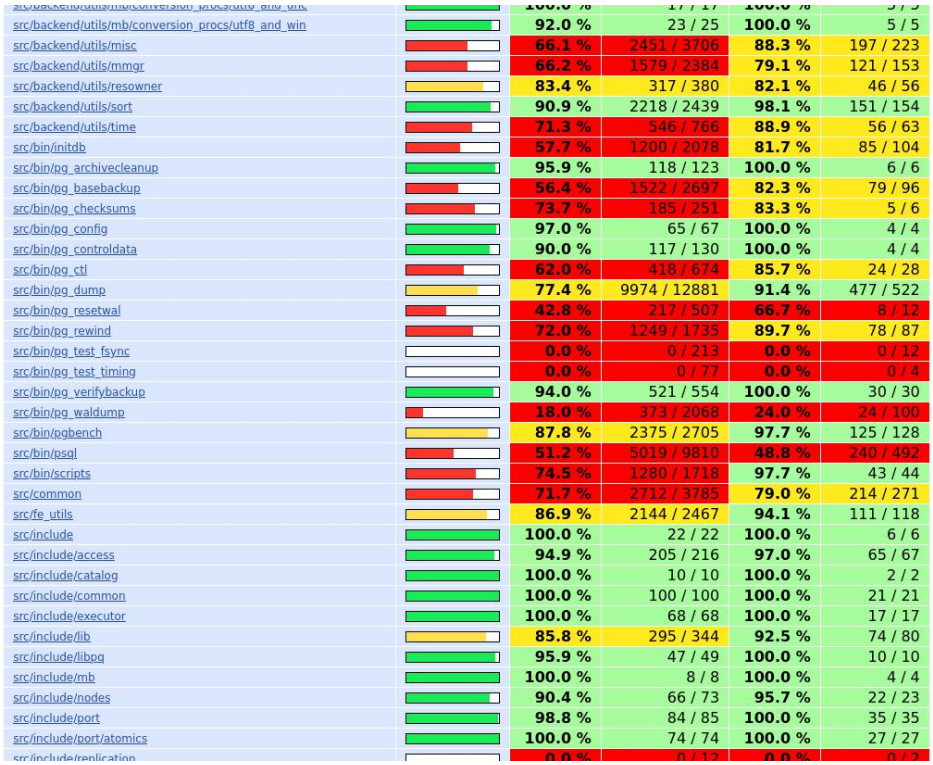
在 LightDB 分支中，日常已经运行的测试用例有 9000 多个，核心测试覆盖率 90% 以上，整体在 60%以上。如下：



LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level			
Test:	PostgreSQL 13.3	Lines:	263718	437330	60.3 %
Date:	2021-09-08 11:57:24	Functions:	15414	23934	64.4 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

src/backend/nodes	58.7 %	7659 / 13057	55.8 %	614 / 1100
src/backend/optimizer/geqo	79.9 %	267 / 334	96.6 %	28 / 29
src/backend/optimizer/path	94.9 %	5911 / 6231	97.8 %	266 / 272
src/backend/optimizer/plan	92.9 %	6692 / 7201	97.5 %	271 / 278
src/backend/optimizer/prep	88.2 %	1442 / 1634	96.8 %	60 / 62
src/backend/optimizer/util	90.2 %	5950 / 6598	96.5 %	305 / 316
src/backend/parser	83.4 %	16766 / 20097	97.1 %	433 / 446
src/backend/partitioning	95.2 %	2297 / 2413	98.7 %	78 / 79
src/backend/port	57.3 %	129 / 225	85.0 %	17 / 20
src/backend/postmaster	64.5 %	3845 / 5964	80.6 %	237 / 294
src/backend/regex	70.3 %	2970 / 4224	87.4 %	167 / 191
src/backend/replication	64.8 %	2681 / 4137	76.0 %	136 / 179
src/backend/replication/libpqwalreceiver	53.5 %	183 / 342	71.4 %	15 / 21
src/backend/replication/logical	24.0 %	1154 / 4806	36.7 %	103 / 281
src/backend/replication/pgoutput	0.0 %	0 / 276	0.0 %	0 / 18
src/backend/rewrite	88.5 %	1855 / 2096	98.8 %	80 / 81
src/backend/snowball	71.2 %	47 / 66	100.0 %	6 / 6
src/backend/snowball/libstemmer	3.7 %	512 / 13868	5.9 %	35 / 592
src/backend/statistics	91.1 %	1378 / 1512	87.2 %	68 / 78
src/backend/storage/buffer	80.8 %	1238 / 1532	91.0 %	91 / 100
src/backend/storage/file	71.6 %	981 / 1371	92.0 %	115 / 125
src/backend/storage/freespace	88.2 %	292 / 331	93.8 %	30 / 32
src/backend/storage/ipc	67.8 %	2373 / 3501	77.3 %	194 / 251
src/backend/storage/large_object	94.3 %	281 / 298	100.0 %	14 / 14
src/backend/storage/lmgr	80.1 %	3478 / 4341	90.2 %	229 / 254
src/backend/storage/page	88.0 %	337 / 383	95.7 %	22 / 23
src/backend/storage/smgr	77.1 %	370 / 480	96.1 %	49 / 51
src/backend/storage/sync	64.2 %	79 / 123	100.0 %	7 / 7
src/backend/tcop	78.3 %	2741 / 3501	86.2 %	94 / 109
src/backend/tsearch	89.2 %	3177 / 3562	89.6 %	199 / 222
src/backend/utils/adt	79.5 %	44236 / 55643	82.3 %	3208 / 3900
src/backend/utils/cache	89.1 %	5581 / 6266	96.6 %	403 / 417
src/backend/utils/error	58.4 %	680 / 1164	82.5 %	52 / 63
src/backend/utils/fmgr	72.7 %	1207 / 1661	81.3 %	87 / 107
src/backend/utils/hash	79.5 %	404 / 508	91.4 %	32 / 35
src/backend/utils/init	78.3 %	576 / 736	93.0 %	53 / 57
src/backend/utils/mb	47.5 %	396 / 833	72.5 %	50 / 69



src/backend/utils/mb/conversion_procs/utf8_and_win	92.0 %	23 / 25	100.0 %	5 / 5
src/backend/utils/misc	66.1 %	2451 / 3706	88.3 %	197 / 223
src/backend/utils/mmgr	66.2 %	1579 / 2384	79.1 %	121 / 153
src/backend/utils/resowner	83.4 %	317 / 380	82.1 %	46 / 56
src/backend/utils/sort	90.9 %	2218 / 2439	98.1 %	151 / 154
src/backend/utils/time	71.3 %	546 / 766	88.9 %	56 / 63
src/bin/initdb	57.7 %	1200 / 2078	81.7 %	85 / 104
src/bin/pg_archivecleanup	95.9 %	118 / 123	100.0 %	6 / 6
src/bin/pg_basebackup	56.4 %	1522 / 2697	82.3 %	79 / 96
src/bin/pg_checksums	73.7 %	185 / 251	83.3 %	5 / 6
src/bin/pg_config	97.0 %	65 / 67	100.0 %	4 / 4
src/bin/pg_controldata	90.0 %	117 / 130	100.0 %	4 / 4
src/bin/pg_ctl	62.0 %	418 / 674	85.7 %	24 / 28
src/bin/pg_dump	77.4 %	9974 / 12881	91.4 %	477 / 522
src/bin/pg_resetwal	42.8 %	217 / 507	66.7 %	8 / 12
src/bin/pg_rewind	72.0 %	1249 / 1735	89.7 %	78 / 87
src/bin/pg_test_fsync	0.0 %	0 / 213	0.0 %	0 / 12
src/bin/pg_test_timing	0.0 %	0 / 77	0.0 %	0 / 4
src/bin/pg_verifybackup	94.0 %	521 / 554	100.0 %	30 / 30
src/bin/pg_waldump	18.0 %	373 / 2068	24.0 %	24 / 100
src/bin/pgbench	87.8 %	2375 / 2705	97.7 %	125 / 128
src/bin/psql	51.2 %	5019 / 9810	48.8 %	240 / 492
src/bin/scripts	74.5 %	1280 / 1718	97.7 %	43 / 44
src/common	71.7 %	2712 / 3785	79.0 %	214 / 271
src/fe_utils	86.9 %	2144 / 2467	94.1 %	111 / 118
src/include	100.0 %	22 / 22	100.0 %	6 / 6
src/include/access	94.9 %	205 / 216	97.0 %	65 / 67
src/include/catalog	100.0 %	10 / 10	100.0 %	2 / 2
src/include/common	100.0 %	100 / 100	100.0 %	21 / 21
src/include/executor	100.0 %	68 / 68	100.0 %	17 / 17
src/include/lib	85.8 %	295 / 344	92.5 %	74 / 80
src/include/libpq	95.9 %	47 / 49	100.0 %	10 / 10
src/include/mb	100.0 %	8 / 8	100.0 %	4 / 4
src/include/nodes	90.4 %	66 / 73	95.7 %	22 / 23
src/include/port	98.8 %	84 / 85	100.0 %	35 / 35
src/include/port/atomics	100.0 %	74 / 74	100.0 %	27 / 27

测试代码主要包含在 `src/test`、`src/bin`、`src/pl`、`src/interfaces/ecpg/test` 等各个模块中，比较符合通用单元测试的惯例。

LightDB 与 Oracle 关键特性兼容性与替代实现

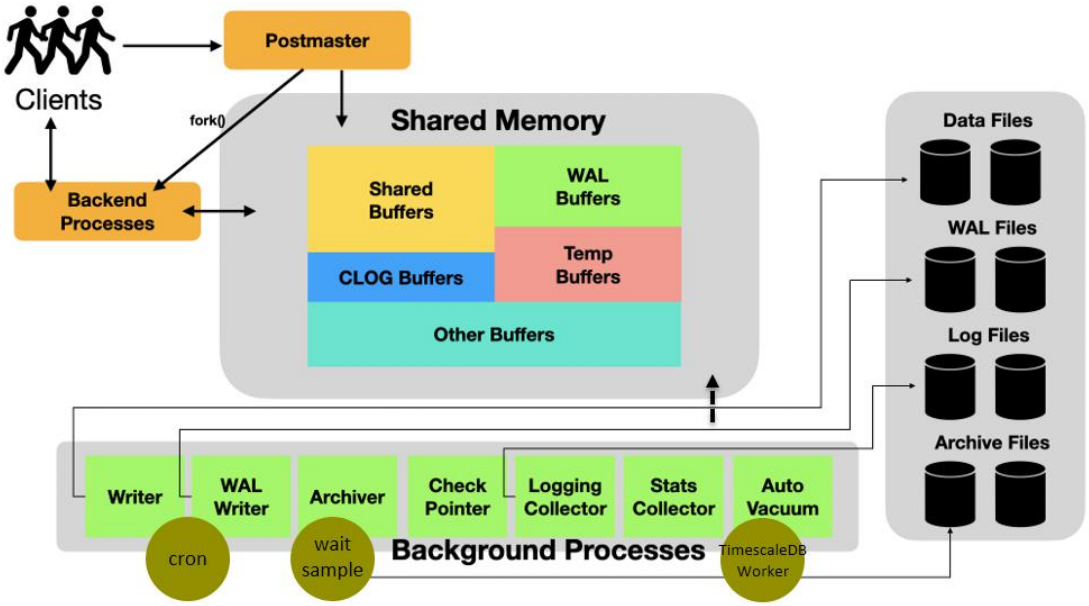
术语差异

因为 PostgreSQL 更偏学术化，Oracle 更偏向工业化，所以 PostgreSQL 一直沿用了历史上的术语，体现在英文文档中和数据字典表中尤为明显。

PostgreSQL	Oracle
集群 （Cluster）	实例 （Instance）
WAL	redo
元组 （tuple）	行
关系 （relation）	表
属性 （attribute）	列
页 （page）	块

架构差异

LightDB 和 Oracle 架构比较类似，都是采用多进程架构，使用共享内存进和信号量行进程间通信，具有一些公用的后台进程。一个客户端连接默认对应一个服务端进程（并行执行除外）。如下所示：

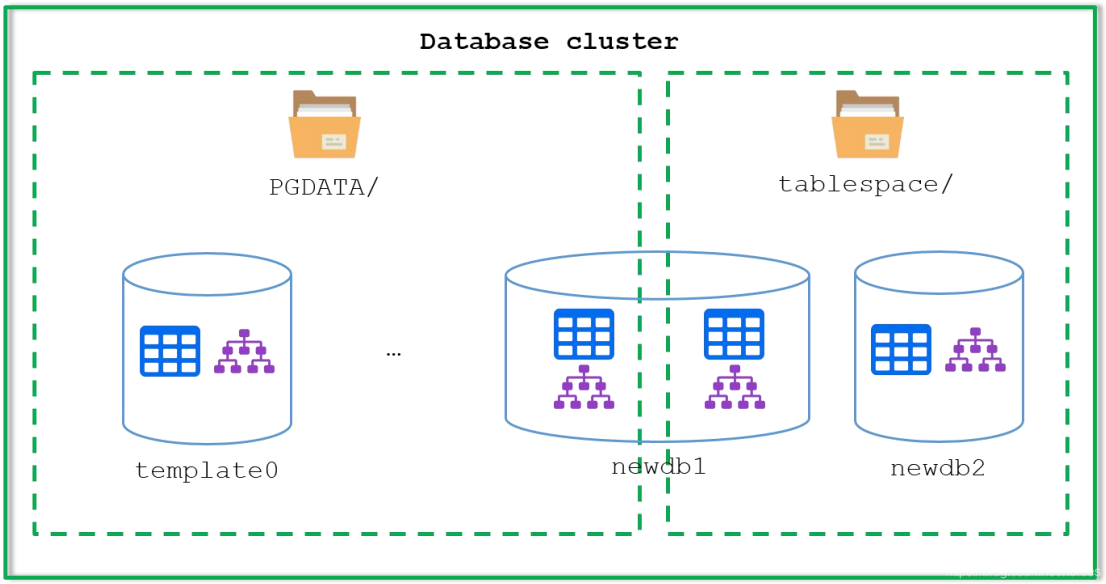


唯一不同的是，Oracle 由监听器负责为客户端建立连接，PostgreSQL 则由 Postmaster 进程负责为客户端建立连接。

存储架构

在一个 LightDB 实例（术语称为集群，英文单词为 Cluster）中，可以包含多个数据库，每个数据库之间不能直接交互，不同数据库的表不能直接关联，任何时候，一个客户端连接只能访问一个数据库，本质上比较像 Oracle 12c 引入的多租户。

