第 11 章　日志与恢复

11.1 实验介绍

日志与恢复是 openGauss 数据库实现事务处理和确保 ACID 特性的重要组成部分。本实验尝试打通目前数据库日志与恢复模块在原理学习与系统实现上的鸿沟。通过 openGauss 数据库中日志与恢复部分的实现源代码，分析与验证相关原理与机制，包括：WAL 日志文件、XLOG 日志记录、日志写入过程、检查点机制、数据库恢复、数据库备份与 PITR 恢复。

首先，通过实验查看 WAL 日志文件的基本信息与命名方式；然后，通过“立即”关闭模式验证数据库在重启时的恢复过程；通过添加代码的方法，在数据库恢复过程中输出调试信息，结合源代码阅读，更加详细地分析 WAL 日志恢复过程；最后，验证数据库备份与 PITR 恢复机制。

日志与恢复机制是数据库系统最为繁杂的功能模块之一，数据库日志与恢复功能的实现需要考虑众多原理中忽略的细节，大量涉及较为底层的系统机制。本实验的实践内容包括较多的源代码阅读与分析，具有较大的挑战性。

11.2 实验目的

1. 理解 WAL 日志文件的工作原理。
2. 理解 XLOG 日志记录的组织。
3. 理解 WAL 日志写入过程。
4. 理解 WAL 日志检查点机制。
5. 理解利用 WAL 日志重做 XLOG 记录进行数据库恢复的原理。
6. 掌握数据库备份与 PITR 恢复方法。
7. 了解与本实验相关的函数与结构体的源代码。

11.3 实验原理

日志与恢复是数据库管理系统中事务管理机制的重要组成部分。数据库的日志记录功能与基于日志的数据恢复逻辑是确保在系统故障的情况下，数据库不会损坏和丢失任何数据的保障手段。日志中记录了数据库中发生的所有动作。当故障发生时，数据库服务器可以通过依次重新执行日志中记录的动作，将数据库恢复到故障发生时的正确状态上。

11.3.1　WAL 日志文件

1. 没有 WAL 日志的插入操作

我们在名为 TABLE_A 的表中插入数据，假设 TABLE_A 在磁盘上只占一个页面。在没有 WAL 日志的数据库中，操作如图 8.1 所示。

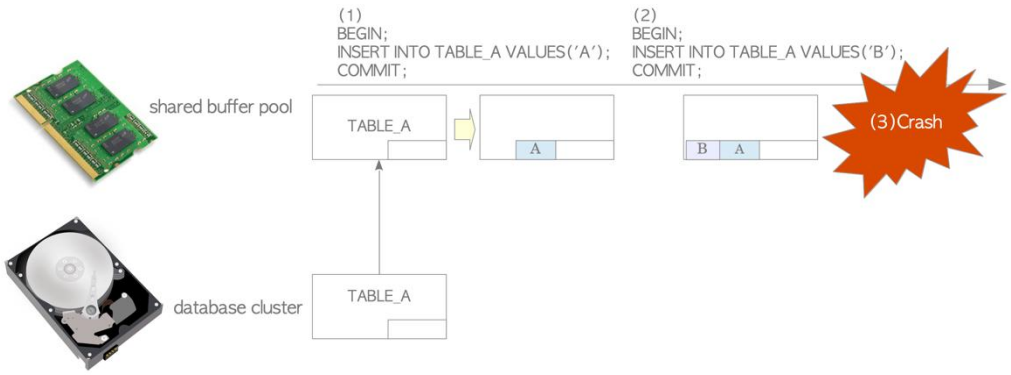

图 8.1 在没有 WAL 日志的数据库中插入数据

- 在第(1)步中，执行第 1 条 INSERT 语句，数据库将 TABLE_A 的页面从磁盘上装载到内存的共享缓存区（shared buffer pool），将一条元组插入该页面，该页面没有立即写回磁盘，修改后的页面称为脏页（dirty page）。
- 在第(2)步中，执行第 2 条 INSERT 语句，向位于缓存区的页面再次插入一条元组，该页面还是没有写回磁盘。
- 在第(3)步中，系统由于某种原因崩溃，此时已经插入的两条元组都将丢失。

