KINGBASE™
人大金仓

KingbaseES V008R003

性能优化指南

北京人大金仓信息技术股份有限公司

约定

以下文本约定适用于本文档：

- 中括号（[和]）表示包含一个或多个可选项。不需要输入中括号本身。
- 花括号（{和}）表示包含两个以上（含两个）的候选，必须在其中选取一个。不需要输入花括号本身。
- | 为分割中括号或者花括号中的两个或两个以上选项。不需要输入“|”本身。
- 点（. . .）表示其之前的元素可以被重复。

性能优化指南

欢迎来到KingbaseES V008R003（以下简称KingbaseES V8R3）教程。本教程从安装与更新、SQL和PL/SQL、管理、开发、性能、安全和高可用七部分对KingbaseES V8R3数据库进行全方位的介绍。

本部分主要为KingbaseES V8R3**性能优化指南**，是对数据库性能的监控和调优。

目录

Part I. 快速入门

目录

Part II. 性能

目录

数据快速加载

sys_bulkload是KingbaseES提供的快速加载数据的命令行工具。用户使用sys_bulkload工具能够把一定格式的文本数据简单、快速的加载到KingbaseES数据库中，或将KingbaseES数据库中的数据快速导出到CSV文件中。

1. sys_bulkload

语法格式：

sys_bulkload [options][control_file_path]

功能：

依照配置文件中指定的信息快速加载数据到KingbaseES数据库中，或者将KingbaseES数据库中的数据快速导出到CSV文件中。

参数说明：

options参数

加载选项

-i *in_file*
--input *in_file*

指定待加载文件或需要导出数据的表，同配置文件的INPUT选项。

-O *output_table*
--output *output_table*

快速加载的目标表或者接收导出数据的文件路径，同配置文件的OUTPUT选项。

-l *log_file*
--logfile *log_file*

结果日志路径，同配置文件中的LOGFILE选项。

-P *bad_file*
--parse-badfile *bad_file*

记录无法正确解析或写失败的日志路径，同配置文件的PARSE_BADFILE选项。

-u *duplicate_badfile*
--duplicate-badfile *duplicate_badfile*

重建索引时与唯一性约束冲突而导入失败的记录的写入路径，同配置文件中的DUPLICATE_BADFILE选项。

-o *optional*
--option *optional*

任何在配置文件中任何可使用的选项，参见第 2 节。可以传入多次选项。对于加载的必填选项如input, output等，为指定参数的方便和简洁已在上面提供了单独长短项配置。

连接选项

-d *dbname*
--dbname *dbname*

指定连接的数据库。如未指定，数据库名将从环境变量KINGBASE_DATABASE中读取。如未设置该环境变量，使用连接的用户名。

-h *host*
--host *host*

指定运行服务器的主机名。如果值以/开始，被用作unix域套接字的目录。

-p *port*
--port *port*

指定TCP的端口号或者本地unix域套接字文件扩展名的服务器监听端口号。

-U *username*
--username *username*

指定连接的用户名。

-W *password*
--password *password*

连接用户的数据库登录密码。

一般选项

-e
--echo

输出发送给服务器的命令。

-E
--elevel

设置输出信息的级别，其中级别包括：DEBUG, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, and PANIC。默认为INFO。

--help

输出帮助信息

--version

输出版本号

control_file_path参数

指定配置文件的路径。配置文件写法参见第 2 节

> **注意：** 部分字符串参数，如 IP，文件路径等参数。在 Windows 系统下需要使用 " 传递，如：... -h "127.0.0.1" -i "C:\KingbaseES\ ... "。在 Linux 系统下则不需要，如：... -h 127.0.0.1 -i /home/KingbaseES/ ...。

2. 配置文件

功能：

包含了加载数据时所需要的一些配置。导入时只需要将配置文件的路径 + 文件名作为参数传入快速加载提供的系统函数或可执行程序即可。

配置文件中参数选项说明：

必填参数

```
TYPE = CSV | TEXT | BINARY | DB
```

加载的数据源的类型

CSV

从CSV格式的文本文件加载数据

TEXT

从TEXT格式的文本文件加载数据

BINARY

从二进制格式的文件加载数据

DB

从数据库的表中导出数据

```
INPUT = PATH | [schem_aname.] table_name
```

需要导入的数据文件路径或者导出数据的源表

PATH

需要导入的数据源文件路径。如果是相对路径，在控制文件中指定时，它将与控制文件相对；当在命令行参数中指定时，相对于当前工作目录。KingbaseES服务器的用户必须具有该文件的读取权限。在“TYPE = CSV | TEXT | BINARY”时可用。

[schem_aname.] table_name

需要导出数据的表名。仅在“TYPE = DB”时可用。

OUTPUT = [schem_aname.] table_name | PATH

指定将数据导入的目标表或者目标文件。

[schem_aname.] table_name

导入数据的表名。在“ TYPE = CSV | TEXT | BINARY ”时可用。

PATH

导出数据的文件路径。如果是相对路径，在控制文件中指定时，它将与控制文件相对；当在命令行参数中指定时，相对于当前工作目录。KingbaseES服务器的用户必须具有该文件的读取权限。仅在“TYPE = DB”时可用。

LOGFILE = PATH

指定一个文件记录日志。如果指定为相对路径，则指定规则与INPUT相同。默认值为$KINGBASE_DATA/sys_bulkload/<timestamp>_<dbname>_<schema>_<table>.log

非必填参数

WRITER = DIRECT | BUFFERED | CSV_FILE

指定数据的加载方式，默认值为 BUFFERED。

DIRECT

直接将数据加载到指定的数据表中，绕过共享缓冲区并跳过WAL日志记录，但一旦出错需要自己的恢复过程。DIRECT方式导入时，利用日志文件进行的数据操作将无法成功，比如逻辑同步。

BUFFERED

通过共享缓冲区将数据加载到表中。使用共享缓冲区编写WAL日志，并可使用KingbaseES的WAL日志进行恢复。

CSV_FILE

当数据的加载方式指定为CSV_FILE时，表示将数据库中的数据导出到CSV格式的文本文件中，该参数一般与“TYPE = DB”配合使用。

LIMIT = n

LIMIT只在导入数据文件时有效，加载n行即停止加载。默认值为最大的64位整数（即(2^64)/2-1 = 9223372036854775807），表示不限制加载行数。

ENCODING = encoding

指定输入数据的编码格式，检查指定的编码格式是否合法。默认不检查。若有需要转化输入文件的格式为数据库的编码格式。如果可以确保输入文件格式与数据库格式一致，不指定该选项，会有助于加载速度的提高，因为会忽略字符集的检查和转化。配置文件中ENCODING选项与数据库编码选项的转化关系详见表 2-1。

表 2-1. 配置文件中ENCODING选项与数据库编码选项的转化关系

配置文件中Encoding选项指定	SQL ASCII	non-SQL ASCII
未指定	既不检查也不转化	既不检查也不转化
SQL_ASCII	既不检查也不转化	只检查
非SQL_ASCII，与数据库一致	只检查	只检查
非SQL_ASCII，与数据库不一致	只检查	检查并转化

CHECK_CONSTRAINTS = YES | NO

指定加载时是否进行约束检查，默认为NO。

PARSE_ERRORS = n

允许出现的错误次数。若错误次数超过该设置值，则快速加载退出运行。-1表示不限制错误个数，0为默认值，表示不允许错误，其他表示允许的错误次数。

FILTER = [schema_name.] function_name[(arg_value, ...)]

只在导入数据文件时有效，指定过滤函数用来转换输入文件的每行，如果函数名在数据库中唯一，可以忽略函数的参数类型定义。如果该选项未指定，输入数据将直接被解析到目标表中。

DUPLICATE_ERRORS = n

允许违反唯一约束的忽略的元组个数。冲突的元组将从表中删除并被记录在重复失败的文件中。如果大于等于重复记录数，记录会被回滚。默认值为0，表示不允许重复记录数，-1表示忽略所有错误。

ON_DUPLICATE_KEEP = NEW | OLD

执行元组如何处理违反唯一约束。被删除的元组会被记录在出错文件中。设置了该选项，同样需要设置DUPLOCATE_ERRORS大于0。默认值为NEW。

NEW

采用输入文件中的最新一条记录的数据替换表中原有的数据。

OLD

保持表中原有数据，删除输入文件中的元组。

PARSE_BADFILE = PATH

指定一个文件路径（若指定路径的文件不存在则自动创建一个新文件），默认值为与导入的数据文件同目录，文件名为导入数据文件的文件名+".badf"后缀的文件。保存数据文件中解析失败的数据行。若在数据文件解析的过程中解析失败，则该数据行追加记录到该文件。

DUPLICATE_BADFILE = PATH

指定一个文件路径（若指定路径的文件不存在则自动创建一个新文件），默认值与导入的数据文件同目录，文件名为导入数据的文件名+".disc"后缀的文件。若存在不能被写入到数据库的元组，则该元组对应的数据文件中的行追加记录到该文件。例如在数据导入过程中元组违背了约束（唯一，主键，非空，check）原则，则该元组不能写入+".badf"后缀的文件路径。若在数据文件解析的过程中解析失败，则该数据行追加记录到该文件。

TRUNCATE = YES | NO

是否删除所有目标表中的数据，默认值为NO。多进程并行和TYPE为DB时不支持该选项。

VERBOSE = YES | NO

出错的元组是否写入到服务器日志中，默认值为NO。

```
DELIMITER = delimiter_character
```

间隔符，数据文件中列与列的间隔符，为单个字符，可以为任何可视化字符。默认值为逗号(,)。当需要一个TAB字符作为间隔符时，用双引号包裹TAB字符，如DELIMITER = "" 。

```
QUOTE = quote_character
```

QUOTE在文件格式为CSV时有效，详情参考COPY语句。默认值为双引号(")。

```
ESCAPE = escape_character
```

ESCAPE在文件格式为CSV时有效，详情参考COPY语句。默认值为反斜杠(\ )。

```
REINDEX = YES | NO
```

导入数据后是否重建索引。默认值为NO。

```
SKIP_LAST_EMPTY_VALUE = YES | NO
```

只对CSV格式有用，最后一列数据为空，是否把它当成一列数据，还是只是当分隔符。默认值为NO。用TPCH测试时生成的CSV文件在行末尾会加一个分隔符，指定该选项为YES，忽略最后一个分隔符。

```
SKIP = n
```

SKIP只在导入TEXT和CSV格式的数据文件时有效，TEXT和CSV格式的数据文件以行为单位进行导入，该选项可以设置跳过多少行数据，这些数据不导入数据库。默认值为0。

```
NULL = null_string
```

指定表示一个空值的字符串。默认值是一个没有引号的空字符串。

```
FORCE_NOT_NULL = column_name
```

该选项强制要求指定的列不为NULL值，默认情况下将空字符串按照NULL值处理，如果指定了该值，则空字符串不再按照NULL值处理，而是按照零长字符串处理。该选项不能与FILTER一起使用。

```
TRACKING_INTERVAL = n
```

指定导入过程中的时间间隔（单位为：秒），客户端反馈导入状况，防止在导入过程中因异常卡死，但用户不知道。默认为0，表示不反馈导入信息，其它非0整数为反馈时间间隔。

```
PROCESSOR_COUNT = n
```

指定服务器并行处理的进程数，具体参数值可根据用户服务器的CPU个数指定。默认值为1。TYPE为BIANRY方式不支持该参数。

值为1。TYPE为BLANKY方式不支持该参数。

```
ASYNC_WRITE = YES | NO
```

指定服务器的写文件是否独立的进程，默认值为NO。当WRITE为DIRECT或者CSV_FILE时，如果PROCESSOR_COUNT大于1，则无论是否指定ASYNC_WRITE，其值始终为YES。

```
DUMP_PARAMS = YES | NO
```

是否将配置参数信息导入到日志文件中。默认值为YES。

其中配置文件中的选项不区分大小写，每个选项占用一行，选项和选项值之间通过等号进行连接，选项的值可以有引号也可以没有引号。如果字符串中有空格、等号等特殊值，则必须加引号。可以通过在正文内容前加入“#”表示注释该行后面的配置。具体格式如下：

```
PROCESSOR_COUNT =4
DELIMITER = ","                          # Delimiter
QUOTE = "\""                             # Quoting character
LOGFILE =/home/bulkload.log
DUPLICATE_BADFIL=/home/bad.dat
TYPE=CSV
INPUT ==/home/data.csv
OUTPUT=test_table
```

3. 支持的数据文件格式

KingbaseES支持CSV、TEXT和BINARY三种数据文件格式。数据文件可以通过Copy To语句生成，三种文件格式详见COPY中文件格式部分。

4. 支持的写入方式

KingbaseES提供了两种方式来写入数据库文件：BUFFERED方式和DIRECT方式。如果导入的目标表已建立索引，要求加载完数据后索引仍然生效。

4.1. BUFFERED方式

BUFFERED同普通Insert语句相比，优势在于节省了语句解析过程中带来的时间损耗。BUFFERED方式，（与INSERT一样）在目标表上添加Row Exclusive锁，因此所有与Row Execlusive锁互斥的操作均会被阻塞。具体如下表（未罗列出来的SQL语句均可顺利执行）：

表 4-1 BUFFERED方式下的阻塞语句

不会被阻塞的SQL	被阻塞的SQL	被间接阻塞的SQL（由于删除其他对象而导致表被删除或表定义被修改）
DML：SELECT、UPDATE、DELETE、MERGE INTO、COPY FROM、COPY TO、VACUUM、ANALYZE、GRANT、REVOKE、Create Index、COMMENT ON TABLE、DBCC CHECKCATALOG、CREATE TABLE…INHERITS、CREATE VIEW、ALTER INDEX RENAME TO、ALTER	ALTER TABLE、DROP TABLE、TRUNCATE TABLE、VACUUM FULL、apply_table_policy()、remove_table_policy()、Create Index、DBCC CHECKDB、DBCC CHECKALLOCATE、DBCC CHECKTABLESPACE、CREATE TRIGGER、REINDEX、ALTER TRIGGER	DROP SCHEMA（表所在的模式）、DROP OWNED（表对应的用户）、REASSIGN OWNED（表对应的用户）
INDEX SET（storage_parameter）、ALTER INDEX..UNUSABLE、ALTER SCHEMA、ALTER TABLESPACE、CREATE VIEW、ALTER INDEX、ALTER SCHEMA、CREATE ROLE、ALTER ROLE、DROP ROLE、ALTER SEQUENCE、CHECKPOINT、INSERT、	CREATE RULE、ALTER INDEX SET TABLESPACE、DROP INDEX、CREATE INDEX、CLUSTER、LOCK	

4.2. DIRECT方式

DIRECT方式不但节省了语句解析带来的时间损失，同时节省了BUFFERED方式在插入数据过程中进行各种检查所花费的时间。

DIRECT方式下，目标表会被加Access Exclusive锁，因此，所有对这个表操作都会被阻塞（与BUFFERED方式相比，用户可以根据目标表上的事务并发情况，使用不同的写入方式）。

5. 支持的导出方式

KingbaseES支持将数据库的数据导出到一个CSV文件格式的文件中。其功能与COPY TO导出数据库数据到一个 CSV 格式文件相同。

6. 性能调优

使用sys_bulkload时，如对性能需求较高，可通过修改服务器配置（即修改kingbase.conf），来改变服务器的资源调度，以提高 sys_bulkload性能。但此类配置参数的修改会对服务器的相关资源调度产生影响，因此建议用户使用完成后还原成修改之前的配置。除了服务器参数，部分sys_bulkload的配置参数也会影响到传输速率。

6.1. 服务器参数配置

下表给出了会对sys_bulkload性能产生影响的服务器参数，以及它的默认值和建议修改值。在实际应用过程中，可以依据服务器的硬件环境适当调整各参数的值。

表 6-1. 修改服务器配置参数

服务器配置参数	默认值	建议值
shared_buffers	128M	2GB
work_mem	4M	100MB
maintenance_work_mem	64M	1GB
autovacuum	on	off
fsync	on	off
full_page_writes	on	off
checkpoint_timeout	5min	1d
wal_buffers	-1	1GB
max_wal_size	1GB	150GB
track_activities	on	off

6.2. sys_bulkload 参数配置

LIMIT

在导入CSV文件并且已知文件行数的情况下，设置LIMIT参数为导入文件的行数可提升导入速度。

REINDEX

若导入数据不需要建立索引，则将REINDEX参数设置为NO(默认值)，可提升导入速度

PROCESSOR_COUNT

无论是通过DIRECT方式或者BUFFERED方式导入，都可以根据当前物理环境合理配置PROCESSOR_COUNT参数优化导入速度。

7. 实例

创建导入数据表

```
create table test(id int primary key, info text, crt_time timestamp);
```

数据文件示例：将下列数据以 test.csv 为文件名保存到KingbaseES服务器所在目录

```
1,29b35ff06c949e7e442c929e1df86396,2017-10-08 10:52:47.746062
2,06fde814525395de5ab85f6d92b04e87,2017-10-08 10:52:47.746573
3,c93f02e8677c9cd7c906c6ad5dbd450e,2017-10-08 10:52:47.746627
4,6541700070ae3d051f965fcef43baf45,2017-10-08 10:52:47.746835
5,3d7e7246016acaa842526b6614d0edf5,2017-10-08 10:52:47.746869
6,1d1ae5a03ef0bad3bc14cd5449ba0985,2017-10-08 10:52:47.746894
7,7745c57c54b97656bec80a502ec13ec7,2017-10-08 10:52:47.746918
8,3c377131f6ef82c3284dc77a3b4ffdf7,2017-10-08 10:52:47.746942
9,5ef98d40aeeadf65eb1f0d7fd86ed585,2017-10-08 10:52:47.746968
10,312c0a0188da9e34fe45aa19d0d07427,2017-10-08 10:52:47.746993
```

7.1. 导入数据

7.1.1. 以 DIRECT 方式导入 CSV 文件

不使用配置文件导入命令示例

```
./sys_bulkload -i /home/kingbase/test.csv -O test -l /home/kingbase/
test.log -o "TYPE=CSV" -o "WRITER=DIRECT" -h localhost -d TEST -U
SYSTEM -W 123
```

导入结果查询

导入成功提示信息：

```
    NOTICE: BULK LOAD START
    NOTICE: BULK LOAD END
        0 Rows skipped.
        10 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.
```

导入结果查询：

```
    TEST=# select * from test;
     id |               info               |          crt_time
    ----+----------------------------------+----------------------------
      1 | 29b35ff06c949e7e442c929e1df86396 | 2017-10-08 10:52:47.746062
      2 | 06fde814525395de5ab85f6d92b04e87 | 2017-10-08 10:52:47.746573
      3 | c93f02e8677c9cd7c906c6ad5dbd450e | 2017-10-08 10:52:47.746627
      4 | 6541700070ae3d051f965fcef43baf45 | 2017-10-08 10:52:47.746835
      5 | 3d7e7246016acaa842526b6614d0edf5 | 2017-10-08 10:52:47.746869
      6 | 1d1ae5a03ef0bad3bc14cd5449ba0985 | 2017-10-08 10:52:47.746894
      7 | 7745c57c54b97656bec80a502ec13ec7 | 2017-10-08 10:52:47.746918
      8 | 3c377131f6ef82c3284dc77a3b4ffdf7 | 2017-10-08 10:52:47.746942
      9 | 5ef98d40aeeadf65eb1f0d7fd86ed585 | 2017-10-08 10:52:47.746968
     10 | 312c0a0188da9e34fe45aa19d0d07427 | 2017-10-08 10:52:47.746993
    (10 行记录)
```