在 LightDB 实例中；一个表空间可以让多个数据库使用；而一个数据库可以使用多个表空间。属于"多对多"的关系。



可以在创建数据库时指定默认表空间，如下：

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

注：数据库中如果包含对象了，则不能修改默认表空间。

也可以创建表时指定表空间，如下：

```
CREATE TABLE cinemas (
```

```
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

在 Oracle 数据库中；一个表空间只属于一个数据库使用；而一个数据库可以拥有多个表空间。属于"一对多"的关系。

系统自带表空间

- 表空间 pg_default 是用来存储系统目录对象、用户表、用户表 index、和临时表、临时表 index、内部临时表的默认空间。对应存储目录$PADATA/base/
- 表空间 pg_global 用来存放系统字典表；对应存储目录$PADATA/global/

在 PostgreSQL 中，临时表空间由参数 temp_tablespaces 指定，默认为当前表空间的 pgsql_tmp 目录。在性能优化中，通常需要优化排序和子查询使用的临时文件，而跟踪需要多少临时文件是一个很麻烦的事，LightDB 包含了一个 GUC 选项 xxx，用于控制事务完成后是否保留临时文件，其位于临时文件目录的 keep 子目录。

表存储

在 posgresql 中，表和文件一一对应，索引也和文件一一对应。文件大小默认 1GB。lightdb 中，单个文件大小最大为 30GB。

通过函数 `pg_relation_filepath('object_name')` 可以查询表和数据库对应的文件名。

平台和操作系统支持

PostgreSQL 默认支持 x86 和 ARM 架构，Oracle 当前不支持 ARM 架构（计划在 2021 年底支持，主要针对 Ampere 的 ARM 处理器 Altra，不支持鲲鹏处理器）。

PostgreSQL 和 Oracle 默认支持 Windows、Redhat/CentOS，对于麒麟操作系统，两者默认均未官方认证。LightDB 默认在麒麟操作系统 V10 下编译和测试，同时支持 Redhat/CentOS，不支持 Windows。

管理差异

在 oracle 中，查询各种内置信息通常是 dba/all/user_xxxs 相关表。比如查询某个表的大小，通过 dba_segments。运行时信息则在各种 v$视图中。

在 pg 中，一般通过表函数进行查询更为便利，详细可以参见 https://www.postgresql.org/docs/current/functions-admin.html#FUNCTIONS-ADMIN-DBOBJECT。

系统参数查看与更改

postgresql 参数相对来说比较复杂一些，除了熟知的参数可修改范围外， 参数本身也

有一个可修改范围的概念。

```
pgbench=# select distinct context from pg_settings order by 1;
context
------------------
backend
internal
postmaster
sighup
superuser
superuser-backend
user
```

其中 internal 是代码中写死的，对外不可修改。例如 block_size 块大小。

pg_file_settings 中查询的是 postgresql.conf 明确设置的参数。pg_show_all_file_settings()是对应的函数封装。

pg_settings 中查询的是所有的参数。pg_show_all_settings()是对应的函数封装。

用户可以通过 show xxx 查看当前会话的参数值，遗憾的是，PG 默认不支持通配，LightDB 21.3 将支持通配查询。

用户可以通过 set 修改当前会话的参数，也可以通过优化器提示 Set(name val)修改语句级别的参数，典型的是 work_mem、enable_xxx 等。

和 oracle 一样，LightDB 支持通过 `ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' | DEFAULT }`永久修改服务器参数值，相当于 scope=spfile。它会将新的值写入 postgresql.conf.auto 文件，重启时，其中的参数设置具有更高的优先级。

如果希望参数立刻生效，可以执行 SELECT pg_reload_conf();或 pg_ctl reload 或 kill -HUP PID。

此时，lightdb.log 中会包含一条日志”LOG: received SIGHUP, reloading configuration files”。

事务

在 postgresql 中，事务的默认隔离级别为读提交，这一点和 oracle 一样。在自动提交方面，postgresql 和 oracle 的行为有点不同，DML 默认也是自动提交，这一点更像 mysql。由参数 default_transaction_isolation 控制默认事务隔离级别。要关闭自动提交，需要先执行 BEGIN 或 START TRANSACTION。

LightDB 支持串行化、可重复度和读提交三种隔离级别，和 Oracle 的语义相同。

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

https://www.postgresql.org/docs/13/transaction-iso.html

客户端及驱动

linux、windows 和 macOS 下优先推荐 Dbeaver，支持语法提示，无服务端和客户端版本限制。SQL Developer、navicat（推荐 15+）、pgadmin 等均存在或多或少兼容性、易用性和锁问题。

C 客户端库

lightdb 在协议上完全兼容 postgresql，因此可以使用 libpq，具体使用可见 https://www.postgresql.org/docs/current/libpq-example.html。

除了 libpq C SDK 外，lightdb 还支持类似于 oracle pro*c 一样的预编译式 ecpg，具体使用可见 https://www.postgresql.org/docs/current/ecpg.html。

JDBC 客户端驱动

```
<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.24</version>
</dependency>
```

postgresql JDBC 官方地址为 https://jdbc.postgresql.org/

注：jre7 和 jre6 结尾分别代表兼容 JDK 7 和 JDK 6。

JDBC 数据类型对应关系

PostgreSQL 与 JDBC 数据类型的对应关系可参见 https://www.cnblogs.com/zhjh256/p/15233946.html

LightDB 限制

特性	PostgreSQL
表中索引的数量	无限制
单个库表数量	1,431,650,303
单索引可以包含的字段数	32
同一字段数多个索引	支持

列数	1600，实际上块 8K 时 450 比较具有可行性
行数	无限制
VARCHAR 总长度	无限制
单行最大长度	8K，超出后，对于宽字段采用 TOAST 机制解决，默认当一行插入的记录超过 2KB 之后，TOAST 就会尝试压缩宽列，如果压缩之后仍然超过 2KB，就会将宽字段分块，源表通过指针指向分块后的 TOAST，如果低于 2KB，则直接存储。所以有些宽列压缩后存储在行内，有些在 TOAST 是可能的。
单实例最大并发	2000 以下（TPC-C 测试结论）
每个连接占用内存	1.7MB（TPC-C 测试结论）
单个表大小	32TB
单个字段大小	1GB
标识符默认大小写 creatTime	小写 creattime
标识符长度	63 字节
大小写敏感的保留	SELECT "from" ,"to" FROM sc;
分区键字段数	32

date、number、varchar2 等数据类型

对于 oracle 常用数据类型 char、varchar2、integer、number、date、timestamp [WITH TIME ZONE]、clob、blob、JSON（oracle 20c 新特性，预览版，未公开发布），PostgreSQL 均提供了对应的类型以及某些情况下更细粒度的分类。

Oracle	PostgreSQL	备注
varchar（本质上是 oracle 的保留关键字，用于以后满足 ANSI SQL 的定义）、varchar2 、 nvarchar 、nvarchar2	varchar, text varchar2（LightDB 扩展）	Oracle 默认最大长度是 2000，PostgreSQL 无限制 Oracle 默认字节位单位（由参数 NLS_LENGTH_SEMANTICS 控制），PostgreSQL 默认字符为单位，所以不需要 nvarchar 类型。
char	char	同上
number	numeric、decimal number（LightDB 21.2 扩展）	不带(precision, scale)时，默认是都不限制，不同于 MySQL，其实现是 scale 为 0，遵从了 SQL 标准。
integer	smallint integer bigint	

date	date（不带时间部分） timestamp（带时间部分） datetime（带时间部分，LightDB 扩展）	
TIMESTAMP	timestamp	精度最高到微秒
TIMESTAMP WITH TIME ZONE	timestamptz 、 timestamp with time zone	
BLOB	bytea LightDB 21.2 支持 BLOB 关键字	
CLOB	text LightDB 21.2 支持 CLOB 关键字	
JSON oracle 20c 新特性，之前版本使用 VARCHAR 或 CLOB	JSONB	

rowid、rownum、sysdate、dual 等伪表与列

oracle	LightDB
rowid	ctid，LightDB 支持 rowid（作为 ctid 的同义词实现）

rownum	不支持。LightDB 21.3 支持 rownum，执行顺序为先 order by，后 rownum，更加符合我们的预期行为；但是 rownum>、>=的行为等同于 oracle，返回 0 行。
	
	如果暂时无法使用 rownum，可以用临时序列方式实现：
	`create temp sequence temp_seq;`
	`select nextval('temp_seq') as ROWNUM, c1`
	`from sometable`
	`ORDER BY c1 desc;  -- 性能比基准慢 1 倍，是指不带行号`
	或者
	`select row_number() over() as id, t.*`
	`from information_schema.tables t;  - 性能比基准慢 4 倍，是指不带行号`
	注： 该特性不支持并行执行。

sysdate	不支持。LightDB 支持
dual	不支持。LightDB 支持

nvl、to_date、to_char、listagg、decode 等函数

oracle	LightDB
nvl 支持任意同类（或可以转到	支持精确相同数据类型。例如 select nvl(1.2,1)报 ERROR: function nvl(numeric, integer) does not exist

同类，也就是 parstXXX 不报错）数据类型	LightDB 21.2 支持任意同类(或可以转到同类, 也就是 parstXXX 不报错）数据类型
COALESCE 支持任意同类（parstXXX 兼容也会报错，和 NVL 不同）数据类型	支持任意同类数据类型，和 NVL 不同 例如 select coalesce(null,23.1,1,2)
listagg	string_agg 。 Syntax: STRING_AGG (expression, separator [order_by_clause]) LightDB 21.3 支持 listagg、wm_concat 同义词
||	支持字符串、整型之间的拼接。 Lightdb 支持任意数据类型之间的拼接

null、空格、空字符串及尾部空格

oracle	LightDB
x = null 永远不成立	开源 postgresql x = null 不成立 LightDB x = null 成立 (transform_null_equals=true)
x is null	x is null
'' = null 永远不成立	开源 postgresql '' = null 不成立 LightDB '' = null 成立 (transform_null_equals=true)
' ' = ''不成立	' ' = ''不成立
' ' = ' '成立	' ' = ' '成立
'a ' = 'a'成立（剔除后缀空格）	'a ' = 'a'不成立（不剔除后缀空格）

字符集与排序规则

Oracle	LightDB
AL32UTF8	UTF-8
ZHS16GBK	EUC_CN

推荐使用 UTF-8 编码。

LightDB 不支持表级别指定字符集，库级别可以指定字符集。

LightDB 创建数据库时指定字符集： createdb -E EUC_CN -T template0 --lc-collate=zh_CN --lc-ctype=zh_CN dbname

https://www.cnblogs.com/zhjh256/p/15518146.html

create table as/create table like

在批处理、同步场景和日常工作过程中，我们经常会因为各种原因需要基于源表创建目

标表。此时 CTAS 的语义一致性就很重要了。

在 oracle 中，只有 create table as，不支持 create table like。

create table as select 会创建表，并将 as select 子查询的结果带到目标表。对于源表明确定义了 NOT NULL 的列，新表也会包含 NOT NULL 约束。除此之外，Oracle 的 ctas 不会包含其他约束如主键、唯一键、外键、检查约束以及索引和默认值。如下：

```
scott@TICKET> create table p
2 ( id number primary key ,
3 username varchar(25) ,
4 passwd varchar(24),
5 email varchar(30),
6 birth date,
7 tel number ,
8 sex char(1));
表已创建。
scott@TICKET> alter table p add age number ;
表已更改。
```

创建一个唯一约束

```
scott@TICKET> alter table p modify tel unique;
表已更改。
```

创建一个检查约束

```
scott@TICKET> alter table p add constraint p_ck_age check(age>0 and age<150);
表已更改。
```

创建一个默认约束

```
scott@TICKET> alter table p modify sex default '0';
表已更改。
scott@TICKET> select constraint_name,constraint_type,status from user_constraints where table_name='P';
CONSTRAINT_NAME C STATUS
------------------------------ - --------
SYS_C0015996 P ENABLED
SYS_C0015997 U ENABLED
P_CK_AGE C ENABLED
```

通过查询创建相关的表信息.