因而，在没有 WAL 日志的情况下，当发生系统故障时，数据库可能丢失数据。

2. 插入操作和数据库恢复

为了解决系统故障时的数据丢失问题，openGauss 数据库引入了 WAL 日志机制，将所有数据更新作为历史记录保存到持久性存储上（如磁盘文件），以备故障时恢复数据使用。在 openGauss 中，这些历史记录被保存为 WAL 日志文件，其中包含 XLOG 日志记录。当增删改等更新操作执行时，XLOG 记录首先被写入到内存中的 WAL 缓存区（WAL buffer）。当当事务提交/回滚时，立即把 WAL 缓冲区中的内容写入磁盘上的 WAL 段文件 (WAL segment file）。XLOG 记录的 LSN（Log Sequence Number）是该记录在 WAL 日志中的唯一标识符，表示其在 WAL 日志文件中的位置。

pg 崩溃恢复的起点是哪里？答案是重做点（REDO point），即最新的检查点开始时 xlog 记录写入的位置。如图 8.2 所示。

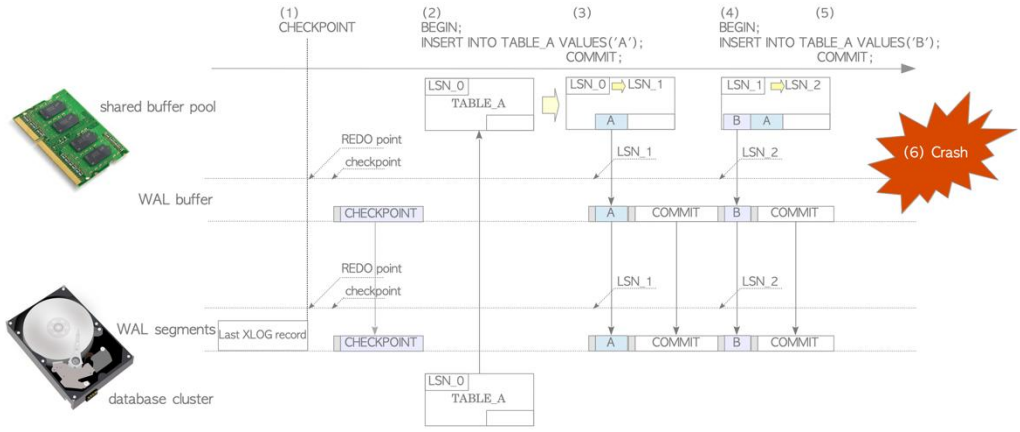

图 8.2 在有 WAL 日志的数据库中插入数据

- 第(1)步：名为 checkpointer 的后台线程定期执行检查点操作（checkpointing）。每当 checkpointer 启动时，会将一条名为 checkpoint record 的 XLOG 记录写入当

前 WAL 段。此记录包含最新的 REDO point 的位置（最新的检查点开始时 XLOG 记录写入的位置）。

- 第(2)步：发出第一个 INSERT 语句。将 TABLE_A 的页面从磁盘文件加载到共享缓冲区中，向该页中插入一个元组，向 WAL 缓冲区的 LSN_1 位置写入一条相应的 XLOG 记录。将 TABLE_A 的 LSN 从 LSN_0 更新到 LSN_1。在本例中，XLOG 记录是“头数据+整个元组”。
- 第(3)步：此时该事务提交，创建并向 WAL 缓冲区写入一条 commit 对应的 XLOG 记录，将 WAL 缓冲区中从 LSN_1 位置开始的所有 XLOG 记录写入 WAL 段文件。
- 第(4)步：发出第二个 INSERT 语句。继续向该页中插入一个新元组，向 WAL 缓冲区 LSN_2 位置写入一条相应的 XLOG 记录，将 TABLE_A 的 LSN 从 LSN_1 更新到 LSN_2。
- 第(5)步：此时该事务提交，创建并向 WAL 缓冲区写入一条 commit 对应的 XLOG 记录，将 WAL 缓冲区中从 LSN_2 位置开始的所有 XLOG 记录写入 WAL 段文件。
- 第(6)步：此时系统崩溃。即使共享缓冲区中的所有数据都丢失，因为页面的所有修改都已作为历史数据写入 WAL 段文件，这些修改可以恢复回来。