7.1.2. 以 BUFFERED 方式导入 TEXT 文件

配置文件示例（以test.ctl为名保存到服务器所在目录，也可自行指定其他目录。）

```
TABLE = test
INPUT = /home/kingbase/test.csv
TYPE = TEXT
SKIP = 2
LIMIT = 5
WRITER = BUFFERED
PROCESSOR_COUNT = 3
```

使用配置文件导入命令示例

```
./sys_bulkload -h localhost -d TEST /home/kingbase/test.ctl -U SYSTEM -W 123
```

使用配置文件导入结果

导入成功提示信息：

```
    NOTICE: BULK LOAD START
    NOTICE: BULK LOAD END
        2 Rows skipped.
        5 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.
```

导入结果查询（配置指定跳过前2行，限制导入条数5条）：

```
    TEST=# select * from test;
     id |               info               |          crt_time
    ----+----------------------------------+---------------------------
      3 | c93f02e8677c9cd7c906c6ad5dbd450e | 2017-10-08 10:52:47.746627
      4 | 6541700070ae3d051f965fcef43baf45 | 2017-10-08 10:52:47.746835
      5 | 3d7e7246016acaa842526b6614d0edf5 | 2017-10-08 10:52:47.746869
      6 | 1d1ae5a03ef0bad3bc14cd5449ba0985 | 2017-10-08 10:52:47.746894
      7 | 7745c57c54b97656bec80a502ec13ec7 | 2017-10-08 10:52:47.746918
    (5 行记录)
```

7.1.3. 导入二进制文件

利用 COPY TO 将数据库表中的数据导入到一个二进制文件中作为测试数据。

导入二进制文件数据命令示例

```
./sys_bulkload -i /home/kingbase/test.bin -O test -o "TYPE=BINARY"
-o "WRITER=DIRECT" -h localhost -d TEST -U SYSTEM -W 123
```

二进制文件导入结果

导入成功提示信息：

```
NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
    0 Rows skipped.
    10 Rows successfully loaded.
    0 Rows not loaded due to parse errors.
    0 Rows not loaded due to duplicate errors.
    0 Rows replaced with new rows.
```

导入结果查询：

```
TEST=# select * from test;
 ID |               INFO               |          CRT_TIME
----+----------------------------------+---------------------------
  1 | 29b35ff06c949e7e442c929e1df86396 | 2017-10-08 10:52:47.746062
  2 | 06fde814525395de5ab85f6d92b04e87 | 2017-10-08 10:52:47.746573
  3 | c93f02e8677c9cd7c906c6ad5dbd450e | 2017-10-08 10:52:47.746627
  4 | 6541700070ae3d051f965fcef43baf45 | 2017-10-08 10:52:47.746835
  5 | 3d7e7246016acaa842526b6614d0edf5 | 2017-10-08 10:52:47.746869
  6 | 1d1ae5a03ef0bad3bc14cd5449ba0985 | 2017-10-08 10:52:47.746894
  7 | 7745c57c54b97656bec80a502ec13ec7 | 2017-10-08 10:52:47.746918
  8 | 3c377131f6ef82c3284dc77a3b4ffdf7 | 2017-10-08 10:52:47.746942
  9 | 5ef98d40aeeadf65eb1f0d7fd86ed585 | 2017-10-08 10:52:47.746968
 10 | 312c0a0188da9e34fe45aa19d0d07427 | 2017-10-08 10:52:47.746993
(10 行记录)
```

7.2. 导出数据

将 text.csv 文件导入表 test 中作为测试数据。

导出数据命令示例

```
./sys_bulkload -d TEST -i TEST -O test_out.csv -o "TYPE=DB" -o
"WRITER=CSV_FILE" -o "DELIMITER=|" -h localhost -U SYSTEM -W 123
```

导出数据结果

导出成功提示信息：

```
    NOTICE: BULK LOAD START
    NOTICE: BULK LOAD END
        0 Rows skipped.
        10 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.
```

导出结果查询：

```
    cat test_out.csv
    1|29b35ff06c949e7e442c929e1df86396|2017-10-08 10:52:47.746062
    2|06fde814525395de5ab85f6d92b04e87|2017-10-08 10:52:47.746573
    3|c93f02e8677c9cd7c906c6ad5dbd450e|2017-10-08 10:52:47.746627
    4|6541700070ae3d051f965fcef43baf45|2017-10-08 10:52:47.746835
    5|3d7e7246016acaa842526b6614d0edf5|2017-10-08 10:52:47.746869
    6|1d1ae5a03ef0bad3bc14cd5449ba0985|2017-10-08 10:52:47.746894
    7|7745c57c54b97656bec80a502ec13ec7|2017-10-08 10:52:47.746918
    8|3c377131f6ef82c3284dc77a3b4ffdf7|2017-10-08 10:52:47.746942
    9|5ef98d40aeeadf65eb1f0d7fd86ed585|2017-10-08 10:52:47.746968
    10|312c0a0188da9e34fe45aa19d0d07427|2017-10-08 10:52:47.746993
```

性能提示

1. 使用EXPLAIN

在KingbaseES数据库中，每一个查询均会产生一个查询计划。选择正确的计划来匹配查询结构和数据的属性对于优化性能来说是重要的，数据库系统中包含了一个复杂的规划器来为查询选择更优计划。用户可以使用EXPLAIN命令查看规划器为任意查询生成的查询计划。

本节中的例子均从KingbaseES V8R3开发源代码的回归测试数据库中抽取而来，并在此之前做过一次VACUUM ANALYZE。用户在尝试本节中的例子时将会得到相似的结果，但估计代价和行计数可能会产生小幅变化，这是由于ANALYZE的统计信息是随机采样而不是精确值，并且代价也与平台有某种程度的相关性。

这些例子使用EXPLAIN的默认"text"输出格式，这种格式紧凑且便于阅读。如想把EXPLAIN的输出传给某程序做进一步分析，应当使用该程序可读的输出格式（XML、JSON 或 YAML）。

1.1. EXPLAIN基础

查询计划的结构是一个计划结点的树形结构。最底层的结点是扫描结点：将从表中返回未经处理的行。 不同的表访问模式有不同的扫描结点类型：顺序扫描、索引扫描、位图索引扫描。 也存在不是表的行来源，例如VALUES子句和FROM中返回集合的函数，它们有自己的结点类型。如果查询需要连接、聚集、排序、或者在未经处理的行上的其它操作，那么在扫描结点之上就会有其它额外的结点来执行这些操作。并且，这些操作通常有多种方法。因此这些位置也有可能出现不同的结点类型。EXPLAIN给计划树中每个结点输出一行，显示基本的结点类型和计划器是为该计划结点的执行所做的开销估计。第一行（最上层的结点）是对该计划的总执行开销的估计；计划器试图最小化的就是这个数字。

例1-1：显示输出：

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

由于该查询没有WHERE子句，必须扫描表中的所有行，因此计划器只能选择使用一个简单的顺序扫描计划。被包含在圆括号中的数字是（从左至右）：

- 估计的启动开销。在输出阶段可以开始之前消耗的时间，例如在一个排序结点里执行排序的时间。
- 估计的总开销。这个估计值基于的假设是计划结点会被运行完成，即所有可用的行都被检索。不过实际上一个结点的父结点可能很快停止读所有可用的行（见下面的LIMIT例子）。
- 这个计划结点输出行数的估计值。同样，也假定该结点能运行到完成。
- 预计这个计划结点输出的行平均宽度（以字节计算）。

开销是用规划器的*开销参数*所决定的捏造单位来衡量的。传统上以取磁盘页面为单位来度量开销；也就是seq_page_cost将被按照习惯设为1.0，其它开销参数将相对于它来设置。本节的例子都假定这些参数使用默认值。

一个上层结点的开销包括它的所有子结点的开销。并且，此开销只反映规划器关心的东西。特别是这个开销没有考虑结果行传递给客户端所花费的时间，这个时间可能是实际花费时间中的一个重要因素；但被规划器忽略了，因为无法通过修改计划来改变（每个正确的计划都将输出同样的行集）。

关于行数值的小技巧，由于它不是计划结点处理或扫描过的行数，而是该结点发出的行数。通常会比被扫描的行数少一些，因为有些被扫描的行会被应用于此结点上的任意WHERE子句条件过滤掉。理想中顶层的行估计会接近于查询实际返回、更新、删除的行数。

回到例1-1：

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的产生非常直接。若执行：

```
SELECT relpages, reltuples FROM sys_class WHERE relname = 'tenk1';
```

将发现tenk1有358个磁盘页面和10000行。开销被计算为（页面读取数*seq_page_cost）+（扫描的行数*cpu_tuple_cost）。默认情况下，seq_page_cost是1.0，cpu_tuple_cost是0.01， 因此估计的开销是 (358 * 1.0) + (10000 * 0.01) = 458。

例1-2：修改查询并增加一个WHERE条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..4800 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

请注意EXPLAIN输出显示WHERE子句被当做一个"过滤器"条件附加到顺序扫描计划结点。这意味

着该计划结点为它扫描的每一行检查该条件，并且只输出通过该条件的行。因为WHERE子句的存在，估计的输出行数降低了。但是，扫描仍将必须访问所有的 10000 行，因此开销没有被降低反而有所上升（准确来说，上升了 10000 * cpu_operator_cost）以反映检查WHERE条件所花费的额外 CPU 时间。

这条查询实际选择的行数是 7000，但是估计的行数只是个近似值。若尝试重复该试验，可能得到的估计值略有不同。此外，这个估计每次会在ANALYZE命令之后改变，因为ANALYZE生成的统计数据是从该表中随机采样计算的。

例 1-3：将条件变得更严格：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
       width=0)
         Index Cond: (unique1 < 100)
```

规划器决定使用一个两步的计划：子计划结点访问一个索引来找出匹配索引条件的行的位置，然后上层计划结点实际地从表中取出那些行。独立地抓取行比顺序地读取它们的开销高很多，但是这样不是所有的表页面都被访问，这么做实际上仍然比一次顺序扫描开销要少（使用两层计划的原因是因为上层规划结点把索引标识出来的行位置在读取之前按照物理位置排序，这样可以最小化单独抓取的开销。结点名称里面提到的"位图"是执行该排序的机制）。

例 1-4：给WHERE子句增加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
       width=0)
         Index Cond: (unique1 < 100)
```

新增的条件stringu1 = 'xxx'减少了估计的输出行计数，但是没有减少开销，因为我们仍然需要访问相同的行集合。请注意，stringu1子句不能被应用为一个索引条件，因为这个索引只是在unique1列上。它被用来过滤从索引中检索出的行。因此开销实际上略微增加了一些以反映这个额外的检查。

例 1-5：在某些情况下规划器将更倾向于一个"simple"索引扫描计划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1
 width=244)
   Index Cond: (unique1 = 42)
```

在这类计划中，表行被按照索引顺序取得，这使得读取它们开销更高，但是其中有一些是对行位置排序的额外开销。你很多时候将在只取得一个单一行的查询中看到这种计划类型。它也经常使用在拥有匹配索引顺序的ORDER BY子句的查询中，因为那样就不需要额外的排序步骤来满足ORDER BY。

例1-6：如果在WHERE引用的多个行上有独立的索引，规划器可能会选择使用这些索引的一个 AND 或 OR 组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
             rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78
             rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

但是这要求访问两个索引，所以与只使用一个索引并把其他条件作为过滤器相比，不一定更优。在该变动涉及到的范围，看到的计划将相应改变。

例1-7：LIMIT的效果：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000
LIMIT 2;

                                QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27
       rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

不是所有的行都被检索，增加一个LIMIT后，规划器将改变它的决定。注意索引扫描结点的总开销和行计数显示出好像它会被运行到完成。但是，限制结点在检索到这些行的五分之一后就会停止，因此它的总开销只是索引扫描结点的五分之一，并且这是查询的实际估计开销。之所以用这

个计划而不是在之前的计划上增加一个限制结点是因为限制无法避免在位图扫描上花费启动开销，因此总开销会超过那种方法（25个单位）的某个值。

例1-8：使用已经讨论过的列，尝试连接两个表：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   -> Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
             rows=10 width=0)
               Index Cond: (unique1 < 10)
   -> Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91
      rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在这个计划中，有一个嵌套循环连接结点，它有两个表扫描作为输入或子结点。该结点的摘要行的缩进反映了计划树的结构。连接的第一个（或"outer"）子结点是一个与前面见到的相似的位图扫描。它的开销和行计数与从`SELECT ... WHERE unique1 < 10`得到的相同，因为将`WHERE`子句`unique1 < 10`用在了那个结点上。`t1.unique2 = t2.unique2`子句现在还不相关，因此它不影响 outer 扫描的行计数。嵌套循环连接结点将为从 outer 子结点得到的每一行运行它的第二个（或"inner"）子结点。当前 outer 行的列值可以被插入 inner 扫描。这里，来自 outer 行的`t1.unique2`值是可用的，所以得到的计划和开销与前面见到的简单`SELECT ... WHERE t2.unique2 = constant`情况相似（估计的开销实际上比前面看到的略低，是因为在`t2`上的重复索引扫描会利用到高速缓存）。循环结点的开销则被以 outer 扫描的开销为基础设置，外加对每一个 outer 行都要进行一次 inner 扫描 （10 * 7.87），再加上用于连接处理一点 CPU 时间。

在此例子里，连接的输出行计数等于两个扫描的行计数的乘积，但通常并不是所有的情况中都如此，因为可能有同时提及两个表的额外`WHERE`子句，并且因此它只能被应用于连接点，而不能影响任何一个输入扫描。

例1-9

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                           QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
```

```
  -> Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
        Recheck Cond: (unique1 < 10)
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
            rows=10 width=0)
              Index Cond: (unique1 < 10)
  -> Materialize  (cost=0.29..8.51 rows=10 width=244)
        -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..8.46
           rows=10 width=244)
              Index Cond: (unique2 < 10)
```

条件t1.hundred < t2.hundred不能在tenk2_unique2索引中被测试，因此它被应用在连接结点。这缩减了连接结点的估计输出行计数，但是没有改变任何输入扫描。

注意这里规划器选择了"物化"连接的 inner 关系，方法是在它的上方放了一个物化计划结点。这意味着t2索引扫描将只被做一次，即使嵌套循环连接结点需要读取其数据十次（每个来自 outer 关系的行都要读一次）。物化结点在读取数据时将它保存在内存中，然后在每一次后续执行时从内存返回数据。

在处理外连接时，可能会看到连接计划结点同时附加有"连接过滤器"和普通"过滤器"条件。连接过滤器条件来自于外连接的ON子句，因此一个无法通过连接过滤器条件的行也能够作为一个空值扩展的行被发出。但是一个普通过滤器条件被应用在外连接条件之后并且因此无条件移除行。在一个内连接中这两种过滤器类型没有语义区别。

例 1-10：改变查询的选择度，将得到一个不同的连接计划：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                              QUERY PLAN
---------------------------------------------------------------------
 Hash Join  (cost=230.47..7198 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101
             width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
                   rows=101 width=0)
                     Index Cond: (unique1 < 100)
```

这里规划器选择了使用一个哈希连接，在其中一个表的行被放入一个内存哈希表，在这之后其他表被扫描并且为每一行查找哈希表来寻找匹配。同样要注意缩进是如何反映计划结构的：tenk1上的位图扫描是哈希结点的输入，哈希结点会构造哈希表。然后哈希表会返回给哈希连接结点，哈希连接结点将从它的 outer 子计划读取行，并为每一个行搜索哈希表。

例 1-11：归并连接

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                              QUERY PLAN
---------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28
       rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.8.200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000
             width=244)
```

归并连接要求它的输入数据被按照连接键排序。在这个计划中，tenk1数据被使用一个索引扫描排序，以便能够按照正确的顺序来访问行。但是onek则更倾向于一个顺序扫描和排序，因为在那个表中有更多行需要被访问（对于很多行的排序，顺序扫描加排序常常比一个索引扫描好，因为索引扫描需要非顺序的磁盘访问）。

一种查看变体计划的方法是强制规划器丢弃它认为开销最低的任何策略，可使用*KingbaseES V8R3《管理员手册》IV. 服务器管理-"KingbaseES服务器参数配置手册"-8.2 规划器方法配制*中描述的启用/禁用标志实现。

例1-12：不认同顺序扫描加排序是处理表onek的最佳方法，可尝试：

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                          QUERY PLAN
---------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28
       rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79
       rows=1000 width=244)
```

显示规划器认为用索引扫描来排序onek的开销要比用顺序扫描加排序的方式高大约12%。当然，可以通过使用EXPLAIN ANALYZE来研究是否为真。

1.2. EXPLAIN ANALYZE

通过使用EXPLAIN的ANALYZE选项来检查规划器估计值的准确性。使用此选项，EXPLAIN会实际执行该查询，显示真实的行计数和在每个计划结点中累计的真实运行时间，并将有EXPLAIN显示的一个普通估计值。详见例1-13。

例1-13.：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=
 0.128..0.377 rows=10 loops=1)
   -> Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
      (actual time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
             rows=10 width=0) (actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   -> Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91
      rows=1 width=244) (actual time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

注意"actual time"值是以毫秒计的真实时间，而cost估计值被以捏造的单位表示，因此它们不大可能匹配上。在这里面要查看的最重要的一点是估计的行计数是否合理地接近实际值。在本例中，估计值是完全正确的，但在实际操作中非常少见。

在某些查询计划中，可以多次执行一个子计划结点。例如，inner 索引扫描可能会因为上层嵌套循环计划中的每一个 outer 行而被执行一次。在这种情况下，loops值报告了执行该结点的总次数，并且 actual time 和行数值是这些执行的平均值。这是为了让这些数字能够与开销估计被显示的方式有可比性。将这些值乘上loops值可以得到在该结点中实际消耗的总时间。在上面的例子中，执行tenk2的索引扫描总共花费了 0.220 毫秒。

在某些情况中，EXPLAIN ANALYZE会显示计划结点执行时间和行计数之外的额外执行统计信息。

例1-14：排序和哈希结点将提供额外的信息

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY
t1.fivethous;

                        QUERY PLAN
-------------------------------------------------------------
```

```
Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=
7.761..7.774 rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort  Memory: 77kB
  ->  Hash Join  (cost=230.47..7198 rows=101 width=488) (actual
      time=0.711..7.427 rows=100 loops=1)
        Hash Cond: (t2.unique2 = t1.unique2)
        ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000
            width=244) (actual time=0.007..2.583 rows=10000 loops=1)
        ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual
            time=0.659..0.659 rows=100 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 28kB
              ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20
                 rows=101 width=244) (actual time=0.080..0.526
                 rows=100 loops=1)
                    Recheck Cond: (unique1 < 100)
                    ->  Bitmap Index Scan on tenk1_unique1  (cost=
                        0.00..5.04 rows=101 width=0) (actual time=
                        0.049..0.049 rows=100 loops=1)
                          Index Cond: (unique1 < 100)
Planning time: 0.194 ms
Execution time: 8.008 ms
```

排序结点显示了使用的排序方法（尤其是，排序是在内存中还是磁盘上进行）和需要的内存或磁盘空间量。哈希结点显示了哈希桶的数量和批数，以及被哈希表所使用的内存量的峰值（如果批数超过一，将会涉及到磁盘空间使用，但是并不会被显示）。