```
scott@TICKET> create table persion as select * from p;
表已创建。
scott@TICKET> select * from persion;
未选定行
scott@TICKET> select constraint_name,constraint_type,status from user_constraints where table_name='PERSION';
未选定行
```

说明，ctas 确实没有包含相关定义。

在 postgresql 中，包含 CREATE TABLE AS 和 CREATE TABLE LIKE 两种定义。CTAS 和 oracle 几乎一致，不仅如此，postgresql 没有将 NOT NULL 约束拷贝过来。

因为本质上 CTAS 是基于 select 的结果自动创建表，语义上并没有限制不能增加表达

式、多表关联的定义，所以没有拷贝语义也是情理中。

LIKE 不同于 CREATE TABLE AS 语句, 它是标准 CREATE TABLE 语句的一个参数项。其定义如下：

```
and like_option is:

{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL }
```

因此可以指定包含或排除某些选项，默认不包含子句的情况下，其和 oracle 的 ctas 一样，会拷贝 NOT NULL 约束。如果我们的需求是：

1. 所有约束、索引和注释在复制时都应被拷贝。
2. 序列不应拷贝，应当为每一张复制的表单独创建一个新序列。

所以可以按照如下编写：

```
create table t_key_event_file_student_102 (like t_key_event_file_student INCLUDING INDEXES INCLUDING COMMENTS INCLUDING CONSTRAINTS INCLUDING DEFAULTS);
```

如果表的主键是基于序列自增的，那就不能包含 INCLUDING DEFAULTS，但是这又会导致默认值拷贝的缺失。在 LightDB 22c 中，CREATE LIKE 的 INCLUDING DEFAULTS 子句将实现对于自增列，自动创建新的序列。

存储过程与函数

postgresql 从 11 开始支持存储过程，在此之前，只支持函数。因为函数支持 OUT 出参，因此，存储过程可以实现的功能，通过函数都可以实现。

postgresql 的数据类型更接近 mysql，但是其存储过程更接近 oracle。

这部分列出各个特性的支持情况，完整实例可参考：https://www.cnblogs.com/zhjh256/p/15399319.html。

特性	postgresql
匿名块/语句块结构	do $$ declare … begin … end; $$
语义声明	LANGUAGE lang_name PARALLEL { UNSAFE | RESTRICTED | SAFE } { IMMUTABLE | STABLE | VOLATILE }
存储过程	IN, INOUT。不支持 OUT 参数。

函数	CREATE FUNCTION somefunc(integer, text) RETURNS integer
	$$
	AS
	DECLARE
	…
	BEGIN
	'function body text'
	END;

	$$ LANGUAGE plpgsql; IN, OUT, INOUT。 参数名可选，没有提供时，可以使用$1,$2 引用。和 shell、c 函数的用法一样。
定义	x int;
记录类型	支持，语法和 oracle 一样。

事务	在函数中，pg 支持一个函数作为一个事务，不支持内嵌事务。存储过程支持事务控制。
	CREATE PROCEDURE transaction_test1()
	LANGUAGE plpgsql
	AS $$
	BEGIN
	FOR i IN 0..9 LOOP
	INSERT INTO test1 (a) VALUES (i);
	IF i % 2 = 0 THEN
	COMMIT;
	ELSE
	ROLLBACK;
	END IF;
	END LOOP;
	END;
	$$;
	
	CALL transaction_test1();

赋值	x:=b;

游标定义及循环	CREATE FUNCTION reffunc2() RETURNS refcursor AS '
	DECLARE
	ref refcursor;
	BEGIN
	OPEN ref FOR SELECT col FROM test;
	RETURN ref;
	END;
	' LANGUAGE plpgsql;
	CREATE OR REPLACE FUNCTION testing_trigger()
	RETURNS SETOF varchar AS
	$cursor_test$
	DECLARE
	cur CURSOR FOR select * from employee emp;
	test_cur RECORD;
	BEGIN
	open cur;
	LOOP
	fetch cur into test_cur;
	exit when test_cur = null;

	if test_cur.customer like '%AB%' then
	return next test_cur.customer;
	end if;
	END LOOP;
	close cur;
	END;
	$cursor_test$
	LANGUAGE plpgsql VOLATILE;

if…else	if … then … else if … then … else … end if;
异常	raise notice 'a is greater then b';
调用无返回值函数	perform
调用有返回值函数	execute select func into …
调用存储过程	call
执行 DDL	直接写 SQL 即可
执行动态 SQL	execute dymsql into … using …

Oracle 到 LightDB 存储过程的迁移也可参见 https://www.postgresql.org/docs/14/plpgsql-porting.html, https://docs.microsoft.com/en-us/azure/postgresql/howto-migrate-from-oracle, https://wiki.postgresql.org/wiki/Oracle_to_Postgres_Conversion

注：在分布式数据库中，是否支持存储过程和函数调用依赖于具体的实现，不能一并而论。如果包含游标出参，则肯定不支持。如果是表函数，也视具体实现。

表函数/游标

在 postgresql 中，将函数返回值定义为 setof 或 table 会使得函数变为表函数，可以在 select 子句或 from 子句中作为表使用。如下：

```
CREATE FUNCTION events_by_type_1(text) RETURNS TABLE(id bigint, name text) AS $$
    SELECT id, name FROM events WHERE type = $1;
$$ LANGUAGE SQL STABLE;
SELECT * FROM events_by_type_1;
```

表函数在 Pg 中使用非常简单和直接。基于这种特性，在 pg 中，如果存储过程或函数要返回游标出参，几乎是多余的。

注：表函数在 pg 内置管理功能中被广泛的使用。

虽然如此，在 oracle 中，返回结果集仍然广泛的使用着游标，即使从 oracle 9i 开始就已经支持表函数。

```
CREATE OR REPLACE PROCEDURE getDBUSERCursor(
        p_username IN DBUSER.USERNAME%TYPE,
        c_dbuser OUT SYS_REFCURSOR)
IS
BEGIN
```

```
    OPEN c_dbuser FOR
    SELECT * FROM DBUSER WHERE USERNAME LIKE p_username || '%';
    
END;
/
```

对应游标返回在 PostgreSQL 中的实现如下：

```
CREATE or replace FUNCTION employeefunc (rc_out OUT refcursor)
RETURNS refcursor
AS
$$
BEGIN
    OPEN rc_out
    FOR
    SELECT *
    FROM employees;
END;
$$ LANGUAGE plpgsql;
```

对应的存储过程版本如下：

```
CREATE or replace procedure employeefunc (rc_out INOUT refcursor)
language plpgsql
AS
$$
BEGIN
    OPEN rc_out
    FOR
    SELECT *
    FROM employees;
END;
$$ LANGUAGE plpgsql;
```

Oracle 兼容性

匿名块

分布式数据库是否匿名块，需看具体数据库的实现。

在 oracle 中，有两个地方会使用匿名 PL/SQL 块。一是标准化 SQL 脚本；二是批处理代码中，有一些复杂的逻辑使用 PL/SQK 块能够通过避免数据在网络间来回传输极大的提升性能，同时事务更加紧凑。

postgresql 也支持匿名块（因为支持 alter/create xxx if not exists，使得在标准化脚本上匿名块是不必要的），见上文。

序列与自增

oracle 中，可以通过 create sequence 创建序列，通过调用 seq.nextval 和 seq.currval 获取下一个值和当前值。

postgresql 中，同样可以通过 create sequence 创建序列，但是要通过 nextval('seq_1') 和 currval('seq_1')获取下一个值和当前值。

oracle 12c 开始支持 IDENTITY 列，相当于 pg 中的 serial 和 bigserial。

postgresql 也支持 identity 列，且语法和 oracle 完全相同。

```
CREATE TABLE uses_identity (
    id bigint GENERATED ALWAYS AS IDENTITY
                (MINVALUE 0 START WITH 0 CACHE 20)
                PRIMARY KEY,
    ...
);
```

后台会自动创建一个 `uses_identity_id_seq` 序列, 且归属者为 `uses_serial.id`。

删除 `uses_identity` 时，会自动删除对应的序列。

对于新建系统来说，建议使用 `serial` 列，可以同时兼容 `lightdb` 分布式版。

除了序列外，PG 还支持临时序列，和临时表的语义一致。

列默认值支持表达式和函数

postgresql 支持将不带参数的内置函数作为字段默认值，如下：

```
postgres=# create table the_table (
postgres(#         trade_id int not null,
postgres(#         group_id int);
CREATE TABLE
postgres=# alter table the_table add column ts timestamp not null default current_timestamp;
ALTER TABLE
postgres=# alter table the_table add column cust_func int not null default (trade_id+group_id);
ERROR:    cannot use column reference in DEFAULT expression
postgres=#
postgres=#    alter    table    the_table    add    column    cust_date    varchar(32)    not    null    default
(to_char(now(),'YYYY-MM-DD hh24:mi:ss'));
ALTER TABLE
postgres=# insert into the_table(trade_id,group_id) values(1,1);
INSERT 0 1
postgres=# select * from the_table;
 trade_id | group_id |                ts                |         cust_date
----------+----------+----------------------------------+---------------------
```

```
          1 |          1 | 2021-09-20 18:45:47.729581 | 2021-09-20 18:45:47
(1 row)
```

虚拟列

postgresql 支持动态生成列，语法和 oracle 相同。

```
postgres=# alter table the_table add gen_col int generated always as (trade_id + group_id) stored;
ALTER TABLE
postgres=# insert into the_table(trade_id,group_id) values(2,2);
INSERT 0 1
postgres=# select * from the_table;
 trade_id | group_id |                    ts                    |         cust_date          | gen_col
----------+----------+------------------------------------------+----------------------------+---------
          1 |          1 | 2021-09-20 18:45:47.729581 | 2021-09-20 18:45:47 |       2
          2 |          2 | 2021-09-20 18:54:46.768194 | 2021-09-20 18:54:46 |       4
(2 rows)
```

从一定程度上来说，虚拟列填充了列默认值不能引用其它列的缺失。

虚拟列索引

虚拟列索引和普通索引并无区别，pg 直接支持。

不可见索引

在当前版本，postgresql 不支持不可见索引。

动态 SQL

主要用于即席查询和 DDL 场景。
pg 支持动态 SQL，如下：

```
sql := 'GRANT SELECT ON TABLE ' || p_table || ' TO ' || p_role;
EXECUTE sql;

sql := 'SELECT count(*) FROM pg_stat_activity WHERE usename = $1';
EXECUTE sql INTO v_count USING p_role;
```

DML returning

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);
```

```
INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;

UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
INSERT INTO dealer (user_id)
    SELECT id
    FROM rows
    RETURNING id;
```

oracle 则不支持对 insert select 应用 returning 子句。

定时任务

pg 默认不支持定时任务，lightdb 支持定时任务，其功能和 oracle dbms_job 类似，最小间隔为 1 分钟，通过结合 sleep 函数，可以实现秒级调度。

具体可参见 https://github.com/citusdata/pg_cron。

sleep

```
    select pg_sleep(1), pg_sleep(.5);
```

外键

LightDB 支持 ISO SQL 定义的外键，虽然很多开发规范（尤其是某些业务很简单的互联网公司制定的规范）建议不要使用外键约束，实际上是因为 MySQL 早期版本对外键、事务一致性、约束等没有实现，只是语法支持，实际上没有语义作用。在即席查询的场景中，外键约束能够帮助优化器做关联剪除 (join prune)，能够极大的提升性能，尤其是分页查询。

CHECK 约束

如果说外键的作用是为了保证数据一致性的话，那么 check 约束的作用就是为了保证没有脏数据进来，尤其是数据字典字段来说，很多开发人员不喜欢在数据库中控制约束，却不去反思正是因为没有约束，过一段时间之后，各种查询就会出现莫名奇怪的不一致。PG 支持各种约束，如下：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
```

```
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

大字段

在 postgresql 中，文本大字段存储在 TEXT 中，其本身没有长度限制，但是 PG 本身限制每列长度为 1GB。一般来说，除非全文检索需要，不应该存储超过 256 的信息。二进制大字段存储在 bytea 中，长度为 2GB。同理，除非是全文检索需要，建议不要存储在数据库中。

as 关键字

位置	oracle	postgresql
column as column_alias max(column) as max_column	as 关键字可选	as 关键字可选
from (subquery) as subq_alias	as 关键字可选	as 关键字必须
from tab as tab1	as 关键字可选	as 关键字可选

DML 兼容性

别名

在 oracle 中，update 的 set 和 where 子句中可以使用表的别名，例如：

```
update test_ft as x set x.body = replace(body,'关于','头条') where 1=10; --这是合法的。
```

在 postgresql 中，这样写会报语法错误，如下所示：

```
update test_ft as x set x.body = replace(body,'关于','头条') where 1=10;
SQL 错误 [42703]: ERROR: column "x" of relation "test_ft" does not exist
  Position: 25
```

用户不能指定别名，如下：

```
update test_ft as x set body = replace(body,'关于','头条') where 1=10; -- 不报错
```

LightDB 21.3 将支持单表可以包含别名，最大程度的便于用户在 IDE 中使用自动补全的优势。

多表更新

postgresql 采用 UPDATE … FROM … WHERE …的格式。

```
UPDATE "M_HISInterfaceDetail" AS mf
SET "CNName"= tb."CNName",
"DataType" =tb."DataType",
"Remark" =tb."Remark"
from (select * FROM "ss" ) as tb
where mf."Map"=tb."Name"
and mf."InterfaceName"='H_OperationMasters'
and tb."InterfaceName"='OPERATION_MASTER'
```

oracle 多表更新方式：

```
UPDATE T1
SET T1.FMONEY = (select T2.FMONEY from t2 where T2.FNAME = T1.FNAME)
WHERE EXISTS(SELECT 1 FROM T2 WHERE T2.FNAME = T1.FNAME);
```

分页查询

在 oracle 中使用 rownum 两层嵌套子查询实现。

```
SELECT *
  FROM (SELECT tt.*, ROWNUM AS rowno
              FROM (   SELECT t.*
                         FROM emp t
                        WHERE hire_date BETWEEN TO_DATE ('20060501', 'yyyymmdd')
                                            AND TO_DATE ('20060731', 'yyyymmdd')
                     ORDER BY create_time DESC, emp_no) tt
             WHERE ROWNUM <= 20) table_alias
 WHERE table_alias.rowno >= 10;
```

postgresql 和 mysql 支持，支持 Limit 语法

```
SELECT * FROM COMPANY order by id LIMIT 4;   -- 返回 id 最小的前 4 行记录
SELECT * FROM COMPANY order by id LIMIT 3 OFFSET 2;   -- 返回 id 最小的第 3-5 行记录
```