例1-15： 被一个过滤器条件移除的行数

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..4800 rows=7000 width=244)
 (actual time=0.016..5.107 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

这些值对于被应用在连接结点上的过滤器条件特别有价值。只有在至少有一个被扫描行或者在连接结点中一个可能的连接对被过滤器条件拒绝时，"Rows Removed"行才会出现。

一个与过滤器条件相似的情况出现在"有损"索引扫描中。

例1-16： 下列查询的搜索中包含一个指定点的多边形：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '
(0.5,2.0)';
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32)
 (actual time=0.044..0.044 rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

规划器认为此采样表太小不值得进行一次索引扫描，因此将得到了一个普通的顺序扫描，其中的所有行都被过滤器条件拒绝。

例1-17： 若强制使得一次索引扫描可以被使用，将得到：

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '
(0.5,2.0)';

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.1.8.15
 rows=1 width=32) (actual time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

索引返回一个候选行，然后被索引条件的重新检查拒绝。这是因为一个 GiST 索引对于多边形包含测试是 "有损的"：它确实返回覆盖目标的多边形的行，然后用户必须在那些行上做精确的包含性测试。

例1-18： EXPLAIN有一个BUFFERS选项可以和ANALYZE一起使用来得到更多运行时统计信息：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1
< 100 AND unique2 > 9000;

                        QUERY PLAN
-------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
 (actual time=0.32.0.342 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual
       time=0.309..0.309 rows=0 loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
             rows=101 width=0) (actual time=0.04.0.043 rows=100
             loops=1)
               Index Cond: (unique1 < 100)
```

```
                Buffers: shared hit=2
          ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78
              rows=999 width=0) (actual time=0.227..0.227 rows=999
              loops=1)
                Index Cond: (unique2 > 9000)
                Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

BUFFERS提供的数字帮助标识查询的哪些部分是对 I/O 最敏感的。

记住因为EXPLAIN ANALYZE会实际的运行查询，所以任何副作用都将可能发生，即使查询可能输出的任何结果被丢弃来支持打印EXPLAIN数据。

例1-19： 仅分析一个数据修改查询而不改变表，可以在分析完后回滚命令

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1
WHERE unique1 < 100;

                      QUERY PLAN
------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250)
 (actual time=14.628..14.628 rows=0 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=
       101 width=250) (actual time=0.101..0.439 rows=100
       loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
      rows=101 width=0) (actual time=0.04.0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

如上述例子所展示，当查询是一个INSERT、UPDATE或DELETE命令时，应用表更改的实际工作由顶层插入、更新或删除计划结点完成。这个结点之下的计划结点执行定位旧行或者计算新数据的工作。因此在上面，将看到已经见过的位图表扫描，它的输出被交给一个更新结点，更新结点会存储被更新过的行。还有一点值得注意的是，尽管数据修改结点可能要可观的运行时间（这里，它消耗最大份额的时间），规划器当前并没有对开销估计增加任何东西来说明这些工作。这是因为这些工作对每一个正确的查询计划都得做，所以它不影响计划的选择。

例1-20： 当UPDATE或者DELETE命令影响继承层次时，输出为：

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                               QUERY PLAN
-------------------------------------------------------------------------
```

```
Update on parent  (cost=0.00..24.53 rows=4 width=14)
  Update on parent
  Update on child1
  Update on child2
  Update on child3
  -> Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
        Filter: (f1 = 101)
  -> Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1
     width=14)
        Index Cond: (f1 = 101)
  -> Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1
     width=14)
        Index Cond: (f1 = 101)
  -> Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1
     width=14)
        Index Cond: (f1 = 101)
```

在上述例子中，更新节点需要考虑三个子表以及最初提到的父表。因此有四个输入的扫描子计划，每一个对应于一个表。为清楚起见，在更新节点上标注了将被更新的相关目标表，显示的顺序与相应的子计划相同（这些标注是从KingbaseES V8 开始新增的，在老版本中读者必须通过观察子计划才能知道这些目标表）。

EXPLAIN ANALYZE显示的Planning time是从一个已解析的查询生成查询计划并进行优化所花费的时间，其中不包括解析和重写。

EXPLAIN ANALYZE显示的Execution time包括执行器的启动和关闭时间，以及运行被触发的任何触发器的时间，但是它不包括解析、重写或规划的时间。如果有花在执行BEFORE触发器的时间，它将被包括在相关的插入、更新或删除结点的时间内；但是用来执行AFTER触发器的时间没有被计算，因为AFTER触发器是在整个计划完成后被触发的。每个触发器（BEFORE或AFTER）也被独立地显示。注意延迟约束触发器将不会被执行，直到事务结束，并且因此完全不会被EXPLAIN ANALYZE考虑。

1.3. 警告

在两种有效的方法中EXPLAIN ANALYZE所度量的运行时间可能偏离同一个查询的正常执行时间。首先，由于不会有输出行被递交给客户端，网络传输开销和 I/O 转换开销没有被包括在内。其次，由EXPLAIN ANALYZE所增加的度量时间可能会很大，特别是在那些gettimeofday()操作系统调用很慢的机器上。可以使用sys_test_timing工具来度量在系统上的计时开销。

EXPLAIN的结果不应该被外推到与实际测试的非常不同的情况。例如，一个很小的表上的结果不能被假定成适合大型表。规划器的开销估计不是线性的，并且因此它可能为一个更大或更小的表选择不同的计划。一个极端例子是，在一个只占据一个磁盘页面的表上，不管索引是否可用总能得到一个顺序扫描计划，而。规划器认识到它在任何情况下都将采用一次磁盘页面读取来处理

该表，因此用额外的页面读取去查看一个索引是没有价值的。

在一些情况中，实际的值和估计的值不会匹配得很好，但这并非错误。一种这样的情况发生在计划结点的执行被LIMIT或类似的效果所停止。

例1-21：在 LIMIT查询中

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND
unique2 > 9000 LIMIT 2;

                      QUERY PLAN
-------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=
 0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42
       rows=10 width=244) (actual time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

索引扫描结点的估计开销和行计数被显示成好像它会运行到完成。但是实际上限制结点在得到两个行之后就停止请求行，因此实际的行计数只有 2 并且运行时间远低于开销估计所建议的时间。这并非估计错误，仅仅是估计值和实际值显示方式上的不同。

归并连接也存在类似的现象。如果一个归并连接用尽了一个输入并且其中的最后一个键值小于另一个输入中的下一个键值，它将停止读取另一个输入。在这种情况下，不会产生更多的匹配并且不需要扫描第二个输入的剩余部分。将导致不读取一个子结点的所有内容，其结果就像在LIMIT中所提到的。另外，如果 outer（第一个）子结点包含带有重复键值的行，inner（第二个）子结点会被倒退并且被重新扫描来找能匹配那个键值的行。EXPLAIN ANALYZE会统计相同 inner 行的重复发出，就好像它们是真实的额外行。当有很多 outer 重复时，对 inner 子计划结点所报告的实际行计数会显著地大于实际在 inner 关系中的行数。

出于实现的限制，BitmapAnd 和 BitmapOr 结点总是报告它们的实际行计数为零。

2. 规划器使用的统计信息

在上一节中，查询规划器需要估计一个查询要检索的行数，这样才能对查询计划做出好的选择。本节对系统用于这些估计的统计信息进行一个简要的介绍。

统计信息的一个部分是每个表和索引中的项的总数，以及每个表和索引占用的磁盘块数。这些信息保存在sys_class表的reltuples和relpages列中。

例2-1：查看统计信息

```
SELECT relname, relkind, reltuples, relpages
FROM sys_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

tenk1包含 10000 行，其索引也有这么多行，但是索引远比表小。

出于效率考虑，reltuples和relpages不是实时更新的，因此它们通常包含部分过时的值。它们被VACUUM、ANALYZE和几个 DDL 命令（例如CREATE INDEX）更新。一个不扫描全表的VACUUM或ANALYZE操作（常见情况）将以它扫描的部分为基础增量更新reltuples计数，这就导致会产生一个近似值。在任何情况中，规划器将缩放它在sys_class中找到的值来匹配当前的物理表尺寸，这样可以得到一个较接近的近似值。

大多数查询只是检索表中行的一部分，因为它们会通过WHERE子句限制行。因此规划器需要估算WHERE子句的选择度，即符合WHERE子句中每个条件的行的比例。用于这个任务的信息存储在sys_statistic系统表中。在sys_statistic中的项由ANALYZE和VACUUM ANALYZE命令更新，并且总是近似值。

除直接查看sys_statistic之外，手工检查统计信息的时候最好查看其视图sys_stats。所有人均可读取sys_stats，而sys_statistic只能由超级用户读取（这样可以避免非授权用户从统计信息中获取其他人的表的内容相关信息。sys_stats视图被限制为只显示当前用户可读的表）。

例2-2：手工检查统计信息

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM sys_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                        Ramp+
         |           |            | I- 880                        Ramp+
         |           |            | Sp Railroad                       +
         |           |            | I- 580                            +
         |           |            | I- 680                        Ramp
 name    | t         |  -0.284859 | I- 880                        Ramp+
         |           |            | I- 580                        Ramp+
```

```
         |            |             | I- 680                          Ramp+
         |            |             | I- 580                              +
         |            |             | State Hwy 13                    Ramp
(2 rows)
```

注意，这两行显示了相同的列，一个对应开始于road表（inherited=t）的完全继承层次，另一个只包括road表本身（inherited=f）。

ANALYZE在sys_statistic中存储的信息量（特别是每个列的most_common_vals中的最大项数和histogram_bounds数组）可以用ALTER TABLE SET STATISTICS命令为每一列设置，或者通过设置配置变量default_statistics_target进行全局设置。目前的默认限制是 100 项。提高该限制可能会让规划器的估计更加精准（特别是对那些有不规则数据分布的列），其代价是在sys_statistic中消耗了更多空间，并且需要更多的时间来计算估计数值。相比之下，比较低的限制值更适合那些数据分布比较简单的列。

3. 用显式JOIN子句控制规划器

用户可以在一定程度上用显式JOIN语法控制查询规划器。

在一个简单的连接查询中：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以自由地按照任何顺序连接给定的表。例如，它可以生成一个使用WHERE条件a.id = b.id连接 A 到 B 的查询计划，然后用另外一个WHERE条件把 C 连接到这个连接表。或者可以先连接 B 和 C 然后再连接 A 得到同样的结果。也可以连接 A 到 C 然后把结果与 B 连接 — 。但这样做效率低，因为必须生成完整的 A 和 C 的迪卡尔积，而在WHERE子句中没有可用条件来优化该连接（KingbaseES执行器中的所有连接都发生在两个输入表之间，所以它必须以这些形式之一建立结果）。重要的一点是这些不同的连接可以给出语义等效的结果，但在执行开销上却可能有巨大的差别。因此，规划器会对它们进行探索并尝试找出最高效的查询计划。

当一个查询只涉及两个或三个表时，不需要考虑很多连接顺序。但是可能的连接顺序数随着表数目的增加成指数增长。当超过十个左右的表以后，实际上根本不可能对所有可能性做一次穷举搜索，甚至对六七个表的连接都需要相当长的时间进行规划。当有太多的输入表时，KingbaseES规划器将从穷举搜索切换为一种遗传概率搜索，它只需要考虑有限数量的可能性（切换的阈值用geqo_threshold运行时参数设置）。遗传搜索用时更少，但是并不一定会找到最好的计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度：

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

尽管这个查询语句表面上和前一个非常相似，但它们的语义却不同，因为就算 A 里有任何一行不能匹配 B 和 C的连接表中的行，它也必须被输出。因此这里规划器对连接顺序没有什么选择：它必须先连接 B 到 C，然后把 A 连接到该结果上。相应地，这个查询比前面一个花在规划上的时间更少。在其它情况下，规划器就有可能确定多种连接顺序都是安全的。例如：

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid
= c.id);
```

将 A 首先连接到 B 或 C 都是有效的。当前，只有FULL JOIN完全约束连接顺序。大多数涉及LEFT JOIN或RIGHT JOIN的实际情况都在某种程度上可以被重新排列。

显式连接语法（INNER JOIN、CROSS JOIN或无修饰的JOIN）在语义上和FROM中列出输入关系是一样的，因此它不约束连接顺序。

即使大多数类型的JOIN并不完全约束连接顺序，但仍然可以指示KingbaseES查询规划器将所有JOIN子句当作有连接顺序的约束来对待。例如，下列的三个查询在逻辑上是等效的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref
= c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果要求规划器遵循JOIN的顺序，那么第二个和第三个要比第一个花在规划上的时间少。这个效果对于只有三个表的连接而言是微不足道的，但对于数目众多的表，将非常有用。

强制规划器遵循显式JOIN的连接顺序，可以把运行时参数join_collapse_limit设置为 1。

用户不必为了缩短搜索时间而完全的约束连接顺序，因为可以在一个普通FROM列表里使用JOIN操作符：

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置join_collapse_limit = 1，那么这就强迫规划器先把 A 连接到 B，然后再连接到其它的表上，但并不约束它的这个选择。在此例子中，可能的连接顺序的数目减少了 5 倍。

按照这种方法约束规划器的搜索范围是一个有用的技巧，不管是对减少规划时间还是对引导规划器生成好的查询计划。如果规划器按照默认选择了一个糟糕的连接顺序，可以通过JOIN语法强迫它选择一个更好的顺序 — 假设用户已经知道一个更好的顺序。推荐通过实验确定最好的顺序。

一个类似的影响规划时间的问题是把子查询压缩到其父查询中：

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能在使用包含连接的视图时出现；该视图的SELECT规则将被插入到引用视图的地方，得到与上文非常相似的查询。通常，规划器会尝试把子查询压缩到父查询里：

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

通常会生成一个比独立的子查询更好些的计划（例如，outer 的WHERE条件可能先把 X 连接到 A 上，这样就消除了 A 中多行，因此避免了形成子查询的全部逻辑输出）。但是同时，会增加规划的时间；此处，将用五路连接问题替代两个独立的三路连接问题。这样的差别是巨大的，因为可能的计划数是按照指数增长的。如果有超过from_collapse_limit个FROM项将会导致父查询规划时间显著增加，规划器将尝试通过停止提升子查询来避免卡在巨大的连接搜索问题中。可以通过调高或调低这个运行时参数在规划时间和计划的质量之间取得平衡。

from_collapse_limit和join_collapse_limit的命名相似，因为它们做的几乎是同一件事：一个控制规划器何时将把子查询"平面化"，另外一个控制何时把显式连接平面化。通常，要么把join_collapse_limit设置成和from_collapse_limit一样（这样显式连接和子查询的行为类似），要么把join_collapse_limit设置为 1（如果想用显式连接控制连接顺序）。但可以把它们设置成不同的值，这样就可以细粒度地调节规划时间和运行时间之间的平衡。

4. 填充一个数据库

第一次填充数据库时可能需要插入大量的数据。本节包含一些如何让这个处理尽可能高效的建议。

4.1. 禁用自动提交

在使用多个INSERT时，关闭自动提交并且只在最后做一次提交（在普通 SQL 中，这意味着在开始发出BEGIN并且在结束时发出COMMIT。）。如果允许每一个插入都被独立地提交，KingbaseES要为每一个被增加的行做很多工作。在一个事务中做所有插入的一个额外好处是：如果一个行插入失败则所有之前插入的行都会被回滚，这样不会被卡在部分载入的数据中。

4.2. 使用COPY

使用COPY在一条命令中装载所有需要插入的内容，而不是一系列INSERT命令。 COPY命令是为装载大量行而优化过的；没INSERT那么灵活，但是在大量数据装载时产生的负荷更少。因为COPY是单条命令，因此使用这种方法填充表时无须关闭自动提交。

如果不能使用COPY，那么使用PREPARE来创建一个预备INSERT语句也有所帮助，然后根据需要使用EXECUTE多次。避免重复分析和规划INSERT的负荷。不同接口以不同的方式提供该功能，可参阅接口文档中的"预备语句"。

请注意，在载入大量行时，使用COPY总是比使用INSERT快，即便使用PREPARE并且将多个插入成批地放入一个单一事务也是如此。

同样的事务中，COPY比更早的CREATE TABLE或TRUNCATE命令更快。在这种情况下，不需要写 WAL，因为在一个错误的情况下，包含新载入数据的文件不管怎样都将被移除。但是，只有当wal_level设置为minimal（此时所有的命令必须写 WAL）时才会应用这种考虑。

4.3. 移除索引

如果正在载入一个新创建的表，最快的方法是创建该表，用COPY批量载入该表的数据，之后创建表需要的任何索引。在已存在数据的表上创建索引要比在每一行被载入时增量地更新它更快。

如果正在对现有表增加大量的数据，删除索引、载入表然后重新创建索引可能是最好的方案。当然，在缺少索引的期间，其它数据库用户的数据库性能将会下降。用户在删除唯一索引之前需谨慎考虑，因为唯一约束提供的错误检查在缺少索引的时候会丢失。

4.4. 移除外键约束

和索引一样，"成批地"检查外键约束比一行行检查效率更高。因此，先删除外键约束、载入数据然后重建约束会很有用。同样，载入数据和约束缺失期间错误检查的丢失之间也存在平衡。

重要的是，当在已有外键约束的情况下向表中载入数据时，每个新行需要增加一个在服务器的待处理触发器事件（因为是一个触发器的触发会检查行的外键约束）列表的条目。载入数百万行会导致触发器事件队列溢出可用内存，造成不能接受的交换或者甚至是命令的彻底失败。因此在载入大量数据时，需要（而不仅仅是期望）删除并重新应用外键。如果临时移除约束不可接受，那唯一的其他办法可能是就是将载入操作分解成更小的事务。

4.5. 增加maintenance_work_mem

在载入大量数据时，临时增大maintenance_work_mem配置参数可以改进性能。该参数也可以帮助加速CREATE INDEX命令和ALTER TABLE ADD FOREIGN KEY命令。它不会对COPY本身起很大作用，所以这个建议只有在使用上述的一个或两个命令时才有用。

4.6. 增加max_wal_size

临时增大max_wal_size配置参数也可以让大量数据载入更快。这是因为向KingbaseES中载入大量的数据将导致检查点的启动比平常（由checkpoint_timeout配置参数指定）更频繁。无论何时启动一个检查点时，所有脏页都必须被刷写到磁盘上。通过在批量数据载入时临时增加max_wal_size，所需的检查点数目可以被缩减。

4.7. 禁用 WAL 归档和物理同步

当使用 WAL 归档或物理同步向一个安装中载入大量数据时，在录入结束后执行一次新的基础备份比处理大量的增量 WAL 数据更快。为了防止载入时记录增量 WAL，通过将wal_level设置为minimal、将archive_mode设置为off以及将max_wal_senders设置为零来禁用归档和物理同步。但需注意，修改这些设置需要重启服务。

除了可以避免归档器或 WAL 发送者处理 WAL 数据的时间，这样做将实际上也可以使某些命令执行更快，因为它们被设计为在wal_level为minimal时完全不写 WAL（通过在最后执行一个fsync而不是写 WAL，能以更小地代价保证崩溃时的安全性）。这适用于下列命令：

- CREATE TABLE AS SELECT
- CREATE INDEX（以及类似 ALTER TABLE ADD PRIMARY KEY的变体）
- ALTER TABLE SET TABLESPACE
- CLUSTER
- COPY FROM，当目标表已经被创建或者在同一个事务的早期被截断

4.8. 事后运行ANALYZE

KingbaseES建议用户运行ANALYZE，当表中的数据分布被调整时（包括向表中批量载入大量数据）。运行ANALYZE（或者VACUUM ANALYZE）保证规划器有表的最新统计信息。如果没有统计数据或者统计数据过时，规划器在查询规划时可能做出错误的决定，导致在任意表上的查询语句性能变低。需注意，如果启用 autovacuum 守护进程，该进程可能会自动运行ANALYZE。

4.9. 关于sys_dump的一些注释

sys_dump生成的转储脚本将自动应用上述若干个（但不是全部）技巧。要尽可能快地载入sys_dump转储，用户需要手动做一些额外的事情（请注意，这些要点适用于恢复一个转储，而不适用于创建它时。同样的要点也适用于使用ksql载入一个文本转储或用sys_restore从一个sys_dump归档文件载入）。

默认情况下，sys_dump使用COPY，并且当它在生成一个完整的模式和数据转储时，会很小心地先装载数据，然后创建索引和外键。因此在这种情况下，一些指导方针是被自动处理的。用户需要做的是：

- 为maintenance_work_mem和max_wal_size设置适当的（即比正常值大的）值。
- 如果使用 WAL 归档或物理同步，在转储时考虑禁用它们。在载入转储之前，可通过将archive_mode设置为off、将wal_level设置为minimal以及将max_wal_senders设置为零（在录入dump前）来实现禁用。之后，将它们设回正确的值并执行一次新的基础备份。
- 采用sys_dump和sys_restore的并行转储和恢复模式进行实验并且找出要使用的最佳并发任务数量。通过使用-j选项的并行转储和恢复能带来比串行模式高得多的性能。
- 考虑是否应该在一个单一事务中恢复整个转储。要这样做，将-1或--single-transaction命令行选项传递给ksql或sys_restore。 当使用这种模式时，即使是一个很小的错误也会回滚整个恢复，可能会丢弃已经处理了很多个小时的工作。根据数据间的相关性，可能手动清理更好。如果使用一个单一事务并且关闭了 WAL 归档，COPY命令将运行得最快。
- 如果在数据库服务器上有多个 CPU 可用，可以考虑使用sys_restore的--jobs选项。允许并行数据载入和索引创建。
- 之后运行ANALYZE。

一个只涉及数据的转储仍将使用COPY，但它不会删除或重建索引，并且它通常不会触碰外键。[1]因此当载入一个只有数据的转储时，如果希望使用那些技术，需要负责删除并重建索引和外键。在载入数据时增加max_wal_size仍然有用，但避免增加maintenance_work_mem；不如说在以后手工重建索引和外键时已经做了这些。并且不要忘记在完成后执行ANALYZE。

备注

[1] 可以通过使用--disable-triggers选项的方法获得禁用外键的效果 — 但此操作是消除外键验证。使用该选项可能会插入坏数据。

5. 非持久设置

持久性是数据库一个保证已提交事务的记录的特性（即使是发生服务器崩溃或断电）。然而，持久性会明显增加数据库的负荷，因此如果站点不需要这个保证，KingbaseES可以被配置得运行更快。在这种情况下，可以调整下列配置来提高性能。除了下面列出的，在数据库软件崩溃的情况下也能保证持久性。当这些设置被使用时，只有操作系统突然停止会产生数据丢失或损坏的风险。

- 将数据库集簇的数据目录放在一个内存支持的文件系统上（即RAM磁盘）。将消除所有的数据库磁盘 I/O，但将数据存储限制到可用的内存量（可能有交换区）。
- 关闭fsync；不需要将数据刷入磁盘。
- 关闭synchronous_commit；可能不需要在每次提交时强制把WAL写入磁盘。这种设置可能会在数据库崩溃时带来事务丢失的风险（但不会破坏数据）。
- 关闭full_page_writes；不需要警惕部分页面写入。
- 增加max_wal_size和checkpoint_timeout；将降低检查点的频率，但会增加`sys_xlog`目录的存储要求。
- 创建不做日志的表来避免WAL写入，但这会让表在崩溃时不安全。

并行查询

KingbaseES能设计出利用多 CPU 让查询更快的查询计划。这种特性被称为并行查询。由于现实条件的限制或因为没有比连续查询计划更快的查询计划存在，很多查询并不能从并行查询获益。但是，对于那些可以从并行查询获益的查询来说，并行查询带来的速度提升是显著的。很多查询在使用并行查询时查询速度比之前快了超过两倍，有些查询是以前的四倍甚至更多的倍数。那些访问大量数据但只返回其中少数行给用户的查询最能从并行查询中获益。本文档将介绍并行查询工作的细节，以及用户应当在什么情况下使用并行查询。

1. 并行查询如何工作

当优化器判断对于某一个特定的查询，并行查询是最快的执行策略时，优化器将创建一个查询计划。该计划包括一个 *Gather* 节点。下面是一个简单的例子：

```
EXPLAIN SELECT * FROM sysbench_accounts WHERE filler LIKE '%x%';
                          QUERY PLAN
------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on sysbench_accounts  (cost=0.00..216018.33
       rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

在任意情形下，Gather节点都只有一个子计划，它是将被并行执行的计划的一部分。如果Gather节点位于计划树的最顶层，那么整个查询将并行执行。若位于计划树的其他位置，那么只有查询的那一部分会并行执行。在上述例子中，查询只访问了一个表，因此除Gather节点本身之外只有一个计划节点。因为该计划节点是 Gather节点的子节点，因此它会并行执行。

使用 EXPLAIN命令, 能够看到规划器选择的工作者数量。当查询执行期间到达Gather节点时，实现用户会话的进程将会请求和规划器选中的工作者数量一样多的后台工作者进程。任何时候能够存在的后台工作者进程的总数由max_worker_processes限制，因此一个并行查询可能会使用比规划中少的工作者来运行，甚至有可能根本不使用工作者。最优的计划可能取决于可用的工作者的数量，因此这可能会导致不好的查询性能。如果这种情况经常发生，就应当考虑提高max_worker_processes的值，这样更多的工作者可以同时运行；或者降低max_parallel_workers_per_gather，这样规划器会要求少一些的工作者。

为一个给定并行查询成功启动的后台工作者进程都将会执行Gather 节点的后代计划的一部分。这些工作者的领导者也将执行该计划，但它还有一个额外的任务：它还必须读取所有由工作者产生的元组。当整个计划的并行部分只产生了少量元组时，领导者通常将表现为一个额外的加速查询执行的工作者。反过来，当计划的并行部分产生大量的元组时，领导者将几乎全用来读取由工作者产生的元组并且执行Gather节点上层计划节点所要求的任何进一步处理。在这种情况下，领导者所作的执行并行部分的工作将会很少。

2. 何时会用到并行查询？

部分设置会导致查询规划器在任何情况下都不生成并行查询计划。为了让并行查询计划能够顺利执行，必须配置好下列设置：

- max_parallel_workers_per_gather必须被设置为大于零的值。更普遍的原则是所用的工作者数量不超过max_parallel_workers_per_gather所配置的数量。

- dynamic_shared_memory_type必须被设置为除none之外的值。并行查询要求动态共享内存以便在合作的进程之间传递数据。

此外，系统一定不能运行在单用户模式下。因为在单用户模式下，整个数据库系统运行在单个进程中，没有后台工作者进程可用。

如果下面的任一条件为真，即便对一个给定查询通常可以产生并行查询计划，规划器也不会为它产生并行查询计划：

- 查询要写任何数据或者锁定任何数据库行。如果一个查询在顶层或者 CTE 中包含了数据修改操作，那么不会为该查询产生并行计划。这是当前实现的一个限制，未来的版本中可能会有所改进。
- 查询可能在执行过程中被暂停。只要在系统认为可能发生部分或者增量式执行，就不会产生并行计划。例如：用DECLARE CURSOR创建的游标将永远不会使用并行计划。类似地，一个FOR x IN query LOOP .. END LOOP形式的 PL/SQL 循环也永远不会使用并行计划，因为当并行查询进行时，并行查询系统无法确保循环中的代码执行起来是安全的。
- 使用了任何被标记为PARALLEL UNSAFE的函数的查询。大多数系统定义的函数都被标记为PARALLEL SAFE，但是用户定义的函数默认被标记为PARALLEL UNSAFE。
- 该查询运行在另一个已经存在的并行查询内部。例如，如果一个被并行查询调用的函数自身发出一个 SQL 查询，那么该查询将不会使用并行计划。这是当前实现的一个限制，但是并不值得移除这个限制，因为它会导致单个查询使用大量的进程。
- 事务隔离级别是可串行化。这是当前实现的一个限制。

即使一些特定的查询已经产生了并行查询计划，在某些情况下也不会并行执行该计划。如果发生这种情况，那么领导者将会自己执行该计划在Gather节点之下的部分，如同Gather节点不存在一样。上述情况将在满足下面的任一条件时发生：

- 由于后台工作者进程的总数不能超过max_worker_processes，导致不能得到后台工作者进程。
- 客户端发送了一个执行消息，并且消息中要求取元组的数量不为零。因为libkci当前未提供方法来发送这种消息，所以此情况只可能发生在不依赖libkci的客户端中。如果这种情况经常发生，建议在可能发生的会话中设置 max_parallel_workers_per_gather，避免产生连续运行时次优的查询计划。
- 事务隔离级别是可串行化。这种情况通常不会出现，因为当事务隔离级别是可串行化时不会产生并行查询计划。但是，如果在产生计划之后并且在执行计划之前把事务隔离级别改成可串行化，这种情况就有可能发生。

3. 并行计划

因为每个工作者只执行完成计划的并行部分，所以不可能简单地产生一个普通查询计划并使用多个工作者运行它。每个工作者都会产生输出结果集的一个完全拷贝，因而查询并不会比普通查询运行得更快甚至还会产生不正确的结果。相反，计划的并行部分一定被查询优化器在内部当作一个部分计划。也就是说，一定要这样来创建计划，使得每个将执行该计划的进程只产生输出行的一个子集，这样可以保证每个需要被输出的行刚好会被合作进程产生一次。

3.1. 并行扫描

当前唯一一种被修改用于并行查询的扫描类型是顺序扫描。因此在并行计划中的驱动表将总是被使用并行顺序扫描进行扫描。关系的块将被划分给合作进程。一次发放一个文件块，这样对于关系的访问仍然保持为顺序访问。在请求一个新页面之前，每一个进程将访问分配给它的页面上的每一个元组。

3.2. 并行连接

驱动表将被使用嵌套循环或者哈希连接连接到一个或者多个其他表。在连接的外侧可以是任何一种被规划器支持可以安全地在并行工作者中运行的非并行计划。例如，它可以是一个索引扫描，基于从内表取得的一列来查找值。每个工作者都将会完整地执行外侧的计划，这也是为什么这里不能支持归并连接。归并连接的外侧常常涉及到排序整个内表，即便使用索引，多次在内表上进行完全索引扫描也效率不高。

3.3. 并行聚集

将查询的聚集部分整个地并行执行是不可能的。例如，如果一个查询涉及到选择COUNT(*)，每个工作者可以计算一个总和，但是这些总和需要被整合在一起以产生最终的答案。如果一个计划涉及到GROUP BY子句，需要为每个组计算出一个单独的总和。即使聚集不能完全地并行执行，但涉及聚集的查询常常是并行查询很好的候选，因为它们通常读很多行但只返回少数几行给客户端。返回很多行给客户端的查询常常受制于客户端读取数据的速度，这种情况下并行查询帮助不大。

KingbaseES通过做两次聚集来支持并行聚集。第一次，每个参与查询计划并行部分执行的进程执行一个聚集步骤，为进程发现的每个分组产生一个部分结果。这在计划中反映为一个PartialAggregate节点。第二次，部分结果通过Gather节点传输给领导者。最后，领导者对所有工作者的部分结果进行重聚集以得到最终的结果。这在计划中反映为一个FinalizeAggregate节点。

并行聚集并不能支持所有的情况。每个聚集对于并行机制一定要是安全的，并且必须有一个结合函数。如果聚集有一个internal类型的转移状态，必须有序列化和反序列化函数。详见CREATE AGGREGATE。对于有序集聚集或者查询涉及GROUPING SETS时不支持并行聚集。只有当查询中涉及的所有连接也是计划中并行不分的一部分时，才能使用并行聚集。

3.4. 并行计划小贴士

如果一个查询能产生并行计划但实际并未产生，可以尝试减小parallel_setup_cost或者parallel_tuple_cost。当然，该计划可能比规划器优先产生的顺序计划还要慢，但也不总是如此。如果将其设置为很小的值（例如把它们设置为零）也不能得到并行计划，那就可能是有某种原因导致查询规划器无法为查询产生并行计划。

在执行一个并行计划时，可以用EXPLAIN (ANALYZE,VERBOSE)来显示每个计划节点在每个工作者上的统计信息。这些信息有助于确定是否所有的工作被均匀地分发到所有计划节点以及从总体上理解计划的性能特点。

4. 并行安全性

规划器把查询中涉及的操作分类成并行安全、并行受限或者并行不安全。并行安全的操作不会与并行查询的使用产生冲突。并行受限的操作不能在并行工作者中执行，但能够在并行查询的领导者中执行。因此，并行受限的操作不能出现在Gather节点之下，但能够出现在包含有Gather节点的计划的其他位置。并行不安全的操作不能在并行查询中执行，甚至不能在领导者中执行。当一个查询包含任何并行不安全操作时，并行查询对这个查询是完全被禁用的。

下面的操作总是并行受限：

- 公共表表达式（CTE）的扫描。
- 临时表的扫描。
- 外部表的扫描，除非外部数据包装器有一个IsForeignScanParallelSafe API。
- 对InitPlan或者SubPlan的访问。

4.1. 为函数和聚集加并行标签

规划器无法自动判定一个用户定义的函数或者聚集是并行安全、并行受限还是并行不安全，因为这需要预测函数可能执行的每一个操作。相当于一个停机问题，因此是不可能的。甚至对于可以做到判定的简单函数也不会尝试，因为那会非常昂贵且容易出错。相反，除非是被标记出来，所有用户定义的函数都被认为是并行不安全的。在使用CREATE FUNCTION或者ALTER FUNCTION时，可以通过指定PARALLEL SAFE、PARALLEL RESTRICTED或者PARALLEL UNSAFE来设置标记。使用CREATE AGGREGATE时，PARALLEL选项可以被指定为SAFE、RESTRICTED或者 UNSAFE。

如果函数和聚集写数据库、访问序列、改变事务状态（即便是临时改变，例如建立一个EXCEPTION块来捕捉错误的 PL/SQL）或者对设置做持久化的更改，它们一定要被标记为PARALLEL UNSAFE。类似地，如果函数访问临时表、客户端连接状态、游标、预备语句或者系统无法在工作者之间同步的后端本地状态，它们必须被标记为PARALLEL RESTRICTED。例如，setseed和 random由于后一种原因而是并行受限的。

一般而言，如果一个函数是受限或者不安全的却被标记为安全，或者它实际是不安全的却被标记为受限，把它用在并行查询中时可能会抛出错误或者产生错误的回答。如果 C 语言函数被错误标记，理论上它会展现出完全不明确的行为，因为系统中无法保护自身不受任意 C 代码的影响。但是，即使在最可能的情况下，造成的结果也不会比其他任何函数更糟糕。但如有疑虑，最好还是标记函数为UNSAFE。

如果在并行工作者中执行的函数要求领导者没有持有的锁，例如读该查询中没有引用的表，那么工作者退出时会释放那些锁（而不是在事务结束时释放）。如果写了一个这样做的函数并且这种不同的行为很重要，应当把这类函数标记为PARALLEL RESTRICTED以确保它们只在领导者中执行。

注意查询规划器不会为了获取一个更好的计划而考虑延迟计算并行受限的函数或者聚集。所以，如果一个被应用到特定表的WHERE子句是并行受限的，查询规划器就不会考虑把对那张表的扫描放置在Gather节点之下。在一些情况中，可以把对表的扫描包括在查询的并行部分并且延迟对WHERE子句的计算，这样它会出现在Gather节点之上。但是，规划器通常不会这样做。

数据分区方案

KingbaseES 支持基本的表分区。本文档介绍分区的概念，以及数据库分区的具体实现。

1. 概述

分区是指将逻辑上的一个大表分成物理上一些小的片。分区有很多益处：

- 在某些情况下，能够使得查询性能显著提升。特别是当表中大部分被大量访问的行位于单个分区或少量分区中时。分区替代了索引的前导列，减小了索引的大小，使索引中使用频繁的部分更有可能被放在内存中。
- 当查询或更新访问一个分区的大部分行时，可以通过该分区上的顺序扫描来取代分散到整个表上的索引和随机访问，从而改善性能。
- 如果需求计划使用分区设计，可以通过增加或移除分区来完成批量载入和删除。`ALTER TABLE NO INHERIT`和`DROP TABLE`都远快于一个批量操作。这些命令也完全避免了由批量`DELETE`造成的`VACUUM`负载。
- 使用频率低的数据可以被迁移到廉价但运行缓慢的存储介质上。

当表非常庞大时，分区所带来的好处是非常值得的。一个表何种情况下会从分区获益取决于实际的应用情况，根据经验，当表的尺寸超过了数据库服务器物理内存时，分区将显著提升表查询时的性能。

目前，KingbaseES支持通过表继承来进行分区。每一个分区被创建为父表的一个子表。父表本身通常为空，仅仅为了表示整个数据集。在尝试建立分区之前，应该先熟悉继承（参见*《SQL和PL/SQL速查手册》II. SQL语言基础-“数据定义”-9. 继承*）。

在KingbaseES中可以实现下列形式的分区：

范围分区

表被根据一个关键列或一组列分区为"范围"，不同的分区的范围之间没有重叠。例如，可以根据日期范围分区，或者根据特定业务对象的标识符分区。

列表分区

通过显式地列出每一个分区中出现的键值来分区表。

2. 实现分区

建立一个分区的表步骤如下：

a. 创建"主"表，所有的分区都将继承它。

   这个表将不会包含任何数据。不要在这个表上定义任何检查约束，除非准备将它们应用到所有分区。同样也不需要定义任何索引或者唯一约束。

b. 创建一些继承于主表的"子"表。通常，这些表不会在从主表继承的列集中增加任何列。

   将这些子表认为是分区，尽管看起来类似普通的KingbaseES表（或者可能是外部表）。

c. 为分区表增加表约束以定义每个分区中允许的键值。

   典型的例子：

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

   要确保这些约束能够保证在不同分区所允许的键值之间不存在重叠。设置范围约束时常见的错误是：

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

   这是错误的，因为键值200并没有被清楚地分配到某一个分区。

   范围分区和列表分区在语法上没有区别，这些术语仅仅为了方便描述而存在。

d. 对于每一个分区，在关键列上创建一个索引，并创建其他所需要的索引（关键索引并不是严格必要的，但是在大部分情况下它都是有用的。如果用户希望键值是唯一的，则还要为每一个分区创建一个唯一或者主键约束。）。

e. 还可以有选择地定义一个触发器或者规则将在主表上的数据插入重定向到合适的分区上。

f. 确保在`kingbase.conf`中constraint_exclusion配置参数未被禁用。如果被禁用，查询将不会按照期望的方式被优化。

假设需要为一个大型的冰淇淋公司构建一个数据库，记录每天在每一个区域的最高气温以及冰淇淋销售额。理论上所需要的表为：

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

由于该表的主要用途是为管理层提供在线报告，因此大部分查询将只会访问上周、上月或者上季度的数据。为了减少需要保存的旧数据的量，决定只保留最近3年的数据。在每一个月的开始，删除最老一个月的数据。

在这种情况下，可以使用分区来满足对于测量表的所有不同需求。按照上面的步骤，建立分区：

a. 主表是measurement表，完全按照以上的方式声明。

b. 下一步为每一个活动月创建一个分区：