不幸的是，postgresql 不支持 LIMIT skip,rowcount 这样的模式，LightDB 21.3 支持。

limit UPDATE 使用场景

在 postgresql 中，DML 语句不能直接使用分页查询，例如：

```
update test_ft set body = replace(body,'关于','头条') limit 10;
SQL 错误 [42601]: ERROR: syntax error at or near "limit"
  Position: 51
```

```
create table dml_limit(id int,name varchar(100));
insert into dml_limit select x,'name' || x from pg_catalog.generate_series(1,10000) x;
with x(id) as (select * from dml_limit order by id limit 10)
delete from dml_limit where id in (select id from x);
select * from dml_limit;
```

压缩

在非 Exadata 版本中，oracle 支持块级压缩技术，包括基本表压缩和 OLTP 压缩，前者主要用于数据仓库场景，后者用于 OLTP 场景。

```
CREATE TABLE test_tab_1 (
  id              NUMBER(10)     NOT NULL,
  description     VARCHAR2(50)   NOT NULL,
  created_date    DATE           NOT NULL
)
COMPRESS FOR ALL OPERATIONS;
```

Exadata 版本支持混合列式压缩，可以认为是列级存储的体现。

PostgreSQL 对于 TOAST 字段支持压缩，14 之前的版本（以及 LightDB 21c）压缩算法为 LZ，14 版本开始支持 LZ 和 LZ4 两种算法，具体可参见 https://www.cnblogs.com/zhjh256/p/15256306.html。

DDL

不同于 oracle 和其他关系型数据库，pg 的 ddl 支持事务的概念。

DDL WAIT/NOWAIT 子句

虽然 postgresql 不支持 DDL 的 wait 子句，但是其支持语句超时特性和 NOWAIT 子句。通过该特性可以完全实现 wait 的效果，如下：

```
begin;
    set statement_timeout = 50;
    lock table only test in ACCESS EXCLUSIVE MODE;
    set statement_timeout = 0;
   # or SET lock_timeout TO '2s';

    alter table test ....;
    -- do whatever you want, timeout is removed.
commit;
```

这样就能够实现最多等待获取锁 50 毫秒，但是 DDL 本身又不影响。

https://www.depesz.com/2019/09/26/how-to-run-short-alter-table-without-long-locking-concurrent-queries/

UUID 类型

```
select uuid_ns_oid();
```

LightDB 21.2 支持 uuid()和 guid()函数，做为 uuid_ns_oid()的别名。LightDB 21.3 支持 sys_guide()函数。

```
postgres=# select uuid();
uuid
--------------------------------------
 43237d49-1441-4003-8b96-79e8574175c5
(1 row)

postgres=# select guid();
guid
--------------------------------------
 9697e1fb-f2ff-46a9-8bf1-6d2405f5610f
(1 row)

postgres=# select uuid_ns_oid();
uuid_ns_oid
--------------------------------------
 6ba7b812-9dad-11d1-80b4-00c04fd430c8
(1 row)
```

JSON 类型

postgresql 支持 JSON 和 JSONB 两种类型，其中 JSON 按照文本存储，JSONB 按照二进制存储（和 mongdb 的 BSON 存储格式类似），两者均支持 JSON 对象和 JSON 对象（如果会修改，不建议使用）数组。

JSONB 对象支持全文检索，支持根据属性和属性值进行检索，特别适合于站内结构化搜索。属性支持 B 树索引和哈希索引。

```
SELECT artist_data->>'artist_id' AS artist_id ,
artist_data->>'artist' AS artist ,
 jsonb_array_elements(artist_data#>'{albums}')->>'album_title' AS album_title ,
jsonb_array_elements(jsonb_array_elements(artist_data#>'{albums}')#>'{album_tracks}')->>'track
_name' AS song_titles ,
jsonb_array_elements(jsonb_array_elements(artist_data#>'{albums}')#>'{album_tracks}')->>'track
_id' AS song_id
FROM v_json_artist_data
WHERE artist_data->>'artist' = 'Metallica' ORDER BY album_title , song_id ;
```

	artist_id	artist	album_title	song_titles	song_id
1	50	Metallica	...And Justice For All	Sad But True	1802
2	50	Metallica	...And Justice For All	The Unforgiven	1804
3	50	Metallica	...And Justice For All	Don't Tread On Me	1806
4	50	Metallica	...And Justice For All	Nothing Else Matters	1808
5	50	Metallica	...And Justice For All	The God That Failed	1810
6	50	Metallica	...And Justice For All	The Struggle Within	1812
7	50	Metallica	...And Justice For All	Helpless	1813
8	50	Metallica	...And Justice For All	The Wait	1815
9	50	Metallica	...And Justice For All	Last Caress/Green Hell	1817
10	50	Metallica	...And Justice For All	Blitzkrieg	1819
11	50	Metallica	...And Justice For All	The Prince	1821
12	50	Metallica	...And Justice For All	So What	1823
13	50	Metallica	...And Justice For All	Overkill	1825
14	50	Metallica	...And Justice For All	Stone Dead Forever	1827
15	50	Metallica	...And Justice For All	Hit The Lights	1829
16	50	Metallica	...And Justice For All	Motorbreath	1831
17	50	Metallica	...And Justice For All	(Anesthesia) Pulling Teeth	1833

200 row(s) fetched - 47ms

```
CREATE INDEX idx_1 ON jsonb.actor USING GIN (jsondata);
```

通过使用 GIN 索引，通常性能可以提升数十倍。

```
CREATE UNIQUE INDEX actor_id_2 ON jsonb.actor((CAST(jsondata->>'actor_id' AS INTEGER)));
```

正则表达式

postgresql 支持 SIMILAR TO 和 POSIX 正则表达式，包括普通的模式匹配、替换、子串截取等，下面列出了 postgresql 支持的正则函数和操作符。

```
'abc' SIMILAR TO 'abc'              true
'abc' SIMILAR TO 'a'                false
'abc' SIMILAR TO '%(b|d)%'          true
'abc' SIMILAR TO '(b|c)%'           false
'-abc-' SIMILAR TO '%\mabc\M%'    true
'xabcy' SIMILAR TO '%\mabc\M%'    false
regexp_replace
regexp_match
regexp_matches
regexp_split_to_table
regexp_split_to_array
```

详细可见 https://www.postgresql.org/docs/13/functions-matching.html。

注：虽然我们在应用中广泛使用正则表达式，但是一般不推荐在数据库中使用正则表达式匹配和其它相关操作。

全文检索

LightDB 支持非常强大和精确的全文检索，详细可参见 https://www.cnblogs.com/zhjh256/p/15357837.html。

不可见列

在 oracle 12c+，支持不可见列特性。

在 postgresql 中，不支持不可见列特性，lightdb 对自动更新时间戳、内置主键列也是采用不可见列实现。

分析函数及 grouping sets

使用分析函数，开发人员可以通过更清晰、简洁的 SQL 代码执行复杂分析。原来需要几十行甚至上百行代码完成的逻辑现在可以使用一条 SQL 语句表示复杂任务，编写和维护速度更快、效率更高。数据库中分析支持的处理优化可大幅提高查询性能。以前需要自联接或复杂过程处理的操作现在可以用原生 SQL 执行。以分组排序为例，如果有下列表：

```
mysql> select * from rank_over;
+------+-------+---------------------+
| id   | subid | curd                |
+------+-------+---------------------+
|    1 |     1 | 2018-09-24 00:47:12 |
|    2 |     1 | 2018-09-24 00:47:38 |
|    3 |     1 | 2018-09-24 00:47:42 |
|    4 |     2 | 2018-09-24 00:47:50 |
|    5 |     2 | 2018-09-24 00:47:54 |
|    6 |     3 | 2018-09-24 00:48:00 |
|    7 |     4 | 2018-09-24 00:48:06 |
|    8 |     3 | 2018-09-24 01:12:10 |
|    9 |     2 | 2018-09-24 01:12:11 |
+------+-------+---------------------+
```

现在要取出每个 subid 下 curd 最大的 1 条。

使用分析函数只需要很简单的 SQL：

```
select t.id,t.subid,t.curd
from(SELECT id,subid,curd,RANK() OVER(PARTITION BY subid ORDER BY curd DESC) RK
    FROM rank_over) t
where t.RK<2
```

如果没有分析函数，则要复杂得多，如下：

```
select t1.*  from
(select (@rowNum1:=@rowNum1+1) as rowNo,id,subid,curd from rank_over a,(Select (@rowNum1 :=0)) b order by a.subid,a.curd desc) t1 left join
(select (@rowNum2:=@rowNum2+1) as rowNo,id,subid,curd from rank_over c,(Select
```

```
(@rowNum2 :=1)) d order by c.subid,c.curd desc) t2 on t1.rowNo=t2.rowNO
where t1.subid<>t2.subid or t2.subid is null
```

它们的结果都是：

```
+-------+------+-------+---------------------+
| rowNo | id   | subid | curd                |
+-------+------+-------+---------------------+
|     1 |    3 |     1 | 2018-09-24 00:47:42 |
|     4 |    9 |     2 | 2018-09-24 01:12:11 |
|     7 |    8 |     3 | 2018-09-24 01:12:10 |
|     9 |    7 |     4 | 2018-09-24 00:48:06 |
+-------+------+-------+---------------------+
4 rows in set (0.00 sec)
```

分析函数不仅用于提高开发效率，而且数据库优化器通常会对分析函数的执行进行优化，典型的即是避免了对基表的二次扫描。

postgresql 将分析函数进行了分组，支持包括内置窗口函数 https://www.postgresql.org/docs/current/functions-window.html，一般聚合函数 https://www.postgresql.org/docs/current/functions-aggregate.html#FUNCTIONS-AGGREGATE-TABLE、统计聚合函数 https://www.postgresql.org/docs/current/functions-aggregate.html#FUNCTIONS-AGGREGATE-STATISTICS-TABLE。

和分析函数一样，GROUPING SETS 的用途是提升性能。

GROUP BY GROUPING SETS--指定分组

```
test=# SELECT   country, product_name, sum(amount_sold)
     FROM    t_sales
     GROUP BY GROUPING SETS ((1), (2))
     ORDER BY 1, 2;
 country   | product_name | sum
-----------+--------------+-----
 Argentina |              | 137
 Germany   |              | 104
 USA       |              | 373
           | Hats         | 323
           | Shoes        | 291
(5 rows)
```

GROUP BY ROLLUP--分层聚合

```
test=# SELECT   country, product_name, sum(amount_sold)
     FROM    t_sales
     GROUP BY ROLLUP (1, 2)
```

```
    ORDER BY 1, 2;
country    | product_name | sum
-----------+--------------+-----
Argentina | Hats         | 111
Argentina | Shoes        |  26
Argentina |              | 137
Germany    | Hats         |  41
Germany    | Shoes        |  63
Germany    |              | 104
USA        | Hats         | 171
USA        | Shoes        | 202
USA        |              | 373
           |              | 614
(10 rows)
```

GROUP BY CUBE--笛卡尔聚合

```
test=# SELECT   country, product_name, sum(amount_sold)
    FROM    t_sales
    GROUP BY CUBE (1, 2)
    ORDER BY 1, 2;
country    | product_name | sum
-----------+--------------+-----
Argentina | Hats         | 111
Argentina | Shoes        |  26
Argentina |              | 137
Germany    | Hats         |  41
Germany    | Shoes        |  63
Germany    |              | 104
USA        | Hats         | 171
USA        | Shoes        | 202
USA        |              | 373
           | Hats         | 323
           | Shoes        | 291
           |              | 614
(12 rows)
```

CTE 与递归 CTE

CTE 主要有两个用处：一、复用子查询；二、性能优化，因为当前版本尚不支持 no_merge 优化器提示，如果希望有些非相关子查询执行 no_merge 物化，可以使用 CTE 实现。

注：对于优化器提示，我们目前最优先的工作是支持在任何子块包含优化器提示，而不仅限于顶层。

```
WITH A as MATERIALIZED/*+ pg12 开始支持*/ (SELECT * from foo where <complicated stuff>),
     B as MATERIALIZED (SELECT * from foo where <other complicated stuff>)
SELECT * from A JOIN B ON <complicated stuff>
order by a_thing;
```

start with...connect by prior

oracle 层次查询

```
select name, age from user_test connect by prior user_id=parent_id start with user_id='a';
```

postgresql 层次查询

```
with recursive cte_name as
(select u1.name, u1.user_id, u1.age from user_test u1 where user_id='a'
UNION ALL select u2.name, u2.user_id, u2.age from user_test u2
join cte_name on cte_name.user_id=u2.parent_id) select name,age from cte_name;
```

sys_connect_by_path

可参考 https://www.cnblogs.com/mxly/p/9458569.html

connect by level

虽然 pg 不支持 connect by level 语法，不过其提供的 generate_series 表函数，能够实现 connect by level 的功能，典型的是造测试数据，如下：

```
postgres=# create table x as select id,'name' || id from generate_series(1,100000000) id;
SELECT 100000000
```

相当于 `select 1 id,'name' || id from dual connect by level < 100000001`

集合操作符

postgresql 支持 INTERSECT、EXCEPT（oracle、MySQL 是 MINUS）。

分区

都知道，在单表数据量巨大时有效采用分区能够极大的提高 SQL 语句的性能（但是也需要注意的是，因为 postgresql 本质上实现了 oracle 分区本地索引的概念，所以对非唯一索引搜索性能相比非分区而言会降低），同时降低维护复杂性。postgresql 在早期版本通过继承实现分区，但是在 10 开始采用了原生分区，并逐渐支持并行执行和运行时剪除。和 oracle 中一样，分区在很多偏向于 OLTP 的 SQL 语句中通常是会降低一些性能的，这里给

出一个分区对性能影响测试的链接。
https://www.enterprisedb.com/postgres-tutorials/how-benchmark-partition-table-performance

LightDB 支持范围、列表、哈希分区以及它们的组合。

在当前版本中，pg 不支持 oracle 的自动间隔分区。

闪回查询

pg 默认不支持闪回特性，可以考虑使用基于时间点的恢复。

dump

```
pg_dump -h localhost -Fc testdb > /home/postgres/dump.sql
```

-Fc 代表自定义格式，-Fd 代表目录格式（并行导出仅支持该模式）。

```
pg_restore -h localhost testdb < /home/postgres/dump.sql
```

pg_dumpall 用于导出整个集群中的所有数据库，一般不建议使用。如果需要备份整个库，建议使用 pg_probackup。

对于大数据库，可以使用并行模式。

```
pg_dump -j num -F d -f out.dir dbname
```

注：对于并行导出，仅支持目录格式，即选项-Fd。

物理备份

postgresql 提供了客户端工具 lt_basebackup 用于物理备份，一般来说，基于功能和易用性考虑，优先使用 lt_probackup，它也是基于 pg_basebackup，但是提供了比 pg_basebackup 更加丰富的功能，包括备份元数据管理、全量备份、增量备份，以及基于时间点的恢复，类似于 Oracle RMAN。

LightDB EM 提供了自动备份功能、集中备份管理功能以及远程备份。

临时表

```
postgres=# create table x1 as select id,'name' || id from generate_series(1,10000000) id;
SELECT 10000000
Time: 13535.121 ms (00:13.535)
postgres=# drop table x1;
DROP TABLE
Time: 181.081 ms
postgres=# create table x1 as select id,'name' || id from generate_series(1,10000000) id;
SELECT 10000000
Time: 12707.884 ms (00:12.708)
```

```
postgres=# create temporary table x1 as select id,'name' || id from generate_series(1,10000000) id;
```

```
SELECT 10000000
Time: 8052.643 ms (00:08.053)
postgres=# drop table x1;
DROP TABLE
Time: 7.414 ms
postgres=# create temporary table x1 as select id,'name' || id from generate_series(1,10000000) id;
SELECT 10000000
Time: 8120.009 ms (00:08.120)
postgres=# drop table x1;
DROP TABLE
Time: 8.081 ms
postgres=# create temporary table x1 as select id,'name' || id from generate_series(1,10000000) id;
SELECT 10000000
Time: 8131.810 ms (00:08.132)

postgres=# create unlogged table x1 as select id,'name' || id from generate_series(1,10000000) id;
SELECT 10000000
Time: 8720.281 ms (00:08.720)
postgres=# drop table x1;
DROP TABLE
Time: 169.182 ms
postgres=# create unlogged table x1 as select id,'name' || id from generate_series(1,10000000) id;
SELECT 10000000
Time: 8684.294 ms (00:08.684)
```

可见临时表比 unlogged 表快 6%左右，unlogged 表比常规表快 60%左右。

由于 pg 即使开启了流复制模式，也可以允许 unlogged 表存在（即没有 force logging 的概念），所以在批处理系统中，中间表采用 unlogged 表能够极大的提升性能，结果集都缓存在 pagecache 中。

FDW(Foreign Data Wrapper)，集数据库连接、外部表及异构数据库访问于一身

外部表

和 oracle 一样，lightdb 支持外部表的概念，在一些需要一次性访问的

```
(postgres@[local]:5000) [postgres] > create extension file_fdw;
CREATE EXTENSION
Time: 752.715 ms
(postgres@[local]:5000) [postgres] > create server srv_file_fdw foreign data wrapper file_fdw;
CREATE SERVER
Time: 23.317 ms
(postgres@[local]:5000) [postgres] > create foreign table t_csv ( a int, b varchar(50) )
```

```
                                    server srv_file_fdw
                                    options ( filename '/var/tmp/data.csv', format 'csv' );
CREATE FOREIGN TABLE
Time: 74.843 ms
(postgres@[local]:5000) [postgres] > select count(*) from t_csv;
 count
-------
  1000
(1 row)

Time: 0.707 ms
(postgres@[local]:5000) [postgres] > create table t_csv2 as select * from t_csv;
SELECT 1000
Time: 147.859 ms
(postgres@[local]:5000) [postgres] > insert into t_csv values (-1,'a');
ERROR:   cannot insert into foreign table "t_csv"
Time: 18.113 ms
(postgres@[local]:5000) [postgres] > explain select count(*) from t_csv;
                          QUERY PLAN
---------------------------------------------------------------------
 Aggregate   (cost=171.15..171.16 rows=1 width=0)
   ->   Foreign Scan on t_csv   (cost=0.00..167.10 rows=1621 width=0)
        Foreign File: /var/tmp/data.csv
        Foreign File Size: 38893
(4 rows)

Time: 0.521 ms
```

数据库链接

注：数据库连接的性能比较差，如果对实时性能要求比较高，推荐采用相关同步技术实时同步到 lightdb。

到其他 lightdb/postgresql 的链接

```
lightdb=# create extension postgres_fdw;

CREATE EXTENSION
lightdb=# CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host
'xx.xx.xx.xx', dbname 'postgres', port '5444');
CREATE SERVER
```

```
lightdb=# CREATE USER MAPPING FOR lightdb SERVER myserver OPTIONS (user 'lightdb', password 'l_123456');
CREATE USER MAPPING
lightdb=# CREATE FOREIGN TABLE test_foreign ( id     int, name   text) SERVER myserver;
CREATE FOREIGN TABLE
```

到 MySQL 数据库的链接

相当于 oracle 中的透明网关。
LightDB-X 包含了 MySQL 数据库链接扩展 mysql_fdw，并默认启用。
需要先安装 MySQL 客户端，配置 LD_LIBRARY_PATH 环境变量。

到 Oracle 数据库的链接

相当于 oracle 中的透明网关。
LightDB-X 包含了 Oracle 数据库扩展 oracle_fdw（默认不启用）。
首先安装 Oracle 客户端，配置 ORACLE_HOME 和 LD_LIBRARY_PATH 环境变量。

mybatis 与 jdbctemplate 支持

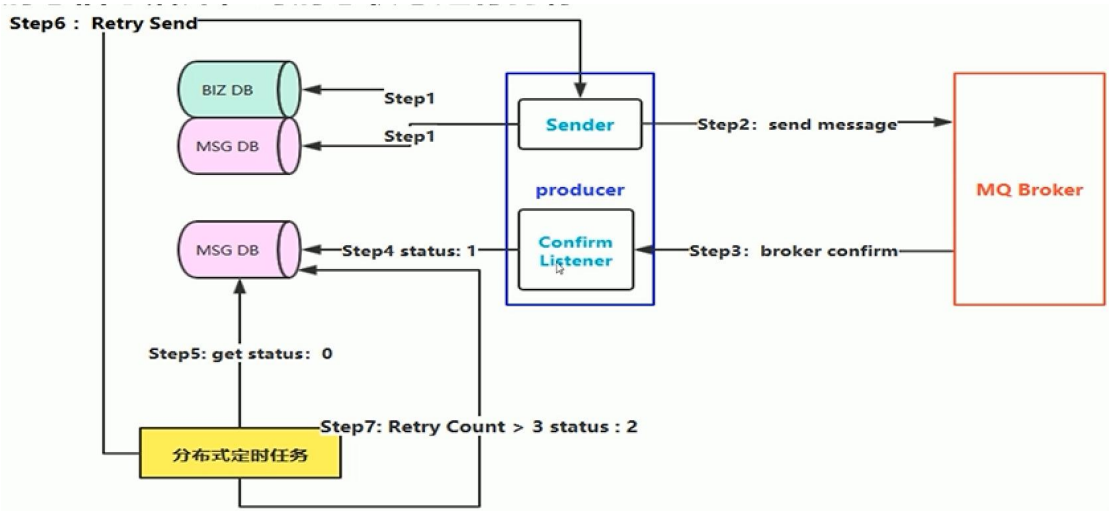
在 SQL 层面，LightDB 完全兼容 PostgreSQL 协议和语法，因此也完全支持 mybatis 和 jdbctemplate，其使用方式和 oracle、mysql 并无差异。

LightDB 的其他实用特性

除了上述开发经常使用到的特性外，LightDB 还包含一些 Oracle、MySQL 中所没有，但是开发经常希望包含的特性，以及一些为了补偿历史设计遗留问题无法变更模型而引入的特性。

支持事务的可靠事件推送

在 MQ 中，最复杂的环节通常不是消息的有序性，而是保持有序情况下的可靠性，以及怎么保证消息日志表和消息本身成事务，这通常会导致非常复杂的控制逻辑，虽然失败发生的概率低于 0.1%。通常类似如下：



在使用缓存的应用中，保持缓存数据的一致性是非常重要的，多个端都可能修改数据库中的数据，如何确保修改的数据立刻推送给缓存，在微服务架构下大量的端时尤其如此。
https://www.cnblogs.com/zhjh256/p/15485431.html

每秒 10 万+可靠计数器

```
postgres=# create sequence idx_seq;
CREATE SEQUENCE

postgres=# \timing on
Timing is on.
postgres=# DO $$
DECLARE i int;
BEGIN
FOR i IN 1..1000000 LOOP
execute 'select nextval(''idx_seq'')';
END LOOP;
END$$;
DO
Time: 6321.316 ms (00:06.321)
```

50 万+ QPS 强一致性可靠 K/V

得益于 LightDB 的多进程架构, LightDB 能轻松实现 10 万级的 TPS, 50 万的 K/V 查询，用户完全可以使用 LightDB 代替 Redis、Couchbase 等 K/V，不仅能够简化技术栈，还能得到更高的可靠性。在非易失性内存、千兆网络、亿级存量记录上（32C Gold 6250，8KB IOPS 53000 19us），只读和 LightDB 在可靠模式和缓存模式下 96 并发连接 QPS 和 TPS 如下。

```
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
```

```
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

可靠模式-读写

```
progress: 60.0 s, 94436.1 tps, lat 1.014 ms stddev 0.403
progress: 61.0 s, 95780.3 tps, lat 1.000 ms stddev 0.392
progress: 62.0 s, 92012.6 tps, lat 1.041 ms stddev 0.495
progress: 63.0 s, 94249.1 tps, lat 1.016 ms stddev 0.436
progress: 64.0 s, 94744.6 tps, lat 1.011 ms stddev 0.421
progress: 65.0 s, 95199.3 tps, lat 1.006 ms stddev 0.408
progress: 66.0 s, 94907.8 tps, lat 1.009 ms stddev 0.416
progress: 67.0 s, 94927.8 tps, lat 1.009 ms stddev 0.402
progress: 68.0 s, 95599.3 tps, lat 1.001 ms stddev 0.456
progress: 69.0 s, 92150.0 tps, lat 1.039 ms stddev 0.474
progress: 70.0 s, 82816.8 tps, lat 1.156 ms stddev 3.202
progress: 71.0 s, 94684.7 tps, lat 1.011 ms stddev 0.390
progress: 72.0 s, 94771.8 tps, lat 1.011 ms stddev 0.400
```

只读

```
progress: 20.0 s, 565186.6 tps, lat 0.167 ms stddev 0.156
progress: 21.0 s, 565714.6 tps, lat 0.167 ms stddev 0.149
progress: 22.0 s, 543750.9 tps, lat 0.174 ms stddev 0.155
progress: 23.0 s, 568791.4 tps, lat 0.166 ms stddev 0.155
progress: 24.0 s, 570629.0 tps, lat 0.166 ms stddev 0.149
progress: 25.0 s, 572086.6 tps, lat 0.165 ms stddev 0.144
progress: 26.0 s, 573350.4 tps, lat 0.165 ms stddev 0.145
progress: 27.0 s, 571241.7 tps, lat 0.165 ms stddev 0.154
progress: 28.0 s, 573166.0 tps, lat 0.165 ms stddev 0.145
progress: 29.0 s, 572714.2 tps, lat 0.165 ms stddev 0.145
progress: 30.0 s, 557148.4 tps, lat 0.170 ms stddev 0.257
```

缓存模式-查询+新增+修改

```
progress: 19.0 s, 111209.4 tps, lat 0.860 ms stddev 0.522
progress: 20.0 s, 115148.9 tps, lat 0.830 ms stddev 0.489
progress: 21.0 s, 117877.8 tps, lat 0.811 ms stddev 0.423
progress: 22.0 s, 117594.4 tps, lat 0.813 ms stddev 0.456
progress: 23.0 s, 118071.1 tps, lat 0.810 ms stddev 0.428
progress: 24.0 s, 118218.5 tps, lat 0.808 ms stddev 0.432
progress: 25.0 s, 118275.5 tps, lat 0.808 ms stddev 0.454
progress: 26.0 s, 117323.6 tps, lat 0.815 ms stddev 0.495
progress: 27.0 s, 118241.6 tps, lat 0.808 ms stddev 0.424
progress: 28.0 s, 118294.7 tps, lat 0.808 ms stddev 0.432
progress: 29.0 s, 118111.3 tps, lat 0.809 ms stddev 0.442
progress: 30.0 s, 118549.0 tps, lat 0.806 ms stddev 0.450
progress: 31.0 s, 118634.3 tps, lat 0.806 ms stddev 0.442
```

==========下面为普通 SSD（8KB IOPS 15000 65us），32C E5-2620 v2

缓存模式-只读