```
CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 ( ) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
```

每一个分区自身都是完整的表，但是它们的定义都是从measurement表继承而来。

这解决了一个问题：删除旧数据。每个月，所需要做的是在最旧的子表上执行一个DROP TABLE命令并为新一个月的数据创建一个新的子表。

c. 必须提供不重叠的表约束。和前面简单地创建分区表不同，实际的表创建脚本应该是：

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01')
) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01')
) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01')
) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01')
) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01')
) INHERITS (measurement);
```

d. 在关键列上也需要索引：

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (
logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (
logdate);
...
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (
logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (
logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (
logdate);
```

不增加更多的索引。

e. 希望应用能够使用INSERT INTO measurement ...并且数据将被重定向到合适的分区表。可以通过为主表附加一个合适的触发器函数来实现这一点。如果数据只被增加到最后一个分区，可以使用一个简单的触发器函数：

```
CREATE OR REPLACE INTERNAL FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plsql;
```

完成函数创建后，创建一个调用该触发器函数的触发器：

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

必须在每个月重新定义触发器函数，这样才会总指向当前分区。而触发器的定义则不需要被更新。

用户可能希望插入数据时服务器会自动地定位应该加入数据的分区。通过一个更复杂的触发器函数可实现：

```
CREATE OR REPLACE INTERNAL FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the
        measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plsql;
```

触发器的定义和以前一样。注意每一个IF测试必须准确地匹配它的分区的CHECK约束。

当该函数比单月形式更加复杂时，并不需要频繁地更新它，因为可以在需要的时候提前加入分支。

> **注意：**在实践中，大部分插入都会进入最新的分区，最好先检查它。为了简洁，为触发器的检查采用了和本例中其他部分一致的顺序。

一个复杂的分区模式需要大量的DDL。在上面的例子中，需要每月创建一个新分区，所以最好能够编写一个脚本自动地生成所需的DDL。

3. 管理分区

通常当初始定义的表倾向于动态变化时，一组分区会被创建。删除旧的分区并周期性地为新数据增加新分区是很常见的。分区的一个最重要的优点是可以通过操纵分区结构来使得任务自发完成，而不需要去物理移除大量数据。

移除旧数据的最简单的方法是直接删除不再需要的分区：

```
DROP TABLE measurement_y2006m02;
```

上述操作可以快速删除百万级别的数据，因为不需逐一删除数据。

另一个常用方式是将分区从表中移除，但会将它作为一个独立的表保留下来：

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

这允许在数据被删除前执行更进一步的操作。这是一个很有用的通过COPY、sys_dump或类似的工具备份数据的方法。也是进行数据聚集、执行其他数据操作或运行报表的好方法。

与之类似，用户也可以增加新分区来处理新数据。可以在被分区的表中创建一个新的空分区：

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

作为一种可选方案，有时创建一个和分区结构一样的新表更方便，并且在有需要时才将它作为一个合适的分区。这使得这些分区数据写入时不用约束检查：

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- 可能做一些其他数据准备工作
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

4. 分区和约束排除

约束排除是一种查询优化技术，可以提升按照以上方式定义的被分区表的性能。例如：

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

如果没有约束排除，上述查询将扫描measurement表的每一个分区。在启用约束排除后，规划器将检查每一个分区的约束来确定该分区需不需要被扫描，因为分区中可能不包含满足查询WHERE子句的行。如果规划器能够证实这一点，则它会将该分区排除在查询计划之外。

可以使用EXPLAIN命令来显示开启了constraint_exclusion的计划和没有开启该选项的计划之间的区别。一个典型的未优化的计划是：

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE
'2008-01-01';

                                  QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m02 measurement  (cost=0.00..30.38
             rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03 measurement  (cost=0.00..30.38
             rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m12 measurement  (cost=0.00..30.38
             rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38
             rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

其中的某些或者全部分区将会使用索引扫描而不是全表顺序扫描，但关键在于根本不需要扫描旧分区来回答这个查询。当开启约束排除后，同一个查询会得到一个更优计划：

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE
'2008-01-01';
                                  QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38
             rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

注意约束排除只由CHECK约束驱动，而非索引的存在。因此，没有必要在关键列上定义索引。是否在给定分区上定义索引取决于希望查询经常扫描表的大部分还是小部分。在后一种情况中索引将会发挥作用。

constraint_exclusion的默认（也是推荐）设置实际上既不是on也不是off，而是一个被称为partition的中间设置，这使得该技术只被应用于将要在被分区表上工作的查询。设置on将使得规划器在所有的查询中检查CHECK约束，即使简单查询不会从中受益。

5. 可选分区方法

另一种将插入重定向到合适的分区表的方法是在主表上建立规则而不是触发器：

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

一个规则比一个触发器具有明显更高的负荷，但是该负荷是由每个查询承担而不是每一个行，因此这种方法可能对于批量插入的情况有益。但是，在大部分情况下触发器方法能提供更好的性能。

注意COPY会忽略规则。如果希望使用COPY来插入数据，希望将数据复制到正确的分区表而不是主表。COPY会引发触发器，因此如果使用触发器方法就可以正常地使用它。

规则方法的另一个缺点是如果一组规则没有覆盖被插入的数据，则该数据将被插入到主表中而不会发出任何错误。

分区也可以使用UNION ALL视图来组织。例如：

```
CREATE VIEW measurement AS
          SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
```

但若要增加或者删除单独的分区，需重新创建视图。在实践中，相对于继承，该方法很少被推荐使用。

6. 警告

下面的警告适用于分区表：

- 没有自动的方法来验证所有的CHECK约束是互斥的。创建代码来生成分区并创建或修改相关对象比手工写命令要更安全。
- 这里显示的方案假设行中的分区键列永远不会改变，或者至少不会改变到需要它移动到另一个分区的程度。由于CHECK约束的存在，这样尝试的UPDATE会失败。如果需要处理这种情况，可以在分区表上放置适当的UPDATE触发器，但这会使结构的管理更加复杂。
- 如果用户手动使用VACUUM或ANALYZE命令，需在每一个分区上都运行一次。以下的命令：

  ```
  ANALYZE measurement;
  ```

  只会处理主表。
- 带有ON CONFLICT子句的INSERT 语句不太可能按照预期的方式工作，因为ON CONFLICT动作只有在指定的目标关系（而非它的子关系）上存在唯一违背的情况下才会被采用。

下列警告适用于约束排除：

- 只有在查询的WHERE子句包含常量（或者外部提供的参数）时，约束排除才会起效。例如，一个与非不变函数（例如CURRENT_TIMESTAMP）的比较不能被优化，因为规划器不确定该函数的值在运行时会落到哪个分区内。
- 保持分区约束简单，否则规划器可能没有办法验证无需访问的分区。按前面的例子所示，为列表分区使用简单相等条件或者为范围分区使用简单范围测试。一个好的经验法则是分区约束只包含使用B-tree可索引操作符的比较，比较的双方应该是分区列和常量。
- 在约束排除期间，主表所有的分区上的所有约束都会被检查，所以大量的分区将会显著地增加查询规划时间。使用这些技术的分区最多在100个分区的情况下工作良好，需避免使用上千个分区。

Part III. 监控

目录

监控数据库活动

数据库管理员常常想了解系统当前的工作状态。本文档将围绕此主题展开。

KingbaseES中拥有用于监控数据库活动并进行性能分析的工具。本文档侧重于描述KingbaseES统计收集器，以及常规的 Unix 监控程序，如ps、top、iostat和vmstat。另外，每当数据库管理员发现性能差的查询时，需要使用KingbaseES的EXPLAIN命令进行进一步的调查。

1. 标准 Unix 工具

在大部分 Unix 平台上，KingbaseES会修改由ps输出的命令标题，识别出个体服务器进程：

```
$ ps auxww | grep ^kingbase
kingbase  15551  0.0  0.1  57536  7132 pts/0    S    18:02
0:00 kingbase -i
kingbase  15554  0.0  0.0  57536  1184 ?        Ss   18:02
0:00 kingbase: writer process
kingbase  15555  0.0  0.0  57536   916 ?        Ss   18:02
0:00 kingbase: checkpointer process
kingbase  15556  0.0  0.0  57536   916 ?        Ss   18:02
0:00 kingbase: wal writer process
kingbase  15557  0.0  0.0  58504  2244 ?        Ss   18:02
0:00 kingbase: autovacuum launcher process
kingbase  15558  0.0  0.0  17512  1068 ?        Ss   18:02
0:00 kingbase: stats collector process
kingbase  15582  0.0  0.0  58772  3080 ?        Ss   18:04
0:00 kingbase: joe runbug 127.0.0.1 idle
kingbase  15606  0.0  0.0  58772  3052 ?        Ss   18:07
0:00 kingbase: tgl regression [local] SELECT waiting
kingbase  15610  0.0  0.0  58772  3056 ?        Ss   18:07
0:00 kingbase: tgl regression [local] idle in transaction
```

（ps的调用方式随不同的平台而变，但是显示的细节没什么差异。上述例子来自于一个近期的 Linux 系统）。列在这里的第一个进程是主服务器进程。为它显示的命令参数是当它被启动时使用的启动参数。接下来的五个进程是由主进程自动启动的后台工作者进程（如果将系统设置为不启动统计收集器，"统计收集器"进程将不会出现；同样"自动清理发动"进程也可以被禁用）。剩余的进程都是处理客户端连接的服务器进程。每个这种进程都会把它的命令行显示设置为这种形式

```
kingbase: user database host activity
```

在该客户端连接的生命期中，用户、数据库以及（客户端）主机项保持不变，但是活动指示器会改变。活动可以是闲置（即等待一个客户端命令）、在事务中闲置（在一个BEGIN块里等待客户端）或者一个命令类型名，例如SELECT。还有，如果服务器进程正在等待一个其它会话持有的锁，等待中会被追加到上述信息中。在上面的例子中，可以推断：进程 15606 正在等待进程 15610 完成其事务并且因此释放一些锁（进程 15610 必定是阻塞者，因为没有其他活动会话。在更复杂的情况中，可能需要查看sys_locks系统视图来决定谁阻塞了谁）。

如果配置了cluster_name，则集簇的名字将会显示在ps的输出中：