```
top - 10:56:21 up 132 days, 18:53,  5 users,  load average: 20.57, 12.69, 6.30
Tasks: 487 total,  26 running, 461 sleeping,   0 stopped,   0 zombie
%Cpu(s): 40.8 us, 17.7 sy,  0.0 ni, 36.0 id,  0.0 wa,  0.0 hi,  5.4 si,  0.0 st
KiB Mem : 39603776+total, 60995492 free,  6324108 used, 32871814+buff/cache
KiB Swap: 33554428 total, 33244592 free,   309836 used. 13682601+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26974 root      20   0       0      0      0 R  52.6  0.0 399:43.56 [nfsd]
26973 root      20   0       0      0      0 R  43.0  0.0 241:56.93 [nfsd]
26972 root      20   0       0      0      0 R  28.1  0.0 123:34.37 [nfsd]
 2732 lightdb   20   0   33.1g   1.1g   1.1g R  18.5  0.3   0:07.81 lightdb: lightdb postgres 10.20.30.218(51758) SELECT
 2755 lightdb   20   0   33.1g   1.1g   1.1g S  18.5  0.3   0:07.56 lightdb: lightdb postgres 10.20.30.218(51804) idle
 2795 lightdb   20   0   33.1g   1.1g   1.1g S  17.9  0.3   0:07.46 lightdb: lightdb postgres 10.20.30.218(51876) idle
 2748 lightdb   20   0   33.1g   1.0g   1.0g S  17.5  0.3   0:07.46 lightdb: lightdb postgres 10.20.30.218(51790) idle
 2792 lightdb   20   0   33.1g   1.0g   1.0g S  17.5  0.3   0:07.35 lightdb: lightdb postgres 10.20.30.218(51870) idle
 2733 lightdb   20   0   33.1g   1.1g   1.0g S  16.9  0.3   0:07.37 lightdb: lightdb postgres 10.20.30.218(51760) idle
 2739 lightdb   20   0   33.1g   1.1g   1.1g S  16.9  0.3   0:07.60 lightdb: lightdb postgres 10.20.30.218(51774) idle
 2798 lightdb   20   0   33.1g   1.0g   1.0g S  16.9  0.3   0:06.81 lightdb: lightdb postgres 10.20.30.218(51882) idle
```

```
progress: 48.0 s, 70381.7 tps, lat 1.305 ms stddev 10.415
progress: 49.0 s, 68078.5 tps, lat 1.390 ms stddev 11.969
progress: 50.0 s, 85574.4 tps, lat 1.159 ms stddev 9.539
progress: 51.0 s, 86892.6 tps, lat 1.065 ms stddev 7.191
progress: 52.0 s, 83069.5 tps, lat 1.139 ms stddev 7.881
progress: 53.0 s, 93423.1 tps, lat 1.031 ms stddev 6.655
progress: 54.0 s, 89754.5 tps, lat 1.049 ms stddev 8.815
progress: 55.0 s, 82092.7 tps, lat 1.078 ms stddev 8.859
progress: 56.0 s, 91807.0 tps, lat 1.124 ms stddev 7.723
progress: 57.0 s, 94286.7 tps, lat 0.933 ms stddev 7.135
progress: 58.0 s, 81530.6 tps, lat 1.232 ms stddev 10.186
progress: 59.0 s, 90130.0 tps, lat 1.008 ms stddev 5.887
```

```
[lightdb@hs-10-20-30-218 ~]$ ping 10.20.30.217
PING 10.20.30.217 (10.20.30.217) 56(84) bytes of data.
64 bytes from 10.20.30.217: icmp_seq=1 ttl=64 time=0.589 ms
64 bytes from 10.20.30.217: icmp_seq=2 ttl=64 time=0.254 ms
64 bytes from 10.20.30.217: icmp_seq=3 ttl=64 time=0.354 ms
64 bytes from 10.20.30.217: icmp_seq=4 ttl=64 time=0.411 ms
64 bytes from 10.20.30.217: icmp_seq=5 ttl=64 time=0.716 ms
^C
--- 10.20.30.217 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 0.254/0.464/0.716/0.168 ms
```

缓存模式--查询+新增+修改

```
progress: 108.0 s, 19184.7 tps, lat 5.178 ms stddev 20.596
progress: 109.0 s, 15949.6 tps, lat 5.800 ms stddev 22.176
progress: 110.0 s, 15714.4 tps, lat 6.366 ms stddev 26.690
progress: 111.0 s, 12990.8 tps, lat 6.934 ms stddev 29.236
progress: 112.0 s, 14499.7 tps, lat 6.819 ms stddev 27.802
progress: 113.0 s, 12518.5 tps, lat 7.967 ms stddev 29.995
progress: 114.0 s, 15957.2 tps, lat 5.412 ms stddev 20.708
progress: 115.0 s, 20831.8 tps, lat 5.026 ms stddev 19.495
progress: 116.0 s, 19171.4 tps, lat 4.929 ms stddev 20.234
progress: 117.0 s, 19547.7 tps, lat 4.730 ms stddev 17.846
progress: 118.0 s, 12552.2 tps, lat 7.335 ms stddev 28.623
progress: 119.0 s, 13972.1 tps, lat 7.102 ms stddev 30.864
progress: 120.0 s, 10851.6 tps, lat 9.194 ms stddev 35.519
transaction type: <builtin: simple update>
scaling factor: 1000
query mode: prepared
number of clients: 96
number of threads: 16
duration: 120 s
number of transactions actually processed: 2138047
latency average = 5.357 ms
latency stddev = 20.960 ms
tps = 17779.586307 (including connections establishing)
tps = 17788.504814 (excluding connections establishing)
```

可靠模式

```
Tasks: 487 total,   7 running, 480 sleeping,   0 stopped,   0 zombie
%Cpu(s): 10.7 us,  9.0 sy,  0.0 ni, 74.7 id,  3.7 wa,  0.0 hi,  1.9 si,  0.0 st
KiB Mem : 39603776+total, 28404260 free,  4060724 used, 36357276+buff/cache
KiB Swap: 33554428 total, 33244592 free,   309836 used. 13848268+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26974 root      20   0       0      0      0 S  43.2  0.0 420:47.45 nfsd
26973 root      20   0       0      0      0 S  34.3  0.0 256:27.84 nfsd
```

```
progress: 36.0 s, 721.8 tps, lat 93.224 ms stddev 162.706
progress: 37.0 s, 3043.0 tps, lat 41.925 ms stddev 81.480
progress: 38.0 s, 2676.7 tps, lat 33.677 ms stddev 108.732
progress: 39.0 s, 5227.5 tps, lat 19.208 ms stddev 54.900
progress: 40.0 s, 2302.4 tps, lat 34.240 ms stddev 39.116
progress: 41.0 s, 1755.4 tps, lat 61.187 ms stddev 133.466
progress: 42.0 s, 8057.5 tps, lat 13.112 ms stddev 32.344
progress: 43.0 s, 3940.1 tps, lat 22.401 ms stddev 45.030
progress: 44.0 s, 176.0 tps, lat 313.881 ms stddev 228.200
progress: 45.0 s, 2988.9 tps, lat 47.702 ms stddev 124.543
progress: 46.0 s, 5362.9 tps, lat 17.755 ms stddev 39.978
progress: 47.0 s, 6615.1 tps, lat 14.224 ms stddev 46.895
progress: 48.0 s, 2756.3 tps, lat 35.261 ms stddev 58.372
progress: 49.0 s, 607.0 tps, lat 163.125 ms stddev 206.680
progress: 50.0 s, 3676.5 tps, lat 25.890 ms stddev 34.145
```

数组类型

数组的用途之一在于保存权限位，1 代表有权限，0 代表无权限。也可以用来保存 IP、

电话等通常由多个部分组成，但是基本上一起使用，但是可能独立分段查询或一起查询的属性。相比使用 substr、instr 进行判断，数组提供了原生的操作符。相比 JSON 类型，数组类型的空间使用率更高。

java 插入 array 类型的数据, 参见 https://www.cnblogs.com/zhjh256/p/15227861.html。

内置乐观锁

相比 oracle 和 mysql 中，为了在实现乐观锁时避免 ABA 必须增加一个版本号，lightdb 内置列 xmin、xmax 可以作为乐观锁机制，并且自动保证可靠性。

会话 1：

```
select xmin,xmax,f.* from the_table f;
xmin|xmax      |trade_id|group_id|ts                       |cust_date                |gen_col|
----+--------+--------+--------+-------------------------+-------------------------+-------+
586 |0|           1|          1|2021-09-20 18:45:47|2021-09-20 18:45:47|          2|
588 |0|           2|          2|2021-09-20 18:54:46|2021-09-20 18:54:46|          4|
```

会话 2：

```
begin;
update the_table set trade_id = 1;
select xmin,xmax,f.* from the_table f;
xmin      |xmax|trade_id|group_id|ts                       |cust_date                |gen_col|
--------+----+--------+--------+-------------------------+-------------------------+-------+
23020551|0    |          1|          1|2021-09-20 18:45:47|2021-09-20 18:45:47|          2|
23020551|0    |          1|          2|2021-09-20 18:54:46|2021-09-20 18:54:46|          3|
```

会话 1：

```
select xmin,xmax,f.* from the_table f;
xmin|xmax      |trade_id|group_id|ts                       |cust_date                |gen_col|
----+--------+--------+--------+-------------------------+-------------------------+-------+
586 |23020551|          1|          1|2021-09-20 18:45:47|2021-09-20 18:45:47|          2|
588 |23020551|          2|          2|2021-09-20 18:54:46|2021-09-20 18:54:46|          4|
```

会话 2：

```
rollback;
select xmin,xmax,f.* from the_table f;
xmin|xmax      |trade_id|group_id|ts                       |cust_date                |gen_col|
----+--------+--------+--------+-------------------------+-------------------------+-------+
586 |23020551|          1|          1|2021-09-20 18:45:47|2021-09-20 18:45:47|          2|
588 |23020551|          2|          2|2021-09-20 18:54:46|2021-09-20 18:54:46|          4|
```

注：虽然会话 2 回滚，但是 xmax 此时也会变化，直到下一次修改后提交才为 0。如果是 commit，这 xmax 会为 0。这个结果的原因在于 pg 只要执行修改，就会新创建记录，但是此时记录是不可见的，也可能会回滚。底层 old tuple 有个指针指向 new tuple，所以 old tuple 和 new tuple 除了维护了 xmin 和 xmax 外，还都维护了事务是否已经提交（避免

vacuum 每次跑去 pg_xact 和 pg_subtrans 问事务是否提交了？），这样执行器查找的时候知道如何检查是否满足 MVCC 的一致性要求。从旧指向新的好处是避免了从 lp 到 tuple 的指针修改，lp 的并发性从读写锁变到了额外的判断，可以大大增加 TPS。

这样会话 1 就知道，在此期间有其他会话尝试修改过记录，即使最后回滚了。

分布式锁

https://developer.aliyun.com/article/78911
https://github.com/digoal/blog/blob/master/201610/20161018_01.md

遗憾的是，当前 advisory lock 不支持高可用，advisory lock 仅在其申请的实例有效，一旦发生了 failover，分布式锁就会丢失。

因为 LightDB TPS 能够达到足够高，因此可以通过乐观锁+一次性 cron 定时任务实现高可用分布式锁。

内置隐藏主键

可以 alter table、create table 的时候通过 with update 子句指定。

```
ALTER TABLE table_name with primary key;
CREATE TABLE table_name (...) with primary key;
```

会自动创建 ltapk 字段，会自动创建索引。select * 时查询不到该字段（类似于隐藏字段），insert into table_name values()同理。是为了弥补一些表早期设计的时候没有考虑后期维护，但是后面增加字段又不方便的场景。

```
postgres=# create table test_with_timestamp(id int,unq_not_null_col int not null) with primary key;
CREATE TABLE
postgres=# alter table test_with_timestamp with update current_timestamp;
ALTER TABLE
postgres=# select * from test_with_timestamp ;
 id | unq_not_null_col
----+------------------
(0 rows)

postgres=# insert into test_with_timestamp values(1,1);
INSERT 0 1
postgres=# insert into test_with_timestamp values(1,8);
INSERT 0 1
postgres=# select * from    test_with_timestamp;
 id | unq_not_null_col
----+------------------
  1 |                1
  1 |                8
(2 rows)
```

自更新时间戳字段

为了最大程度的兼容 mysql,lightdb 支持 mysql 时间戳字段的 on update current_timestamp 特性。

可以 create table、alter table 的时候在列上自动更新时间戳；也可以 alter table、create table 的时候通过 with update 子句指定。

```
ALTER TABLE table_name WITH UPDATE CURRENT_TIMESTAMP;
CREATE TABLE table_name (...) WITH UPDATE CURRENT_TIMESTAMP;
```

后者会自动创建 `ltuat` 字段，可以创建索引。`select * ` 时查询不到该字段（类似于隐藏字段），`insert into table_name values()`同理。是为了弥补一些表早期设计的时候没有考虑后期维护，但是后面增加字段又不方便的场景。

```
postgres=# create table test_with_timestamp(id int) with update current_timestamp;
CREATE TABLE
postgres=# select * from test_with_timestamp ;
 id
----
(0 rows)

postgres=# \dS+ test_with_timestamp
                                                Table "public.test_with_timestamp"
 Column |            Type             | Collation | Nullable |      Default      | Storage | Stats target | Description
--------+-----------------------------+-----------+----------+-------------------+---------+--------------+-------------
 id     | integer                     |           |          |                   | plain   |              |
 ltaut  | timestamp without time zone |           |          | CURRENT_TIMESTAMP | plain   |              |
Access method: heap

postgres=# insert into test_with_timestamp values(1);
INSERT 0 1
postgres=# select * from test_with_timestamp ;
 id
----
  1
(1 row)

postgres=# select id,ltaut from test_with_timestamp ;
 id |           ltaut
----+----------------------------
  1 | 2021-11-09 09:39:15.654441
(1 row)
```

原生高可用与负载均衡

LightDB 原生集成了高可用模块，支持单中心和双中心部署，能够自动预防脑裂。

支持一主多副，副本可用于读写分离。

在高可用级别上支持最大保护模式、最大可用模式、最大性能模式以及最大一致性模式（LightDB 21.3 特性），比 Oracle 多一级。

LightDB 高可用安装默认不提供负载均衡特性，如果用户希望保证强一致的读写分离，需要使用最大一致性模式。

LightDB 22c 将提供原生负载均衡和多主。

LightDB 高可用除了支持 SQL 外，上述提及的可靠通知、计数器、K/V、乐观锁等均原生支持高可用。

LightDB 最佳实践

- 通用规范，如与 LightDB 推荐冲突，以 LightDB 推荐为准 https://www.cnblogs.com/Ferda/archive/2020/07/16/13322465.html
- 记录大小不要超过 TOAST_TUPLE_TARGET 2kb，否则会被拆分到 TOAST 表
- 为了长期保证数据一致性，在应该定义外键的地方不要犹豫某些开发规范建议或禁止使用外键，外键能够让数据库模型更加健壮，让数据长期更加一致。在查询时性能更高，各个环境也能避免动不动被发现数据不一致。不允许使用外键跟面向对象中不允许使用继承没什么区别。
- 使用自增代替 UUID，产生的 REDO 更低，可以使用 lightdb 的 serial 类型或者兼容 mysql 的 auto_increment 特性
- OLTP 单表不要超过 1000 万行*30 字段（300 字节），否则分区
- DSS 表单表不要超过 2GB-8GB（NVMe、非易失性内存），否则分区
- 每个表包含单字段整型主键和最后修改时间戳，可以使用 lightdb 的 on update current-timestamp 特性
- 不要在业务表中存储大对象，使用文件存储，在数据库中存储 URL
- FILLFACTOR 默认为 100%，对于经常会 update 的表如库存，可以降低一些，比如 70%-90%
- pg 的 delete+insert 策略，除了显而易见的多份存储缺点外，有几个考虑：mvcc update 本来就要用，in place 也避免不了保存副本，对于 update 了很多次的表，只要及时 vacuum（vacuum 一般空间浪费不会超过 10%。根据 TPCC 持续超过 24 小时的测试）
- truncate 会立刻回收文件系统空间，比 oracle 更合适，所以可以使用
- 如果有几个 SQL 语句需要顺序执行，要最大化性能的话，可以使用存储过程或者匿名 PL/pgSQL 块
- 能使用 varchar 就不要使用 text 类型，性能大约差 5%
- OUTER JOIN 不仅可以用来修改语义，还能够充当优化器提示的作用（借用 LEFT JOIN），因为语义不可违反。
- LightDB 能够在差不多吞吐量的情况下比 redis 提供更低的响应时间，同时保证 ACID、高可用、自动故障转移、0 I/O 消耗、分片、大结果集缓存。
- 对于很简单的单 SQL 存储过程，用存储过程可能会降低响应时间，高冲突、万兆网络时忽略不计，TPS 也不一定能够上去甚至下降，CPU 大约会降低 1/3-2/3（benchmarksql

的测试结果）。

- 对于性能极端关注的应用（例如关心快 1 微秒还是慢 1 微秒），尽量不要使用 oracle、mysql 兼容函数和数据类型。
- 使用 lightdb pwr 和 lightdb em 分析 lightdb 数据库性能。
- 分布式数据库中，分区的用处是 pruning，分布式的用处是 parallel。
- 有了分布式数据库，是不是就不用分库分表了？可以参见，物理设计优化不可或缺（物理属性+多维分布的冗余、按照领域做 co-location，甚至某些数据双同步）。 https://www.zhihu.com/question/356734992

性能相关

这部分包含和 postgresql 性能优化相关的专题（也可以称为 recipe），它们中的一部分和上一节在内容上有所重复，但是本节讨论的是如何实现最大化性能或最小化锁，并不讨论特性本身。

空闲会话超时管理

idle_in_transaction_session_timeout

TCP 保活

tcp_keepalives_idle

SQL 语句超时

statement_timeout

大表添加字段

postgresql 不支持即时加字段，如果字段有默认值，会导致整个表被重写，耗时很长。这一点和 oracle 是一样的。需要先添加，然后分批 update 的方式防止被锁。

```
-- select, update, insert, and delete block until the catalog is update
(milliseconds)

ALTER TABLE items ADD COLUMN last_update timestamptz;
```

```
-- select and insert go through, some updates and deletes block while the table is rewritten

UPDATE items SET last_update = now();
```

全索引扫描优化

有时候为了优化性能， 我们希望访问路径走全索引扫描（Index Only Scan），此时会在创建索引时包含多个字段。如下：

```
SELECT SUM(eur_value)

  FROM sales

 WHERE subsidiary_id = ?
```

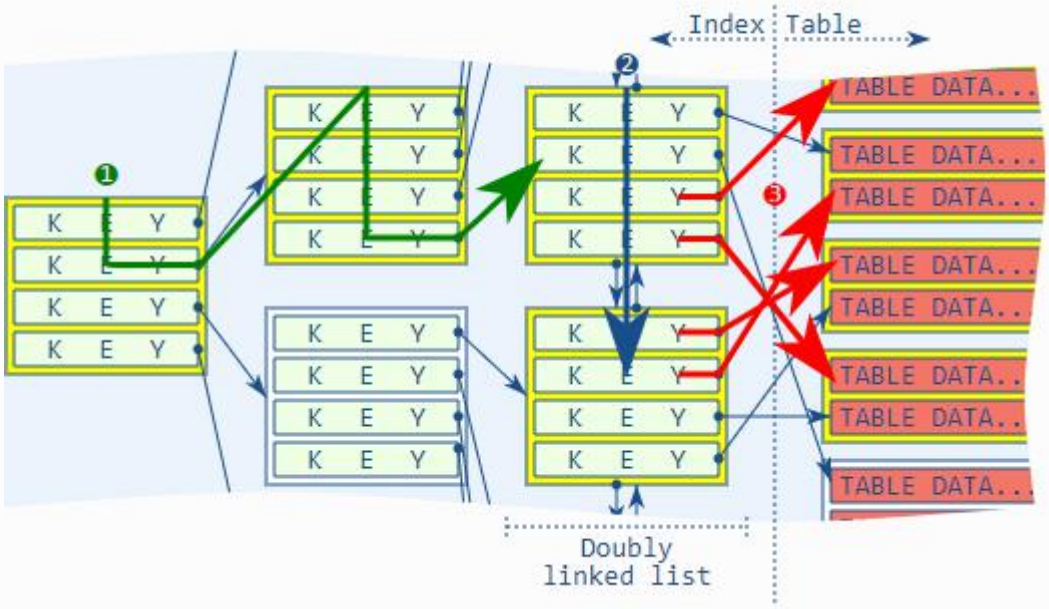
假设 sales 表包含 100 个字段，此时包含两个字段的索引将极大的提升性能。

```
CREATE INDEX idx

    ON sales

     ( subsidiary_id, eur_value );
```

它的缺点在于 eur_value 完全不会被使用到，但是索引的链表需要维护它。如下：



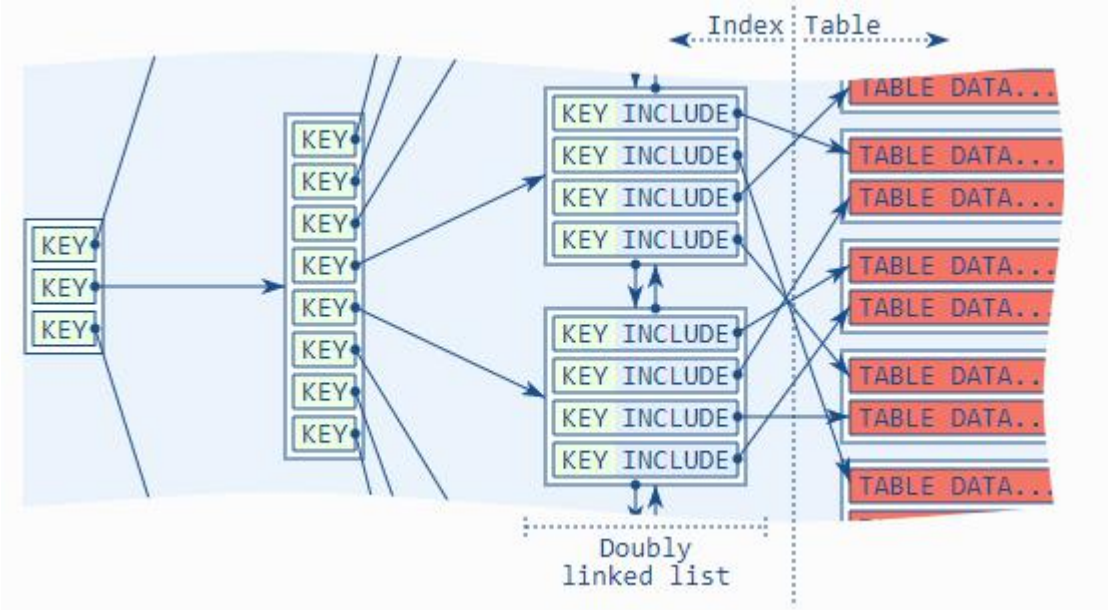
因此，lightdb 支持将用于过滤的字段和结果字段分开，通过 INCLUDE 子句，这样索引结构就不用额外维护 eur_value 字段。

```
CREATE INDEX idx

    ON sales ( subsidiary_id )

    INCLUDE ( eur_value );
```

这样存储结构就变成了下列：



性能相比纯粹的多列索引更佳，当前 oracle、mysql、mariadb 均不支持该特性。

在线创建索引

CREATE INDEX 会阻塞写，虽然不会阻塞读。因此，应该使用并发模式。

```
CREATE INDEX CONCURRENTLY items_value_idx ON items USING GIN (value
jsonb_path_ops);
```

主键使用 uuid 还是自增

在 lightdb 中，使用数字自增类型作为主键要比 uuid 从性能和存储上都更加高效。lightdb 集中式版本支持三种类型的自增方式，如下：

类别	允许显式插入	显式插入后会基于新max(id)做自动修改	truncate table后自动重置	是否跟随事务回滚一同回滚	多对象共享	是否支持手动重置
serial	Y	N	N	N	N	Y: SELECT setval((SELECT pg_get_serial_sequence('myschema.test_serial', 'id')), 1, false);
sequence	Y	N	N	N	Y	"Y: select setval('myschema.seq_1',(select max(id) from myschema.test_seq)::BIGINT); ALTER SEQUENCE test_old_id_seq RESTART WITH 1000;"
identity	Y	N	N	N	N	Y: ALTER TABLE myschema.test_identiy_1 ALTER COLUMN id RESTART WITH 100; TRUNCATE table *** RESTART IDENTITY;

使用自增时，分布式数据库下如何保证全局唯一性及并发性。

在分布式版本中，支持 serial 和序列，不支持 identity。

表连接方式

lightdb 支持哈希连接、排序-合并连接以及嵌套循环三种连接，可以使用优化器提示或 GUC 参数针对语句或会话进行启用或禁用某些连接方式。

优化器提示

和 oracle，mysql 一样，lightdb 也支持优化器提示，用法可以和 oracle 优化器提示语法一样，如下：

```
postgres=# create index idx_big_table on big_table(id);
CREATE INDEX
postgres=# explain select count(1) from big_table;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=24.71..24.72 rows=1 width=8)
   ->  Bitmap Heap Scan on big_table  (cost=2.31..24.63 rows=32 width=0)
         ->  Bitmap Index Scan on idx_big_table  (cost=0.00..2.30 rows=32 width=0)
(3 rows)

postgres=# explain select /*+ SeqScan(t) */ count(1) from big_table t;
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate  (cost=32.40..32.41 rows=1 width=8)
   ->  Seq Scan on big_table t  (cost=0.00..32.32 rows=32 width=0)
(2 rows)
```

详细的优化器提示列表可参见 https://www.hs.net/r/cms/www/itn/forPrd/html/pghint_plan.html#id-1.11.7.40.11。

除了优化器提示外，https://www.postgresql.org/docs/13/explicit-joins.html

SQL 计划管理

LightDB enable_hint_table 可以看成类似 oracle outline 工具，可以在不修改 SQL 的情况下，通过 hint 改变 SQL 的执行计划。

https://www.cnblogs.com/kingbase/p/15321355.html

只读表

lightdb 支持只读表，该特性 PG 官方曾提议，但未被采纳。