```
$ ksql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
kingbase   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00
kingbase: server1: writer process
...
```

如果update_process_title已被关闭，那么活动指示器将不会被更新，进程标题仅在新进程被启动的时候设置一次。在某些平台上这样做可以为每个命令节省可观的开销，但在其它平台上却不明显。

> **提示:** Solaris需要特别的处理。用户需使用/usr/ucb/ps而不是/bin/ps，使用两个w标志，而非一个。另外，对kingbase命令的最初调用必须用一个比服务器进程提供的短的ps状态显示。如果没有满足全部三个要求，每个服务器进程的ps将输出原始的kingbase命令行。

2. 统计收集器

KingbaseES的统计收集器是一个支持收集和报告服务器活动信息的子系统。目前，这个收集器可以对表和索引的访问计数，计数可以按磁盘块和行数来进行。它还跟踪每个表中的总行数、每个表的清理和分析动作的信息。它也统计调用用户定义函数的次数以及在每次调用中花费的总时间。

KingbaseES也支持报告有关系统正在干什么的动态信息，例如当前正在被其他服务器进程执行的命令以及系统中存在哪些其他连接。这个功能是独立于收集器进程存在的。

2.1. 统计收集配置

因为统计收集给查询执行增加了一些负荷，系统可以被配置为收集或不收集信息。这由配置参数控制，它们通常在kingbase.conf中设置。

参数track_activities允许监控当前被任意服务器进程执行的命令。

参数track_counts控制是否收集关于表和索引访问的统计信息。

参数track_functions启用对用户定义函数使用的跟踪。

参数track_io_timing启用对块读写次数的监控。

通常这些参数被设置在kingbase.conf中，这样它们会应用于所有服务器进程，但也可以在单个会话中使用SET命令打开或关闭它们（为了阻止普通用户对管理员隐藏他们的活动，只有超级用户被允许使用SET来改变这些参数）。

统计收集器通过临时文件将收集到的信息传送给其他KingbaseES进程。这些文件被存储在名字由stats_temp_directory参数指定的目录中，默认是sys_stat_tmp。为了得到更优的性能，stats_temp_directory可以被指向一个基于 RAM 的文件系统来降低物理 I/O 需求。当服务器被干净地关闭时，一份统计数据的永久拷贝被存储在sys_stat子目录中，这样在服务器重启后统计信息能被保持。当在服务器启动执行恢复时（例如立即关闭、服务器崩溃以及时间点恢复之后），所有统计计数器会被重置。

2.2. 查看统计信息

表 2-1中列出了可以用来显示系统当前状态的一些预定义视图。表 2-2中列出了可以显示统计收集结果的另一些视图。也可以使用底层统计函数（在第 2.3 节中讨论）来建立自定义的视图。

在使用统计信息监控收集到的数据时，必须了解这些信息并非是实时更新的。每个独立的服务器进程只在进入闲置状态之前才向收集器传送新的统计计数；因此正在进行的查询或事务并不影响显示出来的总数。同样，收集器本身也最多每间隔一定时间（缺省为 500ms，除非在编译服务器的时候修改过）发送一次新的报告。因此显示的信息总是落后于实际活动。但是由track_activities收集的当前查询信息总是最新的。

另一个重点是当一个服务器进程被要求显示这些统计信息时，它首先取得收集器进程最近发出的报告并且接着为所有统计视图和函数使用这个快照，直到它的当前事务的结尾。因此只要你继续当前事务，统计数据将会一直显示静态信息。相似地，当任何关于所有会话的当前查询的信息在一个事务中第一次被请求时，这样的统计信息将被收集。并且在整个事务期间将显示相同的信息。这是一种特性而非缺陷，因为它允许在该统计信息上执行多个查询并且关联结果而不用担心那些数字会在你不知情的情况下改变。但是如果希望用每个查询都看到新结果，要确保在任何事务块之外做那些查询。或者，可以调用sys_stat_clear_snapshot()，丢弃当前事务的统计快照（若有）。下一次对统计性信息的调用将导致获取一个新的快照。

一个事务也可以在视图sys_stat_xact_all_tables、sys_stat_xact_sys_tables、sys_stat_xact_use sys_stat_xact_user_functions中查询到本身的统计信息（这些信息还没有被传送给收集器）。这些数字并不遵从上述的行为，相反它们在事务期间持续被更新。

表 2-1. 动态统计视图

视图名称	描述
sys_stat_activity	每个服务器进程一行，显示与那个进程的当前活动相关的信息，例如状态和当前查询。详见sys_stat_activity。
sys_stat_replication	每一个 WAL 发送一行进程，显示有关到该发送进程连接的后备服务器的复制的统计信息。详见sys_stat_replication。
sys_stat_wal_receiver	只有一行，显示来自 WAL 接收器所连接服务器的有关该接收器的统计信息。详见sys_stat_wal_receiver。
sys_stat_ssl	每个连接（常规连接和复制连接）一行，显示有关在此连接上使用的 SSL 的信息。详见sys_stat_ssl。

表 2-2. 已收集统计信息的视图

视图名称	描述
sys_stat_archiver	只有一行，显示有关 WAL 归档进程活动的统计信息。详见sys_stat_archiver。
sys_stat_bgwriter	只有一行，显示有关后台写进程的活动的统计信息。详见sys_stat_bgwriter。
sys_stat_database	每个数据库一行，显示数据库范围的统计信息。详见sys_stat_database。
sys_stat_database_conflicts	每个数据库一行，显示数据库范围的统计信息，这些信息的内容是关于由于与后备服务器的恢复过程发生冲突而被取消的查询。详见sys_stat_database_conflicts。
sys_stat_all_tables	当前数据库中每个表一行，显示有关访问指定表的统计信息。详见sys_stat_all_tables。
sys_stat_sys_tables	和sys_stat_all_tables一样，但只显示系统表。
sys_stat_user_tables	和sys_stat_all_tables一样，但只显示用户表。
sys_stat_xact_all_tables	和sys_stat_all_tables相似，但计数动作只在当前事务内发生（还没有被包括在sys_stat_all_tables和相关视图中）。用于生存和死亡行数量的列以及清理和分析动作在此视图中不出现。

视图名称	描述
sys_stat_xact_sys_tables	和sys_stat_xact_all_tables一样，但只显示系统表。
sys_stat_xact_user_tables	和sys_stat_xact_all_tables一样，但只显示用户表。
sys_stat_all_indexes	当前数据库中的每个索引一行，显示：表OID、索引OID、模式名、表名、索引名、 使用了该索引的索引扫描总数、索引扫描返回的索引记录数、使用该索引的简单索引扫描抓取的活表(livetable)中数据行数。当前数据库中的每个索引一行，显示与访问指定索引有关的统计信息。详见sys_stat_all_indexes。
sys_stat_sys_indexes	和sys_stat_all_indexes一样，但只显示系统表上的索引。
sys_stat_user_indexes	和sys_stat_all_indexes一样，但只显示用户表上的索引。
sys_statio_all_tables	当前数据库中每个表一行(包括TOAST表)，显示：表OID、模式名、表名、 从该表中读取的磁盘块总数、缓冲区命中次数、该表上所有索引的磁盘块读取总数、该表上所有索引的缓冲区命中总数、在该表的辅助TOAST表(如果存在)上的磁盘块读取总数、 在该表的辅助TOAST表(如果存在)上的缓冲区命中总数、TOAST表的索引的磁盘块读 取总数、TOAST表的索引的缓冲区命中总数。当前数据库中的每个表一行，显示有关在指定表上 I/O 的统计信息。详见sys_statio_all_tables。
sys_statio_sys_tables	和sys_statio_all_tables一样，但只显示系统表。
sys_statio_user_tables	和sys_statio_all_tables一样，但只显示用户表。
sys_statio_all_indexes	当前数据库中每个索引一行，显示：表OID、索引OID、模式名、 表名、索引名、该索引的磁盘块读取总数、该索引的缓冲区命中总数。当前数据库中的每个索引一行，显示与指定索引上的 I/O 有关的统计信息。详见sys_statio_all_indexes。
sys_statio_sys_indexes	和sys_statio_all_indexes一样，但只显示系统表上的索引。

视图名称	描述
sys_statio_user_indexes	和sys_statio_all_indexes一样，但只显示用户表上的索引。
sys_statio_all_sequences	当前数据库中每个序列对象一行，显示：序列OID、模式名、序列名、序列的磁盘读取总数、序列的缓冲区命中总数。当前数据库中的每个序列一行，显示与指定序列上的 I/O 有关的统计信息。详见sys_statio_all_sequences。
sys_statio_sys_sequences	和sys_statio_all_sequences一样，但只显示系统序列（目前没有定义系统序列，因此这个视图总是为空）。
sys_statio_user_sequences	和sys_statio_all_sequences一样，但只显示用户序列。
sys_stat_user_functions	对于所有跟踪功能，函数的OID，模式，名称，数量 通话总时间，和自我的时间。自我时间是 在函数本身所花费的时间量，总时间包括 它调用函数所花费的时间。时间值以毫秒为单位。每一个被跟踪的函数一行，显示与执行该函数有关的统计信息。详见sys_stat_user_functions。
sys_stat_xact_user_functions	和sys_stat_user_functions相似，但是只统计在当前事务期间的调用（还没有被包括在sys_stat_user_functions中）。
sys_stat_progress_vacuum	每个运行VACUUM的后端（包括自动清理工作者进程）一行，显示当前的进度。

针对每个索引的统计信息对于判断哪个索引正被使用以及它们的效果特别有用。

sys_statio_系列视图主要用于判断缓冲区的效果。当实际磁盘读取数远小于缓冲区命中时，这个缓冲能满足大部分读请求而无需进行内核调用。但是，这些统计信息并没有给出所有的内容：由于KingbaseES处理磁盘 I/O 的方式，不在KingbaseES缓冲区中的数据库仍然驻留在内核的 I/O 缓存中，并且因此可以被再次读取而不需要物理磁盘读取。建议希望了解KingbaseES I/O 行为更多细节的用户将KingbaseES统计收集器和操作系统中允许观察内核处理 I/O 的工具一起使用。

表 2-3. sys_stat_activity 视图

列	类型	描述
datid	oid	这个后端连接到的数据库的OID

列	类型	描述
datname	name	这个后端连接到的数据库的名称
pid	integer	这个后端的进程 ID
usesysid	oid	登录到这个后端的用户的 OID
usename	name	登录到这个后端的用户的名称
application_name	text	连接到这个后端的应用的名称
client_addr	inet	连接到这个后端的客户端的 IP 地址。如果这个域为空，它表示客户端通过服务器机器上的一个 Unix 套接字连接或者这是一个内部进程（如自动清理）。
client_hostname	text	已连接的客户端的主机名，由client_addr的反向 DNS 查找报告。这个域将只对 IP 连接非空，并且只有log_hostname被启用时才会非空。
client_port	integer	客户端用以和这个后端通信的 TCP 端口号，如果使用 Unix 套接字则为-1
backend_start	timestamp with time zone	这个进程被启动的时间，即客户端是什么时候连接到服务器的
xact_start	timestamp with time zone	这个进程的当前事务被启动的时间，如果没有活动事务则为空。如果当前查询是它的第一个事务，这一列等于query_start。
query_start	timestamp with time zone	当前活动查询被开始的时间，如果state不是active，这个域为上一个查询被开始的时间
state_change	timestamp with time zone	state上一次被改变的时间

列	类型	描述
wait_event_type	text	后端正在等待的事件类型，如果不存在则为 NULL。可能的值有： • LWLockNamed：后端正在等待一个特定命名的轻量级锁。每一个这样的锁保护共享内存中的一个特定数据结构。wait_event将包含该轻量级锁的名称。 • LWLockTranche：后端正在等待一组相关轻量级锁中的一个。该组中的所有锁都执行一种相似的功能。wait_event将标识这个组中锁的大体目的。 • Lock：后端正在等待一个重量级锁。重量级锁，也称为锁管理器锁或者简单锁，主要保护 SQL 可见的对象，例如表。但是，它们也被用于确保特定内部操作的互斥，例如关系扩展。wait_event将标识等待的锁的类型。 • BufferPin：服务器进程正在等待访问一个数据缓冲区，而此时没有其他进程正在检查该缓冲区。如果另一个进程持有一个最终从要访问的缓冲区中读取数据的打开的游标，缓冲区 pin 等待可能会被拖延。
wait_event	text	如果后端当前正在等待，则是等待事件的名称，否则为 NULL。详见表 2-4。
state	text	这个后端的当前总体状态。可能的值是： • active：后端正在执行一个查询。 • idle：后端正在等待一个新的客户端命令。 • idle in transaction：后端在一个事务中，但是当前没有正在执行一个查询。 • idle in transaction (aborted)：这个状态与idle in transaction相似，不过在该事务中的一个语句导致了一个错误。 • fastpath function call：后端正在执行一个 fast-path 函数。 • disabled：如果在这个后端中track_activities被禁用，则报告这个状态。
backend_xid	xid	这个后端的顶层事务标识符（如果存在）。

列	类型	描述
backend_xmin	xid	当前后端的xmin范围。
query	text	这个后端最近查询的文本。如果state为active，这个域显示当前正在执行的查询。在所有其他状态下，它显示上一个被执行的查询。

sys_stat_activity视图将为每一个服务器进程有一行，显示与该进程的当前活动相关的信息。

> **注意：**wait_event和state列是独立的。如果一个后端处于active状态，它可能是也可能不是某个事件上的waiting。如果状态是active并且wait_event为非空，它意味着一个查询正在被执行，但是它被阻塞在系统中某处。

表 2-4. wait_event 描述

等待事件类型	等待事件名称	描述
	ShmemIndexLock	正等待在共享内存中查找或者分配空间。
	OidGenLock	正等待分配或者赋予一个 OID。
	XidGenLock	正等待分配或者赋予一个事务 ID。
	ProcArrayLock	正等待在事务结尾得到一个快照或者清除事务 ID。
	SInvalReadLock	正等待从共享无效消息队列中检索或者移除消息。
	SInvalWriteLock	正等待在共享无效消息队列中增加一个消息。
	WALBufMappingLock	正等待在 WAL 缓冲区中替换一个页面。
	WALWriteLock	正等待 WAL 缓冲区被写入到磁盘。
	ControlFileLock	正等待读取或者更新控制文件或创建一个新的 WAL 文件。
	CheckpointLock	正等待执行检查点。
	CLogControlLock	正等待读取或者更新事务状态。

等待事件类型	等待事件名称	描述
LWLockNamed	SubtransControlLock	正等待读取或者更新子事务信息。
	MultiXactGenLock	正等待读取或者更新共享多事务状态。
	MultiXactOffsetControlLock	正等待读取或者更新多事务偏移映射。
	MultiXactMemberControlLock	正等待读取或者更新多事务成员映射。
	RelCacheInitLock	正等待读取或者写入关系缓冲区初始化文件。
	CheckpointerCommLock	正等待管理 fsync 请求。
	TwoPhaseStateLock	正等待读取或者更新预备事务的状态。
	TablespaceCreateLock	正等待创建或者删除表空间。
	BtreeVacuumLock	正等待读取或者更新一个 B-树索引的 vacuum 相关的信息。
	AddinShmemInitLock	正等待管理共享内存中的空间分配。
	AutovacuumLock	自动清理工作者或者启动器正等待更新或者读取自动清理工作者的当前状态。
	AutovacuumScheduleLock	正等待确认选中进行清理的表仍需要清理。
	SyncScanLock	正等待为同步扫描得到一个表上扫描的开始位置。
	RelationMappingLock	正等待更新用来存储目录到文件节点映射的关系映射文件。
	AsyncCtlLock	正等待读取或者更新共享通知状态。
	AsyncQueueLock	正等待读取或者更新通知消息。
	SerializableXactHashLock	正等待检索或者存储有关可序列化事务的信息。

等待事件类型	等待事件名称	描述
	SerializableFinishedListLock	正等待访问已结束可序列化事务的列表。
	SerializablePredicateLockListLock	正等待在由可序列化事务持有的所列表上执行一个操作。
	OldSerXidLock	正等待读取或者记录冲突的可序列化事务。
	SyncRepLock	正等待读取或者更新有关同步复制的信息。
	BackgroundWorkerLock	正等待读取或者更新后台工作者状态。
	DynamicSharedMemoryControlLock	正等待读取或者更新动态共享内存状态。
	AutoFileLock	正等待更新kingbase.auto.conf文件。
	ReplicationSlotAllocationLock	正等待分配或者释放一个复制槽。
	ReplicationSlotControlLock	正等待读取或者更新复制槽状态。
	CommitTsControlLock	正等待读取或者更新事务提交时间戳。
	CommitTsLock	正等待读取或者更新事务时间戳的最新设置值。
	ReplicationOriginLock	正等待设置、删除或者使用复制源头。
	MultiXactTruncationLock	正等待读取或者阶段多事务信息。
	OldSnapshotTimeMapLock	正等待读取或者更新旧的快照控制信息。
	clog	正等待一个 clog（事务状态）缓冲区上的 I/O。
	commit_timestamp	正等待提交时间戳缓冲区上的 I/O。

等待事件类型	等待事件名称	描述
LWLockTranche	subtrans	正等待子事务缓冲区上的 I/O。
	multixact_offset	正等待多事务偏移缓冲区上的 I/O。
	multixact_member	正等待多事务成员缓冲区上的 I/O。
	async	正等待 async（通知）缓冲区上的 I/O。
	oldserxid	正等待 oldserxid 缓冲区上的 I/O。
	wal_insert	正等待把 WAL 插入到一个内存缓冲区。
	buffer_content	正等待读取或者写入内存中的一个数据页。
	buffer_io	正等待一个数据页面上的 I/O。
	replication_origin	正等待读取或者更新复制进度。
	replication_slot_io	正等待一个复制槽上的 I/O。
	proc	正等待读取或者更新 fast-path 锁信息。
	buffer_mapping	正等待把一个数据块与缓冲池中的一个缓冲区关联。
	lock_manager	正等待增加或者检查用于后端的锁，或者正等待加入或者退出一个锁定组（并行查询使用）。
	predicate_lock_manager	正等待增加或者检查谓词锁信息。
	relation	正等待获得一个关系上的锁。
	extend	正等待扩展一个关系。
	page	正等待获得一个关系上的页面的锁。
	tuple	正等待获得一个元组上的锁。

等待事件类型	等待事件名称	描述
Lock	transactionid	正等待一个事务结束。
	virtualxid	正等待获得一个虚拟 xid 锁。
	speculative token	正等待获取一个 speculative insertion lock。
	object	正等待获得一个非关系数据库对象上的锁。
	userlock	正等待获得一个用户锁。
	advisory	正等待获得一个咨询用户锁。
BufferPin	BufferPin	正等待在一个缓冲区上加 pin。

> **注意:** 对于扩展安装的切片（tranche），这个名称由扩展指定并且会被wait_event显示出来。很有可能在其他后端不知道的情况下，用户在其中一个后端中注册了切片（通过在动态共享内存中分配），那么此这种情况会显示extension。

下面的例子展示了如何查看等待事件

```
SELECT pid, wait_event_type, wait_event FROM sys_stat_activity WHERE
wait_event is NOT NULL;
 pid  | wait_event_type |  wait_event
------+-----------------+---------------
 2540 | Lock            | relation
 6644 | LWLockNamed     | ProcArrayLock
(2 rows)
```

表 2-5. sys_stat_replication 视图

列	类型	描述
pid	integer	一个 WAL 发送进程的进程 ID
usesysid	oid	登录到这个 WAL 发送进程的用户的 OID
usename	name	登录到这个 WAL 发送进程的用户的名称
application_name	text	连接到这个 WAL 发送进程的应用的名称
client_addr	inet	连接到这个 WAL 发送进程的客户端的 IP 地址。如果这个域为空，它表示该客户端通过服务器机器上的一个Unix 套接字连接。
client_hostname	text	连接上的客户端的主机名，由一次对client_addr的逆向 DNS 查找报告。这个域将只对 IP 连接非空，并且只有在log_hostname被启用时非空
client_port	integer	客户端用来与这个 WAL 发送进程通讯的 TCP 端口号，如果使用 Unix 套接字则为-1
backend_start	timestamp with time zone	这个进程开始的时间，即客户端是何时连接到这个WAL 发送进程的
backend_xmin	xid	由hot_standby_feedback报告的这个后备机的xmin水平线。
state	text	当前的 WAL 发送进程状态
sent_location	sys_lsn	在这个连接上发送的最后一个事务日志的位置
write_location	sys_lsn	被这个后备服务器写入到磁盘的最后一个事务日志的位置
flush_location	sys_lsn	被这个后备服务器刷入到磁盘的最后一个事务日志的位置
replay_location	sys_lsn	被重放到这个后备服务器上的数据库中的最后一个事务日志的位置
sync_priority	integer	这个后备服务器被选中为同步后备服务器的优先级
sync_state	text	这个后备服务器的同步状态

sys_stat_replication视图中将为每一个 WAL 发送进程包含一行，用来显示与该发送进程连接的后备服务器的复制统计信息。这个视图中只会列出直接连接的后备机，下游后备服务器的信息不包含在此。

表 2-6 sys_stat_wal_receiver 视图

列	类型	描述
pid	integer	WAL 接收器进程的进程 ID
status	text	WAL 接收器进程的活动状态
receive_start_lsn	sys_lsn	WAL 接收器启动时使用的第一个事务日志位置
receive_start_tli	integer	WAL 接收器启动时使用的第一个时间线编号
received_lsn	sys_lsn	已经接收到并且已经被杀入磁盘的最后一个事务日志的位置，这个域的初始值是 WAL 接收器启动时使用的第一个日志位置
received_tli	integer	已经接收到并且已经被杀入磁盘的最后一个事务日志的时间线编号，这个域的初始值是 WAL 接收器启动时使用的第一个日志所在的时间线编号
last_msg_send_time	timestamp with time zone	从源头 WAL 发送器接收到的最后一个消息的发送时间
last_msg_receipt_time	timestamp with time zone	从源头 WAL 发送器接收到的最后一个消息的接收时间
latest_end_lsn	sys_lsn	报告给源头 WAL 发送器的最后一个事务日志位置
latest_end_time	timestamp with time zone	报告给源头 WAL 发送器最后一个事务日志位置的时间
slot_name	text	这个 WAL 接收器使用的复制槽的名称
conninfo	text	这个 WAL 接收器使用的连接串，安全相关的域会被隐去。

sys_stat_wal_receiver事务只包含一行，它显示了从 WAL 接收器所连接的服务器得到的有关该接收器的统计信息。

表 2-7. sys_stat_ssl 视图

列	类型	描述
pid	integer	一个后端或者 WAL 发送进程的进程 ID
ssl	boolean	如果在这个连接上使用了 SSL 则为真
version	text	在用的 SSL 版本，如果这个连接上没有使用 SSL 则为 NULL
cipher	text	在用的 SSL 密码的名称，如果这个连接上没有使用 SSL 则为 NULL
bits	integer	使用的加密算法中的位数，如果这个连接上没有使用 SSL 则为 NULL
compression	boolean	如果使用了 SSL 压缩则为真，否则为假，如果这个连接上没有使用 SSL 则为 NULL
clientdn	text	来自所使用的客户端证书的识别名（DN）域，如果没有提供客户端证书或者这个连接上没有使用 SSL 则为 NULL。如果 DN 域长度超过NAMEDATALEN（标准编译中是 64 个字符），则它会被截断。

sys_stat_ssl视图将为每一个后端或者 WAL 发送进程包含一行，用来显示这个连接上的 SSL 使用情况。可以把它与sys_stat_activity或者sys_stat_replication通过pid列连接来得到更多有关该连接的细节。

表 2-8. sys_stat_archiver视图

列	类型	描述
archived_count	bigint	已被成功归档的 WAL 文件数量
last_archived_wal	text	最后一个被成功归档的 WAL 文件名称
last_archived_time	timestamp with time zone	最后一次成功归档操作的时间
failed_count	bigint	失败的归档 WAL 文件尝试的数量
last_failed_wal	text	最后一次失败的归档操作的 WAL 文件名称
last_failed_time	timestamp with time zone	最后一次失败的归档操作的时间
stats_reset	timestamp with time zone	这些统计信息最后一次被重置的时间

sys_stat_archiver视图将总是一个单一的行，该行包含着有关集簇的归档进程的数据。

表 2-9. sys_stat_bgwriter视图

列	类型	描述
checkpoints_timed	bigint	已经被执行的计划中检查点的数量
checkpoints_req	bigint	已经被执行的请求检查点的数量
checkpoint_write_time	double precision	在文件被写入磁盘的检查点处理部分花费的总时间，以毫秒计
checkpoint_sync_time	double precision	在文件被同步到磁盘中的检查点处理部分花费的总时间，以毫秒计
buffers_checkpoint	bigint	在检查点期间被写的缓冲区数目
buffers_clean	bigint	被后台写进程写的缓冲区数目
maxwritten_clean	bigint	后台写进程由于已经写了太多缓冲区而停止清洁扫描的次数
buffers_backend	bigint	被一个后端直接写的缓冲区数量
buffers_backend_fsync	bigint	一个后端不得不自己执行fsync调用的次数（通常即使后端自己进行写操作，后台写进程也会处理这些）
buffers_alloc	bigint	被分配的缓冲区数量
stats_reset	timestamp with time zone	这些统计信息上次被重置的时间

sys_stat_bgwriter视图将总是只有单独的一行，它包含集簇的全局数据。

表 2-10. sys_stat_database视图

列	类型	描述
datid	oid	一个数据库的 OID
datname	name	这个数据库的名称
numbackends	integer	当前连接到这个数据库的后端数量。这是在这个视图中唯一一个返回反映当前状态值的列。所有其他列返回从上次重置以来积累的值。
xact_commit	bigint	在这个数据库中已经被提交的事务的数量
xact_rollback	bigint	在这个数据库中已经被回滚的事务的数量
blks_read	bigint	在这个数据库中被读取的磁盘块的数量

列	类型	描述
blks_hit	bigint	磁盘块被发现已经在缓冲区中的次数，这样不需要一次读取（这只包括 KingbaseES 缓冲区中的命中，而不包括在操作系统文件系统缓冲区中的命中）
tup_returned	bigint	在这个数据库中被查询返回的行数
tup_fetched	bigint	在这个数据库中被查询取出的行数
tup_inserted	bigint	在这个数据库中被查询插入的行数
tup_updated	bigint	在这个数据库中被查询更新的行数
tup_deleted	bigint	在这个数据库中被查询删除的行数
conflicts	bigint	由于与恢复冲突而在这个数据库中被取消的查询的数目（冲突只发生在后备服务器上，详见sys_stat_database_conflicts）。
temp_files	bigint	在数据库中被查询创建的临时文件的数量。所有临时文件都被统计，不管为什么创建这些临时文件（如排序或哈希），并且不管log_temp_files设置。
temp_bytes	bigint	在数据库中被查询写到临时文件中的数据总量。所有临时文件都被统计，不管为什么创建这些临时文件（如排序或哈希），并且不管log_temp_files设置。
deadlocks	bigint	在数据库中被检测到的死锁数
blk_read_time	double precision	在数据库中后端花费在读取数据文件块的时间，以毫秒计
blk_write_time	double precision	在这个数据库中后端花费在写数据文件块的时间，以毫秒计
stats_reset	timestamp with time zone	统计信息前一次被重置的时间

sys_stat_database视图将为集簇中的每一个数据库包含一行，每一行显示数据库范围的统计信息。

表 2-11. sys_stat_database_conflicts 视图

列	类型	描述
datid	oid	一个数据库的 OID
datname	name	这个数据库的名称
confl_tablespace	bigint	这个数据库中由于表空间被删掉而取消的查询数量
confl_lock	bigint	这个数据库中由于锁超时而取消的查询数量
confl_snapshot	bigint	这个数据库中由于旧快照而取消的查询数量
confl_bufferpin	bigint	这个数据库中由于被占用的缓冲区而取消的查询数量
confl_deadlock	bigint	这个数据库中由于死锁而取消的查询数量

sys_stat_database_conflicts视图为每一个数据库包含一行，用来显示数据库范围内由于与后备服务器上的恢复过程冲突而被取消的查询的统计信息。这个视图将只包含后备服务器上的信息，因为冲突会不发生在主服务器上。

表 2-12. sys_stat_all_tables视图

列	类型	描述
relid	oid	一个表的 OID
schemaname	name	这个表所在的模式的名称
relname	name	这个表的名称
seq_scan	bigint	在这个表上发起的顺序扫描的次数
seq_tup_read	bigint	被顺序扫描取得的活着的行的数量
idx_scan	bigint	在这个表上发起的索引扫描的次数
idx_tup_fetch	bigint	被索引扫描取得的活着的行的数量
n_tup_ins	bigint	被插入的行数
n_tup_upd	bigint	被更新的行数（包括 HOT 更新的行）
n_tup_del	bigint	被删除的行数
n_tup_hot_upd	bigint	被更新的 HOT 行数（即不要求独立索引更新的行更新）
n_live_tup	bigint	活着的行的估计数量
n_dead_tup	bigint	死亡行的估计数量
n_mod_since_analyze	bigint	从这个表最后一次被分析后备修改的行的估计数量

列	类型	描述
last_vacuum	timestamp with time zone	前一次这个表被手动清理的时间（不统计VACUUM FULL）
last_autovacuum	timestamp with time zone	前一次这个表被自动清理守护进程清理的时间
last_analyze	timestamp with time zone	前一次这个表被手动分析的时间
last_autoanalyze	timestamp with time zone	前一次这个表被自动清理守护进程分析的时间
vacuum_count	bigint	这个表已被手工清理的次数（不统计VACUUM FULL）
autovacuum_count	bigint	这个表已被自动清理守护进程清理的次数
analyze_count	bigint	这个表已被手工分析的次数
autoanalyze_count	bigint	这个表已被自动清理守护进程分析的次数

sys_stat_all_tables视图将为当前数据库中的每一个表（包括 TOAST 表）包含一行，该行显示与对该表的访问相关的统计信息。sys_stat_user_tables和sys_stat_sys_tables视图包含相同的信息，但是通过过滤分别只显示用户和系统表。

表 2-13. sys_stat_all_indexes视图

列	类型	描述
relid	oid	这个索引的基表的 OID
indexrelid	oid	这个索引的 OID
schemaname	name	这个索引所在的模式的名称
relname	name	这个索引的基表的名称
indexrelname	name	这个索引的名称
idx_scan	bigint	在这个索引上发起的索引扫描次数
idx_tup_read	bigint	在这个索引上由扫描返回的索引项数量
idx_tup_fetch	bigint	被使用这个索引的简单索引扫描取得的活着的表行数量

sys_stat_all_indexes视图将为当前数据库中的每个索引包含一行，该行显示关于对该索引访问的统计信息。sys_stat_user_indexes和sys_stat_sys_indexes视图包含相同的信息，但是通过过滤分别显示用户和系统索引。

索引可以被简单索引扫描、"位图"索引扫描以及优化器使用。在一次位图扫描中，多个索引的输出可以被通过 AND 或 OR 规则组合，因此当使用一次位图扫描时难以将取得的个体堆行与特定的索引关联起来。因此，一次位图扫描会增加它使用的索引的sys_stat_all_indexes.idx_tup_read计数，并且为每个表增加sys_stat_all_tables.idx_tup_fetch计数，但是它不影响sys_stat_all_indexes.idx_tup_fetch。如果所提供的常量值不在优化器统计信息记录的范围之内，优化器也会访问索引来检查，因为优化器统计信息可能已经“不新鲜”了。

> **注意：**即便不用位图扫描，idx_tup_read和idx_tup_fetch计数也可能不同，因为idx_tup_read统计从该索引取得的索引项，而idx_tup_fetch统计从表取得的活着的行。如果使用该索引取得了任何死亡行或还未提交的行，或者通过只用索引扫描的方式避免了任何堆获取，后者将比较小。

表 2-14. sys_statio_all_tables视图

列	类型	描述
relid	oid	一个表的 OID
schemaname	name	这个表所在的模式的名称
relname	name	这个表的名称
heap_blks_read	bigint	从这个表读取的磁盘块数量
heap_blks_hit	bigint	在这个表中的缓冲区命中数量
idx_blks_read	bigint	从这个表上所有索引中读取的磁盘块数
idx_blks_hit	bigint	在这个表上的所有索引中的缓冲区命中数量
toast_blks_read	bigint	从这个表的 TOAST 表（如果有）读取的磁盘块数
toast_blks_hit	bigint	在这个表的 TOAST 表（如果有）中的缓冲区命中数量
tidx_blks_read	bigint	从这个表的 TOAST 表索引（如果有）中读取的磁盘块数
tidx_blks_hit	bigint	在这个表的 TOAST 表索引（如果有）中的缓冲区命中数量

sys_statio_all_tables视图将为当前数据库中的每个表（包括 TOAST 表）包含一行，该行显示指定表上有关 I/O 的统计信息。sys_statio_user_tables和sys_statio_sys_tables视图包含相同的信息，但是通过过滤分别只显示用户表和系统表。

表 2-15. sys_statio_all_indexes视图

列	类型	描述
relid	oid	这个索引的基表的 OID
indexrelid	oid	这个索引的 OID
schemaname	name	这个索引所在的模式的名称
relname	name	这个索引的基表的名称
indexrelname	name	这个索引的名称
idx_blks_read	bigint	从这个索引读取的磁盘块数
idx_blks_hit	bigint	在这个索引中的缓冲区命中数量

sys_statio_all_indexes视图将为当前数据库中的每个索引包含一行，该行显示指定索引上有关 I/O 的统计信息。sys_statio_user_indexes和sys_statio_sys_indexes视图包含相同的信息，但通过过滤分别只显示用户索引和系统索引。

表 2-16 sys_statio_all_sequences视图

列	类型	描述
relid	oid	一个序列的 OID
schemaname	name	这个序列所在的模式的名称
relname	name	这个序列的名称
blks_read	bigint	从这个序列中读取的磁盘块数
blks_hit	bigint	在这个序列中的缓冲区命中数量

sys_statio_all_sequences视图将为当前数据库中的每个序列包含一行，该行显示在指定序列上有关 I/O 的统计信息。

表 2-17. sys_stat_user_functions视图

列	类型	描述
funcid	oid	一个函数的 OID
schemaname	name	这个函数所在的模式的名称
funcname	name	这个函数的名称
calls	bigint	这个函数已经被调用的次数
total_time	double precision	在这个函数以及它所调用的其他函数中花费的总时间，以毫秒计
self_time	double precision	在这个函数本身花费的总时间，不包括被它调用的其他函数，以毫秒计

列	类型	描述
sys_stat_user_functions视图将为每一个被追踪的函数包含一行，该行显示有关该函数执行的统计信息。track_functions参数控制被跟踪的函数。		

2.3. 统计函数

其他查看统计信息的方法是直接使用查询，这些查询使用上述标准视图所用的底层统计信息访问函数。如需了解如函数名等细节，可参考标准视图的定义（例如，在ksql中可发出\d+ sys_stat_activity）。针对每一个数据库统计信息的访问函数把一个数据库 OID 作为参数来标识要报告的数据库。而针对每个表和每个索引的函数要求表或索引 OID。针对每个函数统计信息的函数用一个函数 OID。注意只有在当前数据库中的表、索引和函数才能被这些函数看到。

与统计收集相关的额外函数被列举在表 2-18中。

表 2-18. 额外统计函数

函数	返回类型	描述
sys_backend_pid()	integer	处理当前会话的服务器进程的进程 ID
sys_stat_get_activity(integer)	setof record	返回具有指定 PID 的后端相关的一个记录，或者在指定NULL的情况下为系统中每一个活动后端返回一个记录。被返回的域是sys_stat_activity视图中的那些域的一个子集。
sys_stat_get_snapshot_timestamp()	带时区的时间戳	返回当前统计信息快照的时间戳
sys_stat_clear_snapshot()	void	抛弃当前的统计快照
sys_stat_reset()	void	把用于当前数据库的所有统计计数器重置为零（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）。
sys_stat_reset_shared(text)	void	把某些集簇范围的统计计数器重置为零，具体哪些取决于参数（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）。调用sys_stat_reset_shared('bgwriter')把sys_stat_bgwriter视图中显示的所有计数器清零。调用sys_stat_reset_shared('archiver')将会把sys_stat_archiver视图中展示的所有计数器清零。
sys_stat_reset_single_table_counters(oid)	void	把当前数据库中用于单个表或索引的统计数据重置为零（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）
sys_stat_reset_single_function_counters(oid)	void	把当前数据库中用于单个函数的统计信息重置为零（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）

sys_stat_get_activity是sys_stat_activity视图的底层函数，它返回一个行集合，其中包含有关每个后端进程所有可用的信息。有时只获得该信息的一个子集可能会更方便。在那些情况中，可以使用一组更老的针对每个后端的统计访问函数，这些函数显示在表 2-19中。这些访问函数使用一个后端 ID 号，范围从 1 到当前活动后端数目。函数sys_stat_get_backend_idset提供了一种方便的方法为每个活动后端产生一行来调用这

些函数。例如，要显示PID以及所有后端当前的查询：