```
postgres=# vacuum full big_table ;
VACUUM
postgres=# alter table big_table set read only;
NOTICE:  you must vacuum the table before set it read only
ALTER TABLE
postgres=# create index idx_big_table1 on big_table(id);
CREATE INDEX
postgres=# insert into big_table values (1,lpad('123', 8096, 'abc'));
ERROR:  table "big_table" is read only
postgres=# alter table big_table add new_col text;
ERROR:  table "big_table" is read only
```

只读表不仅可以用来防止误修改，还能提高查询的性能，避免不必要的自动 vacuum。

分区

分区剪除关联

列表很长的 IN 优化

递归 CTE 除了可以用来实现层次查询外，也是优化大集合、高选择性 DISTINCT 和长列表 IN 查询的技术。
https://www.postgresql.eu/events/nordicpgday2018/sessions/session/1858/slides/68/query-optimization-techniques_talk.pdf

并行执行

PG 并行执行有一些限制：

- insert select 不支持并行执行
- union all 不支持并行执行
- rownum 不支持并行执行
- 序列不支持并行执行

强制并行执行
可以使用优化器提示强制走并行执行。

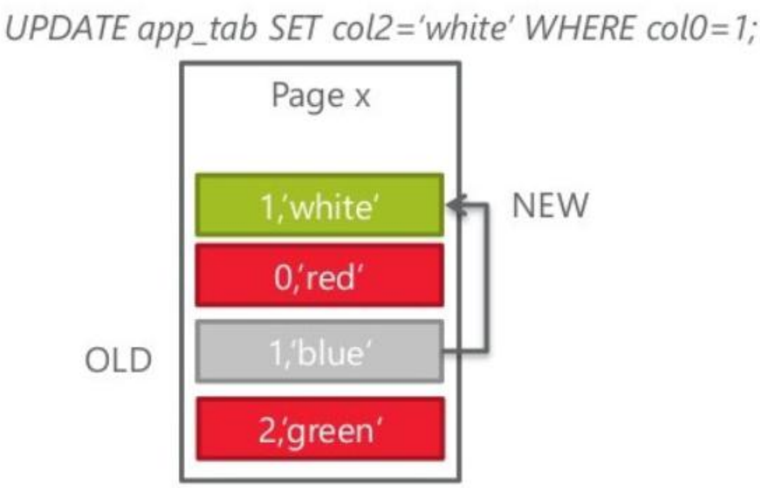
```
postgres=# explain (analyze,verbose) select /*+ Parallel(a 3 hard) Parallel(b 3 hard) Ordered(b a) */count(1) from big_table a,big_table b where a.id=b.id;
                                                   QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=0.92..0.93 rows=1 width=8) (actual time=0.576..0.631 rows=1 loops=1)
   Output: count(1)
   ->  Hash Join  (cost=0.40..0.84 rows=32 width=0) (actual time=0.348..0.551 rows=1024 loops=1)
         Hash Cond: (b.id = a.id)
         ->  Gather  (cost=0.00..0.00 rows=32 width=4) (actual time=0.101..0.158 rows=32 loops=1)
               Output: b.id
               Workers Planned: 3
               Workers Launched: 0
               ->  Parallel Seq Scan on public.big_table b  (cost=0.00..0.00 rows=10 width=4) (actual time=0.002..0.019 rows=32 loops=1)
                     Output: b.id
         ->  Hash  (cost=0.00..0.00 rows=32 width=4) (actual time=0.239..0.256 rows=32 loops=1)
               Output: a.id
               Buckets: 1024  Batches: 1  Memory Usage: 10kB
               ->  Gather  (cost=0.00..0.00 rows=32 width=4) (actual time=0.190..0.245 rows=32 loops=1)
                     Output: a.id
                     Workers Planned: 3
                     Workers Launched: 0
                     ->  Parallel Seq Scan on public.big_table a  (cost=0.00..0.00 rows=10 width=4) (actual time=0.005..0.042 rows=32 loops=1)
                           Output: a.id
 Planning Time: 0.393 ms
 Execution Time: 0.674 ms
(21 rows)
```

GIN 索引（即席查询性能神器）

https://www.cnblogs.com/zhjh256/p/15357837.html

redo 与 undo

PostgreSQL 没有 undo，通过下列方式实现 MVCC：
DML 插入新记录
将原始记录标记为 OLD
旧记录增加指向新记录的指针



vacuum 进程负责回收旧元组占用的空间（以及 vacuum full 重新组织），更新可见性映射，重置块的事务 ID 防止回卷。

需要注意的是，索引是不维护多版本的，多版本的信息由元祖负责维护并保证 MVCC。因此一个缺点是，explain 中看到的 Index only scan 并不一定真的是 index only scan，还是有可能需要回表检查，运行时由 VM 中的位确定。如果表是只读表，那么通过执行计划，我们就能确定 index only scan 一定是 index only scan。

vacuum

默认情况下，自动 vacuum 在被调度时，如果表被修改或删除的数据量超过 50（`autovacuum_vacuum_threshold`）及 20% 数据被修改或删除（`autovacuum_vacuum_scale_factor`），新增的记录超过（`autovacuum_vacuum_insert_threshold`）1000 及 20% 数据被新增（`autovacuum_vacuum_insert_scale_factor`），会进行 vacuum 操作。这意味着如果表很大，但是每次修改的数据量很少，比如客户表、资金表、以及一些历史流水表。那么 vacuum 就很长时间不会被运行，从而容易发生 wraparound vacuum。
vacuum 在运行的时候，是根据块进行扫描的，会先扫描关系的 VM（其中为每页保存 2 个 bit，第二位标记是否已冻结），确定某个块是否已经被冻结，如果已经被冻结，则表示不需要执行 vacuum 操作。
可能发生 wraparound 的一个可能场景是批处理中，for 循环逐条处理和提交，没有在事务中。这样事务 ID 就会使用的极快，从而导致发生的概率大大增加，所以 POC 大容量压测，使用 LightDB-EM 监控就很重要。

文本数据导入导出

导出推荐使用 copy to 或 ltsql \COPY 命令
导入推荐使用 lt_bulkload，参见 https://www.cnblogs.com/zhjh256/p/15522157.html
http://www.postgis.us/presentations/PGOpen2018_data_loading.pdf

java 大数据处理

对于 java 大数据导入导出，从通用的角度考虑，优先推荐使用 JDBC 批处理特性。具体可参见 https://www.cnblogs.com/zhjh256/p/10008474.html。
https://stackoverflow.com/questions/46988855/correct-way-to-use-copy-postgres-jdbc
https://jdbc.postgresql.org/documentation/publicapi/org/postgresql/copy/CopyManager.html
https://github.com/randomtask1155/JDBCBulkInsert

缓存预热

lightdb 不仅支持和 oracle、mysql 一样的缓存预热机制，在此基础上，还提供自助按需式的缓存预热功能，包括共享缓存、页面缓存，以及表级预热。
https://www.cnblogs.com/zhjh256/p/15513698.html

锁表分析

锁的兼容性矩阵如下：

	Existing Lock Mode							
Requested Lock Mode	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL.					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCL.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACCESS EXCL.	X	X	X	X	X	X	X	X

`pgrowlocks` 可以用来查看某个表的行锁信息。
https://www.cnblogs.com/zhjh256/p/15235863.html

`pg_blocking_pids(pid)` 函数可以查询阻塞某个 `PID` 的 `PID` 列表。如下：

AWR 与 ASH

https://www.hs.net/r/cms/www/itn/forPrd/LightDB-21.2-html/pgprofile.html
https://www.hs.net/r/cms/www/itn/forPrd/LightDB-21.2-html/pgprofile.html#id-1.11.7.42.11

oracle 中对应的 v$session

pg_stat_activity
ltcenter

explain

explain 可以用来查看 SQL 语句的解释计划，它并不总是和实际的执行计划相同。因此，explain 提供了很多子句可以用来查看实际的执行计划以及资源消耗（相当于 oracle 的 set autotrace traceonly），它会先执行 SQL 语句，然后显示执行计划以及相关的资源消耗统计。如下所示：

```
postgres=# explain (analyze,verbose,buffers,settings) select count(1) from big_table a,big_table b where a.id=b.id;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=61.68..61.69 rows=1 width=8) (actual time=0.896..0.897 rows=1 loops=1)
   Output: count(1)
   Buffers: shared hit=1088
   ->  Nested Loop  (cost=0.14..61.60 rows=32 width=0) (actual time=0.014..0.791 rows=1024 loops=1)
         Buffers: shared hit=1088
         ->  Seq Scan on public.big_table a  (cost=0.00..32.32 rows=32 width=4) (actual time=0.005..0.021 rows=32 loops=1)
               Output: a.id, a.name1
               Buffers: shared hit=32
         ->  Index Only Scan using idx_big_table1 on public.big_table b  (cost=0.14..0.91 rows=1 width=4) (actual time=0.001..0.019 rows=32 loops=32)
               Output: b.id
               Index Cond: (b.id = a.id)
               Heap Fetches: 1024
               Buffers: shared hit=1056
 Settings: effective_cache_size = '11GB', enable_partitionwise_aggregate = 'on', min_parallel_table_scan_size = '2GB', parallel_setup_cost = '10000',
4MB', work_mem = '128MB'
 Planning:
   Buffers: shared hit=16
 Planning Time: 0.236 ms
 Execution Time: 0.932 ms
(18 rows)
```

为了便于自定义处理，EXPLAIN 也支持 JSON 格式返回。如下：

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                   +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
```

https://www.postgresql.org/docs/current/sql-explain.html 可以查看详细选项。

查看正在运行 SQL 的执行计划

当前 lightdb 不支持查看正在执行的 SQL 的执行计划，慢 SQL 也只是执行完成后才会体现。

进度报告

lightdb 可以查看一些执行可能较慢的操作的进度信息，类似于 oracle 的 pg_stat_progress_xxx 视图。当前报告进度的操作包括：

- pg_stat_progress_analyze
- pg_stat_progress_create_index
- pg_stat_progress_vacuum

- pg_stat_progress_cluster
- pg_stat_progress_basebackup
- pg_stat_progress_copy

因为目前不支持查看 SQL 语句的进度，该特性目前帮助不是特别大。

参见：https://www.postgresql.org/docs/current/progress-reporting.html

杀掉会话

```
select pg_terminate_backend(pid);  --底层通过发送 SIGTERM 信号
```

取消正在执行的 SQL

```
SELECT pg_cancel_backend(pid);  --底层通过给 pid 发送 SIGINT 信号实现
```

查看正在执行的 SQL

select * from pg_stat_activity;

性能视图

不同于 oracle 和 mysql，lightdb 没有专门存储性能视图的 schema。主要是 pg_stat_ 和 pg_statio_开头。

监控管理

LightDB 内置数据库管理平台 LightDB EM，类似于 Oracle Enterprise Manager。LightDB 21.3 开始，将同时提供云管控台，单个云管控台可管理上百个 LightDB、PostgreSQL 以及其它数据库的实例。

同时支持 x86 和 arm 平台下 Redhat、CentOS、Kylin 操作系统部署。

数据同步

LightDB 支持逻辑复制、物理流复制、notify、轮训拉取模式这三种机制将数据同步给下游。

迁移

数据迁移

lightdb 包含从 mysql/mariadb、oracle 到 lightdb 的表结构和数据迁移工具，用户可以进行自助式迁移。

代码转换

基准性能测试工具

近年来，出现了很多各种数据库的基准测试，每个厂商都会说自己比其它性能要好。所以“哪个更快”这个问题不会有一个最终的答案，它在很大程度上取决于具体的使用场景、查询、用户和连接数量等因素。

不过，如果你确实想知道，可以使用 LightDB 推荐的三方测试套件进行测试。

HammerDB 可以用于测试 TPC-C 和 TPC-H，主要采用存储实现，不包含外键，支持 Oracle、MySQL、LightDB、Greenplum、SQL Server、Greenplum 等。

BenchmarkSQL 可以用于测试 TPC-H，主要采用 JDBC 直接写 SQL 实现，包含外键，支持 Oracle、MySQL、OceanBase、LightDB、达梦、openGauss 等。

简单的增删改查和类 K/V 可以使用 YCSB。

所以哪个性能更好取决于你想测试哪种场景下的表现，仅仅因为 A 比 B 高 20%而不考虑其他因素是有失偏颇的，各 benchmarksql 工具的对比可详见 https://www.cnblogs.com/zhjh256/p/14792894.html。

示例数据库与用户

LightDB 后续将内置示例数据库和用户 ltfin, 用于演示金融业务下相关的 LightDB 实现。

数据库选型

系统类型	数据特点	适合的数据库类型	推荐数据库
交易类系统	结构化数据	OLTP 数据库	RDBMS 或 NewSQL
支撑类系统			
渠道类系统			
管理类系统			
联机分析类系统	结构化数据	OLTP 数据库	RDBMS 或 NewSQL
	半结构化数据	OLAP 数据库	NewSQL 或 NoSQL
	非结构化数据	OLAP 数据库	NewSQL 或 NoSQL
离线分析类系统	半结构化数据	Hadoop 或 OLAP 数据库	Hadoop 或 NewSQL、NoSQL
	非结构化数据		

系统类型	系统特点	所处理主要数据的特点	所属交易类型
交易类系统	为客户提供银行业务，面对的是银行客户，如核心系统、信贷、信用卡、支付类系统、理财业务、资金业务、特色业务等	结构化数据	OLTP
支撑类系统	该类系统用于支撑银行其他系统的运行，如 ESB、交换、调度、外联等		
渠道类系统	该类系统为客户提供使用渠道，如柜面、ATM、网银、手机银行等系统，渠道类系统有传统渠道和互联网渠道		
管理类系统	为银行内部用户提供管理功能，面对的是银行内部员工，如 OA、绩效、人力、财务管理等		
联机分析类系统	该类系统为银行的业务、决策、合规等提供数据支持，如风险管理、在线决策支持、反洗钱、关联交易分析等	结构化数据	OLTP
		半结构化数据	OLAP
离线分析类系统	用于离线分析，如大数据平台、基础数据平台等	半结构化数据	离线分析
		非结构化数据	

数据类型	数据特点	适合的数据库类型	推荐数据库
结构化数据	由二维表结构来逻辑表达和实现的数据，严格地遵循数据格式与长度规范，数据处理时效性要求较高。	OLTP 数据库	RDBMS 或 NewSQL
半结构化数据	是结构化的数据，但是结构变化很大。通常需要了解数据细节	OLAP 数据库	NewSQL 或 NoSQL
非结构化数据	不宜采用关系型数据库来处理的数据，创建和管理数据项时，通常采用多值字段和变长字段，广泛应用于全文检索和各种多媒体信息处理领域。但数据处理时效性要求相对较低。	OLAP 数据库或 Hadoop	NewSQL、NoSQL 或 Hadoop