```
SELECT sys_stat_get_backend_pid(s.backendid) AS pid,
       sys_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT sys_stat_get_backend_idset() AS backendid) AS s;
```

表 2-19. 针对每个后端的统计函数

函数	返回类型	描述
sys_stat_get_backend_idset()	setof integer	当前活动后端 ID 号的集合（从 1 到活动后端数目）
sys_stat_get_backend_activity(integer)	text	这个后端最近查询的文本
sys_stat_get_backend_activity_start(integer)	timestamp with time zone	最近查询被开始的时间
sys_stat_get_backend_client_addr(integer)	inet	该客户端连接到这个后端的 IP 地址
sys_stat_get_backend_client_port(integer)	integer	该客户端用来通信的 TCP 端口号
sys_stat_get_backend_dbid(integer)	oid	这个后端连接到的数据库的 OID
sys_stat_get_backend_pid(integer)	integer	这个后端的进程 ID
sys_stat_get_backend_start(integer)	timestamp with time zone	这个进程被开始的时间
sys_stat_get_backend_userid(integer)	oid	登录到这个后端的用户的 OID
sys_stat_get_backend_wait_event_type(integer)	text	如果后端正在等待，则是等待事件类型的名称，否则为 NULL。详见表 2-4。

函数	返回类型	描述
sys_stat_get_backend_wait_event(integer)	text	如果后端正在等待，则是等待事件的名称，否则为 NULL。详见表 2-4。
sys_stat_get_backend_xact_start(integer)	timestamp with time zone	当前事务被开始的时间

3. 查看锁

sys_locks系统表是监控数据库活动的另一个有用工具。该方式允许数据库管理员查看在锁管理器里面未解决的锁的信息。例如，这个功能可以被用于：

- 查看当前所有未解决的锁、在一个特定数据库中的关系上所有的锁、在一个特定关系上所有的锁，或者由一个特定KingbaseES会话持有的所有的锁。
- 判断当前数据库中带有最多未授予锁的关系（它很可能是数据库客户端的竞争源）。
- 判断锁竞争给数据库总体性能带来的影响，以及锁竞争随着整个数据库流量的变化范围。

sys_locks视图的细节在sys_locks中，可在其中查看更多有关KingbaseES锁和管理并发性的信息。

4. 执行报告

KingbaseES 的某些命令执行时间较长，这时数据库服务器可以在命令执行过程中报告执行进度，目前，可以报告执行进度的命令为 VACUUM。

4.1. VACUUM 执行进度报告

在 VACUUM 运行时，视图 sys_stat_progress_vacuum 中包含了所有执行vacuum的后台服务进程的情况。下面的表格详细解释了这个视图的定义以及每一个字段的含义。注意，目前这个视图中并不包含执行 VACUUM FULL 的后台服务进程。

表 4-1. sys_stat_progress_vacuum 视图

字段	类型	描述
pid	integer	进程的pid。
datid	oid	数据库的OID。
datname	name	数据库名称。
relid	oid	正在执行VACUUM的表的OID。
phase	text	当前处理的阶段。 参见 表 4-2。
heap_blks_total	bigint	VACUUM开始执行时表的页面的总和，若之后执行的命令导致表的页面数增加，本次 VACUUM并不处理。
heap_blks_scanned	bigint	目前已经扫描的页面数，这个数字最终会等于 heap_blks_total。
heap_blks_vacuumed	bigint	已经vacuum处理完成的页面数。
index_vacuum_count	bigint	索引完成的vacuum次数。
max_dead_tuples	bigint	根据 maintenance_work_mem的设置，可以在内存中保存的最多需要被vacuum的元祖个数，同时不引发新的index的vacuum。
num_dead_tuples	bigint	自动上一次index vacuum后已经收集的可以被vacuum的元祖个数。

表 4-2. VACUUM 的阶段

阶段	描述
初始化	VACUUM正在准备开始扫描堆。这个阶段应该很简短。
扫描堆	VACUUM正在扫描堆。如果需要，它将会对每个页面进行修建以及碎片整理，并且可能会执行冻结动作。heap_blks_scanned列可以用来监控扫描的进度。
清理索引	VACUUM当前正在清理索引。如果一个表拥有索引，那么每次清理时这个阶段会在堆扫描完成后至少发生一次。如果maintenance_work_mem不足以存放找到的死亡元组，则每次清理时会多次清理索引。
清理堆	VACUUM当前正在清理堆。清理堆与扫描堆不是同一个概念，清理堆发生在每一次清理索引的实例之后。如果heap_blks_scanned小于heap_blks_total，系统将在这个阶段完成之后回去扫描堆；否则，系统将在这个阶段完成后开始清理索引。
清除索引	VACUUM当前正在清除索引。这个阶段发生在堆被完全扫描并且对堆和索引的所有清理都已经完成以后。
截断堆	VACUUM正在截断堆，以便把关系尾部的空页面返还给操作系统。这个阶段发生在清除完索引之后。
执行最后的清除	VACUUM在执行最终的清除。在这个阶段中，VACUUM将清理空闲空间映射、更新sys_class中的统计信息并且将统计信息报告给统计收集器。当这个阶段完成时，VACUUM也就结束了。

5. 动态追踪

KingbaseES提供了功能来支持数据库服务器的动态追踪。允许在代码中的特定点上调用外部工具来追踪执行过程。

一些探针或追踪点已经被插入在源代码中。这些探针的目的是被数据库开发者和管理员使用。默认情况下，探针不被编译到KingbaseES中，用户需要显式地告诉配置脚本使得探针可用。

目前，在写文档时DTrace已被支持，它在 Solaris、OS X、FreeBSD、NetBSD 和 Oracle Linux 上可用。Linux 的SystemTap项目提供了一种可用的 DTrace 等价物。支持其他动态追踪工具在理论上可以通过改变src/include/utils/probes.h中的宏定义实现。

5.1. 动态追踪的编译

默认情况下，探针是不可用的，因此用户需要显式地告诉配置脚本让探针在KingbaseES中可用。启用 DTrace 支持，在配置时指定--enable-dtrace。

5.2. 内建探针

如表 5-1所示，源代码中提供了一些标准探针。表 5-2显式了在探针中使用的类型。当然，可以增加更多探针来增强KingbaseES的可观测性。

表 5-1. 内建 DTrace 探针

名称	参数	描述
transaction-start	(LocalTransactionId)	在一个新事务开始时触发的探针。arg0 是事务 ID。
transaction-commit	(LocalTransactionId)	在一个事务成功完成时触发的探针。arg0 是事务 ID。
transaction-abort	(LocalTransactionId)	当一个事务失败完成时触发的探针。arg0 是事务 ID。
query-start	(const char *)	当一个查询的处理被开始时触发的探针。arg0 是查询字符串。
query-done	(const char *)	当一个查询的处理完成时触发的探针。arg0 是查询字符串。
query-parse-start	(const char *)	当一个查询的解析被开始时触发的探针。arg0 是查询字符串。
query-parse-done	(const char *)	当一个查询的解析完成时触发的探针。arg0 是查询字符串。
query-rewrite-start	(const char *)	当一个查询的重写被开始时触发的探针。arg0 是查询字符串。
query-rewrite-done	(const char *)	当一个查询的重写完成时触发的探针。arg0 是查询字符串。
query-plan-start	()	当一个查询的规划被开始时触发的探针。
query-plan-done	()	当一个查询的规划完成时触发的探针。
query-execute-start	()	当一个查询的执行被开始时触发的探针。
query-execute-done	()	当一个查询的执行完成时触发的探针。

名称	参数	描述
statement-status	(const char *)	任何时候当服务器进程更新它的sys_stat_activity.status时触发的探针。arg0 是新的状态字符串。
checkpoint-start	(int)	当一个检查点被开始时触发的探针。arg0 保持逐位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。
checkpoint-done	(int, int, int, int, int)	当一个检查点完成时触发的探针（检查点处理过程中序列中列出的下一个触发的探针）。arg0 是要写的缓冲区数量。arg1 是缓冲区的总数。arg2、arg3 和 arg4 分别包含了增加、删除和循环回收的 WAL 文件的数量。
clog-checkpoint-start	(bool)	当一个检查点的 CLOG 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
clog-checkpoint-done	(bool)	当一个检查点的 CLOG 部分完成时触发的探针。arg0 的含义与clog-checkpoint-start中相同。
subtrans-checkpoint-start	(bool)	当一个检查点的 SUBTRANS 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
subtrans-checkpoint-done	(bool)	当一个检查点的 SUBTRANS 部分完成时触发的探针。arg0 的含义与subtrans-checkpoint-start中相同。
multixact-checkpoint-start	(bool)	当一个检查点的 MultiXact 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
multixact-checkpoint-done	(bool)	当一个检查点的 MultiXact 部分完成时触发的探针。arg0 的含义与multixact-checkpoint-start中相同。
buffer-checkpoint-start	(int)	当一个检查点的写缓冲区部分被开始时触发的探针。arg0 保持逐位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。

名称	参数	描述
buffer-sync-start	(int, int)	当我们在检查点期间开始写脏缓冲区时（在标识哪些缓冲区必须被写之后）触发的探针。arg0 是缓冲区总数，arg1 是当前为脏并且需要被写的缓冲区数量。
buffer-sync-written	(int)	在检查点期间当每个缓冲区被写完之后触发的探针。arg0 是缓冲区的 ID。
buffer-sync-done	(int, int, int)	当所有脏缓冲区被写之后触发的探针。arg0 是缓冲区总数。arg1 是检查点进程实际写的缓冲区数量。arg2 是期望写的数目（buffer-sync-start的 arg1）；arg1 和 arg2 的任何的不同反映在该检查点期间有其他进程刷写了缓冲区。
buffer-checkpoint-sync-start	()	在脏缓冲区被写入到内核之后并且在开始发出 fsync 请求之前触发的探针。
buffer-checkpoint-done	()	当同步缓冲区到磁盘完成时触发的探针。
twophase-checkpoint-start	()	当一个检查点的两阶段部分被开始时触发的探针。
twophase-checkpoint-done	()	当一个检查点的两阶段部分完成时触发的探针。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	当一次缓冲区读被开始时触发的探针。arg0 和 arg1 包含该页的分叉号和块号（如果这是一次关系扩展请求，arg1 为 -1）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。表示真，对共享缓冲区表示假。 arg6 为真表示一次关系扩展请求，为假表示正常读。

名称	参数	描述
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	当一次缓冲区读完成时触发的探测器。arg0 和 arg1 包含该页的分叉号和块号（如果这是一次关系扩展请求，arg1 现在包含新增加块的块号）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。表示真，对共享缓冲区表示假。 arg6 为真表示一次关系扩展请求，为假表示正常读。arg7 为真表示在池中找到该缓冲区，为假表示没有找到。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	在发出对一个共享缓冲区的任意写请求之前触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个写请求完成时触发的探针（注意这只反映传递数据给内核的时间，它通常并没有实际地被写入到磁盘）。参数和buffer-flush-start的相同。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个服务器进程开始写一个脏缓冲区时触发的探针（如果这经常发生，表示shared_buffers太小，或需要调整后台写入器的控制参数）。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一次脏缓冲区写完成时触发的探针。参数与buffer-write-dirty-start相同。
wal-buffer-write-dirty-start	()	当一个服务器进程因为没有可用 WAL 缓冲区空间开始写一个脏 WAL 缓冲区时触发的探针（如果这经常发生，表示wal_buffers太小）。
wal-buffer-write-dirty-done	()	当一次脏 WAL 缓冲区完成时触发的探针。

名称	参数	描述
xlog-insert	(unsigned char, unsigned char)	当一个 WAL 记录被插入时触发的探针。arg0 是该记录的资源管理者（rmid）。arg1 包含 info 标志。
xlog-switch	()	当请求一次 WAL 段切换时触发的探针。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始从一个关系读取一块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一次块读取完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。arg6 是实际读取的字节数，而 arg7 是请求读取的字节数（如果两者不同就意味着麻烦）。
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始向一个关系中写入一个块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个块写操作完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3和 arg4 包含表空间、数据库和关系 OID来标识该关系。对于一个本地缓冲区，arg5 是创建临时关系的后端 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。arg6 是实际写的字节数，而 arg7 是要求写的字节数（如果这两者不同，则意味着麻烦）。

名称	参数	描述
sort-start	(int, bool, int, int, bool)	当一次排序操作开始时触发的探针。arg0 指示是堆排序、索引排序或数据排序。arg1 为真表示唯一值强制。arg2 是键列的数目。arg3 是允许使用的工作内存数（以千字节计）。如果要求随机访问排序结果，那么 arg4 为真。
sort-done	(bool, long)	当一次排序完成时触发的探针。arg0 为真表示外排序，为假表示内排序。arg1 是用于一次外排序的磁盘块的数目，或用于一次内排序的以千字节计的内存。
lwlock-acquire	(char *, int, LWLockMode)	当成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是该 LWLock 在其所在切片中的偏移量。arg2 所请求的锁模式，是排他或共享。
lwlock-release	(char *, int)	当一个 LWLock 被释放时（但是注意还没有唤醒任何一个被释放的等待者）触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是该 LWLock 在其所在切片中的偏移量。
lwlock-wait-start	(char *, int, LWLockMode)	当一个 LWLock不是当即可用并且一个服务器进程因此开始等待该锁变为可用时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是该 LWLock 在其所在切片中的偏移量。arg2 所请求的锁模式，是排他或共享。
lwlock-wait-done	(char *, int, LWLockMode)	当一个进程从对一个 LWLock 的等待中被释放时（它实际还没有得到该锁）时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 是该 LWLock 在其所在切片中的偏移量。 arg2 所请求的锁模式，是排他或共享。
lwlock-condacquire	(char *, int, LWLockMode)	当调用者指定无需等待而成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 是该 LWLock 在其所在切片中的偏移量。 arg2 所请求的锁模式，是排他或共享。

名称	参数	描述
lwlock-condacquire-fail	(char *, int, LWLockMode)	当调用者指定无需等待而没有成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 是该 LWLock 在其所在切片中的偏移量。 arg2 所请求的锁模式，是排他或共享。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr锁）的请求由于锁不可用开始等待时触发的探针。arg0 到 arg3 是标识被锁定对象的标签域。arg4 指示被锁对象的类型。arg5 表示被请求的锁类型。
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求结束等待时（即已经得到锁）触发的探针。参数与lock-wait-start一样。
deadlock-found	()	当死锁检测器发现死锁时触发的探针。

表 5-2. 定义用在探针参数中的类型

类型	定义
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	char

5.3. 使用探针

例5-1：展示了一个分析系统中事务计数的 DTrace 脚本，可以用来代替一次性能测试之前和之后的`sys_stat_database`快照：

```
#!/usr/sbin/dtrace -qs

kingbase$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

kingbase$1:::transaction-abort
{
      @abort["Abort"] = count();
}

kingbase$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

当被执行时，该例子中脚本的输出为：

```
# ./txn_count.d `pgrep -n kingbase` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                          2312105013
```

> **注意：**SystemTap 为追踪脚本使用一个不同于 DTrace 的标记，但是底层的探针是兼容的。值得注意的是，这样写的时候，SystemTap 脚本必须使用双下划线代替连字符来引用探针名。在未来的 SystemTap 版本中这很可能会被修复。

DTrace 脚本需要细心地编写和调试，否则被收集的追踪信息可能会毫无意义。在大部分发生问题的情况中，它只是发生问题的部件，而不是底层系统。当讨论使用动态追踪发现的信息时，一定要封闭使用的脚本来使用探针以便被检查和讨论。

5.4. 定义新探针

开发者可以在代码中任意位置定义新的探针，当然要重新编译之后才能生效。下面是插入新探针的步骤：

a. 决定探针名称以及探针可用的数据

b. 把该探针定义加入到src/backend/utils/probes.d

c. 如果sys_trace.h还不存在于包含该探针点的模块中，在模块添加它，并在源代码中期望的位置插入TRACE_KINGBASEQL探针宏

d. 重新编译并验证新探针是可用的

例5-2：如何增加一个用事务ID追踪所有新事务的探针

a. 决定探针将被命名为transaction-start并且需要一个LocalTransactionId类型的参数

b. 将该探针定义加入到src/backend/utils/probes.d：

```
probe transaction__start(LocalTransactionId);
```

注意探针名字中双下划线的使用。在一个使用探针的 DTrace 脚本中，双下划线需要被替换为一个连字符，因此，对用户而言transaction-start是文档名。

c. 在编译时，transaction__start被转换成一个宏调用TRACE_KINGBASEQL_TRANSACTION_START（注意这里是单下划线），可以通过包括头文件sys_trace.h获得。将宏调用加入到源代码中的合适位置。这种情况下，类似于：

```
TRACE_KINGBASEQL_TRANSACTION_START(vxid.localTransactionId);
```

d. 在重新编译和运行新的二进制文件之后，通过运行下面的 DTrace 命令来检查新增的探针是否可用。应当看到类似下面的输出：

```
# dtrace -ln transaction-start
   ID    PROVIDER          MODULE           FUNCTION NAME
18705 kingbase49878     kingbase     StartTransactionCommand
transaction-start
18755 kingbase49877     kingbase     StartTransactionCommand
transaction-start
18805 kingbase49876     kingbase     StartTransactionCommand
transaction-start
18855 kingbase49875     kingbase     StartTransactionCommand
transaction-start
18986 kingbase49873     kingbase     StartTransactionCommand
transaction-start
```

向C代码中添加追踪宏时，有一些事情需要注意：

- 为探针参数指定的数据类型要匹配宏中使用的变量的数据类型，否则会发生编译错误。

- 在大多数平台上，如果用--enable-dtrace编译了KingbaseES，无论何时只要有一个宏时，都会评估该宏的参数，即使没有进行追踪它也会这样做。通常不需要担心你是否只在报告一些局部变量的值。但要注意将开销大的函数调用放置在这些参数中。如果需要这样做，考虑通过检查追踪是否真的被启用来保护该宏：

```
if (TRACE_KINGBASEQL_TRANSACTION_START_ENABLED())
    TRACE_KINGBASEQL_TRANSACTION_START(some_function(...));
```

每个追踪宏有一个对应的ENABLED宏。

监控磁盘使用

1. 判断磁盘用量

每个表都有一个主要的堆磁盘文件，大多数数据都存储在其中。如果一个表有着可能会很宽（尺寸大）的列， 则另外还有一个TOAST文件与这个表相关联， 它用于存储因为太宽而不能存储在主表里面的值。如果有这个附属文件，那么TOAST表上会有一个可用的索引。 当然，同时还可能有索引和基表关联。每个表和索引都存放在单独的磁盘文件里 — 如果文件超过 1G 字节，甚至可能多于一个文件。

用户可以以三种方式监视磁盘空间：使用*《SQL和PL/SQL速查手册》II. SQL语言基础-“函数和操作符”-“系统管理函数”表 7-1*中列出的SQL函数、使用 oid2name 模块或者人工观察系统目录。SQL函数是最容易使用的方法，同时也是KingbaseES推荐的方法。

在一个最近清理过或者分析过的数据库上使用ksql，发出查询命令来查看任意表的磁盘用量：

```
SELECT sys_relation_filepath(oid), relpages FROM sys_class WHERE relname
 = 'customer';

 sys_relation_filepath | relpages
-----------------------+----------
 base/16384/16806      |       60
(1 row)
```

每个页通常都是 8K 字节（记住，relpages只会由VACUUM、ANALYZE和少数几个 DDL 命令如CREATE INDEX所更新）。如果想直接检查表的磁盘文件，那么文件路径名应该有用。

显示TOAST表使用的空间，可以使用一个类似下面这样的查询：

```
SELECT relname, relpages
FROM sys_class,
     (SELECT reltoastrelid
      FROM sys_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM sys_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
```

```
 sys_toast_16806       |         0
 sys_toast_16806_index |         1
```

显示索引的尺寸：

```
SELECT c2.relname, c2.relpages
FROM sys_class c, sys_class c2, sys_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname         | relpages
-----------------------+----------
 customer_id_indexdex |       26
```

找出最大的表和索引：

```
SELECT relname, relpages
FROM sys_class
ORDER BY relpages DESC;

       relname         | relpages
-----------------------+----------
 bigtable              |     3290
 customer              |     3144
```

2. 磁盘满失败

数据库管理员最重要的磁盘监控任务就是确保磁盘不会写满。当数据磁盘写满后，或许不会导致数据的崩溃，但会使得系统变得不可用。若保存 WAL 文件的磁盘被变满，会使数据库服务器致命错误并可能发生关闭。

因此，数据库管理员为保证磁盘不被写满，可以删除一些垃圾文件来释放磁盘空间，或使用表空间将部分数据库文件移动到其他文件系统上去。

> **提示:** 某些文件系统的性能在磁盘即将写满时才会急剧恶化，因此需避免磁盘完全满时才采取行动。

如果系统支持每个用户的磁盘配额，那么数据库将自然受制于服务器作为用户运行的任何配额。超过配额将产生与完全耗尽磁盘空间相同的不良影响。

健康状态检查

数据库健康状态检查可以对数据库的运行情况进行分析，及时发现潜在的性能问题、数据库异常、数据文件异常等，从而防患于未然。

健康状态检查分为在线健康检查和离线健康检查两种方式。在线状态检查是KingbaseES的一个扩展，通过ksql执行SQL语句来执行检查；线状态检查是KingbaseES的一个工具，通过sys_hm工具对数据库进行健康状态检查。执行离线状态检查时，数据库必须停止运行。

1. 检查类型

表 1-1. 检查类型

检查类型	注释	是否支持离线	描述
DB Structure Integrity Check	数据库文件完整性检	是	检查数据目录中的文件列表是否完整
Data Block Integrity Check	数据文件进行页面完整性检查	是	检查具体表所在文件的块是否完整
Xlog Integrity Check	Xlog完整性检查	是	检查最后一个checkpoint的所在Xlog文件是否存在
Logical Block Check	数据块中逻辑内容的检查	否	检查具体表所在文件块的内容
All Control Files Check	检查数据库的所有控制文件	是	检查所有控制文件是否存在问题
Control File Backup Check	检测某个控制文件的副本	是	检查控制文件副本是否存在问题
All Datafiles Check	检查所有的数据文件	是	校验所有的数据文件（存储表数据的文件），是否存在文件损坏
Single Datafile Check	检查磁盘上的某个数据文件	是	校验具体数据文件（存储表数据的文件），是否存在文件损坏
All Xlog Check	检查所有的xlog完整性	是	校验所有Xlog文件，是否存在损坏

检查类型	注释	是否支持离线	描述
Single Xlog Check	单个Xlog文件检查	是	校验特定Xlog文件，是否存在损坏
Archived Xlog Check	归档Xlog文件检查	是	检查具体归档文件，是否存在损坏
Dictionary Integrity Check	数据字典的完整性检查	否	检查指定（所有）数据字典的完整性
Autovacuum Integrity Check	自动回收检查	否	检查自动回收是否生效
Long Time Connection Check	长连接检查	否	找出长时间处于IDLE状态的连接，默认是7200秒
Long Transaction Detection	长事务检查	否	找出长时间处于IDLE Transaction的事务，默认是7200秒
Connection Percent Check	连接数占比检查	否	检查当前连接数和最大连接数的占比
Disk Usage Check	磁盘使用率检查	是	检查磁盘使用量和磁盘总容量的占比
IO Usage Check	IO使用率检查	是	检查IO使用率
OS Load Average Check	操作系统负载检查	是	检查当前系统负载
CPU Usage Check	CPU使用率检查	是	CPU使用率检查
Memory Usage Check	内存使用率检查	是	内存使用率检查
License Validity Check	License有效性检查	是	License有效性检查
DB Version Check	版本一致性检查	是	检查服务器bin目录下的二进制文件版本是否一致：kingbase、sys_dump、sys_restore、initdb
DB User Count Check	数据库用户数量检查	是	数据库用户数量检查
Lock Wait Check	锁等待检查	是	锁等待检查

检查类型	注释	是否支持离线	描述
IO Schedule Check	IO调度检查	是	IO调度检查
Network Check	网络检查	是	检查系统所有网络是否可用

2. 检查参数

每种检查类型的输入参数详细信息保存在视图SYS_HM.PARAM中。

表 2-1. 检查参数

检查类型	参数类型	参数名称	默认值
DB Structure Integrity Check	INTEGER	RELFILENODE	
Data Block Integrity Check	INTEGER	BLOCKNUM	
Data Block Integrity Check	INTEGER	RELTABLESPACE	
Data Block Integrity Check	INTEGER	RELFILENODE	
Logical Block Check	INTEGER	RELFILENODE	
Logical Block Check	INTEGER	RELTABLESPACE	
Logical Block Check	INTEGER	BLOCKNUM	
Control File Backup Check	TEXT	CTL_FILE_ABS_PATH	
Single Datafile Check	INTEGER	RELFILENODE	
Single Datafile Check	INTEGER	RELTABLESPACE	
Single Xlog Check	TEXT	XLOG_FILE_ABS_PATH	
Archived Xlog Check	TEXT	XLOG_FILE_ABS_PATH	
Dictionary Integrity Check	TEXT	RELNAME	
Autovacuum Integrity Check	INTEGER	THRESHOLD	500000000
Long Time Connection Check	INTEGER	TIME	7200
Memory Usage Check			
IO Schedule Check			
Network Check			
IO Usage Check			
OS Load Average Check			
Connection Percent Check			
License Validity Check			

检查类型	参数类型	参数名称	默认值
All Control Files Check			
CPU Usage Check			
Lock Wait Check			
Disk Usage Check			
DB User Count Check			
DB Structure Integrity Check			
DB Version Check			
Xlog Integrity Check			
All Xlog Check			
All Datafiles Check			

3. 执行检查

3.1. 在线检查

3.1.1. 执行检查

在线状态检查通过下面的函数来进行检查：

```
select * from sys_hm.run_check(check_type_name name,
                               run_name name,
                               input_params text DEFAULT NULL,
                               timeout number DEFAULT NULL);
```

`check_type_name`

检查类型名称

`run_name`

本次检查的名称，由用户自定义，检查名称作为检查的唯一标识符，不能和以前的检查重复

`input_params`

输入参数，'key1=val1;key2=val2'，各种检查的输入参数详情见第 2 节

3.1.2. 示例

执行'DB Structure Integrity Check'检查：

```
KINGBASE=# select * from sys_hm.run_check('DB Structure Integrity Check',
'tst1');
                          RUN_CHECK
---------------------------------------------------------------
 Basic Run Information.                                       +
 Run Name                        :tst2                        +
 Run Id                          :3                           +
 Check Name                      :DB Structure Integrity Check +
 Mode                            :Online                      +
 Status                          :Success                     +
 Start Time                      :2018-01-05 07:03:35.083074+08+
 End Time                        :2018-01-05 07:03:35.083074+08+
 Timeout                         :0                           +
 Error Number                    :0                           +
                                                              +
 Input Paramters for the Run:                                 +

(1 row)
```

3.1.3. 显示结果

调用函数sys_hm.show_run(run_name name)可显示名称为run_name的检查的结果

```
KINGBASE=# select * from sys_hm.show_run('tst1');
                        SHOW_RUN
-------------------------------------------------------------------------------
 Basic Run Information.                                                       +
 Run Name                 :tst2                                               +
 Run Id                   :3                                                  +
 Check Name               :DB Structure Integrity Check                       +
 Mode                     :Online                                             +
 Status                   :Success                                            +
 Start Time               :2018-01-05 07:03:35.083074+08                      +
 End Time                 :2018-01-05 07:03:35.083074+08                      +
 Timeout                  :0                                                  +
 Error Number             :2                                                  +
                                                                              +
 Check Result             :                                                   +
  could not open file "/home/gliu/kingbase_data/sys_xlog": No such file
  or directory                                                                +
  could not open file "/home/gliu/kingbase_data/sys_xlog/archive_status":
  No such file or directory                                                   +

                                                                              +
 Input Paramters for the Run:                                                 +

(1 row)
```

3.2. 离线检查

3.2.1. 执行检查

离线状态检查运行sys_hm工具进行检查：

```
Usage:
  sys_hm [OPTION]... sys_hm [options]

Options:
  -d              generate debug output (verbose mode)
  -D              data directory
  -r              run name
  -t              check type
  -p              check param
  -S, --show      show this check type, then exit
  -V, --version   output version information, then exit
  -?, --help      show this help, then exit
```

-d

debug模式，会打印更多的信息

-D

要检查的数据目录

-r

本次检查的名称

-t

本次检查的类型

-p

本次检查的输入参数，'key1=val1;key2=val2'，各种检查的输入参数详情见第 2 节

-S

显示离线检查所支持的类型

3.2.2. 示例

```
[localhost]$./sys_hm -D"/opt/db/data" -t"Data Block Integrity
 Check" -r"Data Block Integrity Check" -p"FILEPATH=/opt/db/
 data/base/1/3603@BLOCKNUM=2"
Basic Check Information.
Run Name             :Data Block Integrity Check
Check Name           :Data Block Integrity Check
Mode                 :Offline
Status               :Success
Start Time           :2018-01-04 16:31:42.150946 CST
End Time             :2018-01-04 16:31:42.150967 CST
Error Number         :1
Check Result         :The enter blocknum 2 is too large for file"/
opt/db/data/base/1/3603"
```

版权申明

联系我们

欢迎您针对此手册提出宝贵意见和建议，您的意见和建议将成为完善此手册的重要部分。

如果您发现了手册中的错误，或有更好的意见和建议，可通过以下方式发送给我们。

电子邮箱：support@kingbase.com.cn

传真：86-10-59111032

服务热线：4006011188

通信地址：北京市朝阳区容达路7号E座2层