下面将说明如何将数据库恢复到崩溃之前的状态。不需要任何特殊操作，故障之后重启 openGauss 时会自动进入恢复模式（recovery-mode），会从重做点开始顺序读取和重做（replay）相应 WAL 段文件中的 XLOG 记录。如图 8.3 所示。

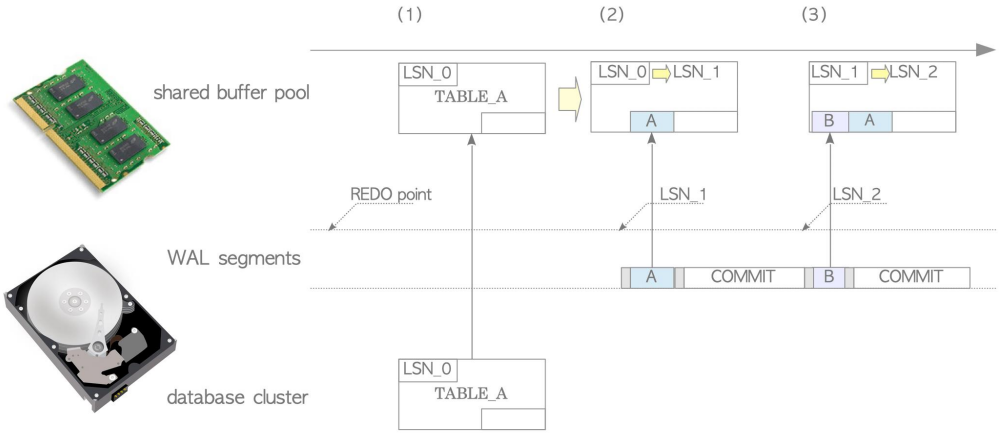

图 8.3 使用 WAL 日志的数据库恢复

- 第(1)步：从相应的 WAL 段文件中读取第一个 INSERT 语句的 XLOG 记录，并将 TABLE_A 的页面从数据库磁盘文件加载到共享缓冲区中。
- 第(2)步：在重做 XLOG 记录之前，会比较 XLOG 记录的 LSN 和相应页面的 LSN。重做 XLOG 记录的规则为：

  如果 XLOG 记录的 LSN 大于页面的 LSN（XLOG 中数据较新），则 XLOG 记录中的数据部分将插入页面，并将页面 LSN 更新为 XLOG 记录的 LSN。

  如果 XLOG 记录的 LSN 较小（数据文件已是最新数据），不用做任何操作，直接读取下一个 WAL 数据。

  在本例中，XLOG 记录需要被重做，因为 XLOG 记录的 LSN（LSN_1）大于 TABLE_A 的 LSN（LSN_0）；重做后，将 TABLE_A 的 LSN 从 LSN_0 更新到 LSN_1。
- 第(3)步：按照同样的方式重做其余的 XLOG 记录。

openGauss 可以通过按时间顺序重做 WAL 段文件中的 XLOG 记录完成整个恢复过程，因此，openGauss 实现的 XLOG 是 REDO 日志。

是不是只要按照上述方式写 WAL 日志文件就一定能够进行故障恢复呢？答案是否定的。因为还需要解决部分写（partial writes）的问题，实现一种称为全页写（full-page writes）的机制。

当数据库正在执行将脏页写入到磁盘文件过程中，操作系统发生故障，此时 TABLE_A 的页面数据损坏，即只有页面中的一部分数据写入了磁盘文件，这称为部分写问题。这种页面是损坏的。在故障恢复时，XLOG 记录无法在该损坏的页面上重做。

openGauss 默认支持全页写功能来处理部分写问题。openGauss 会在每个检查点之后，每个页面第一次发生变更时，将“头数据+整个页面”作为一条 XLOG 记录写入 WAL 缓冲区。这种包含整个页面的 XLOG 记录称为备份块（backup block）或全页镜像 (full-page image)。

下面给出在带有全页写功能的 WAL 日志机制下插入数据的流程。如图 8.4 所示。

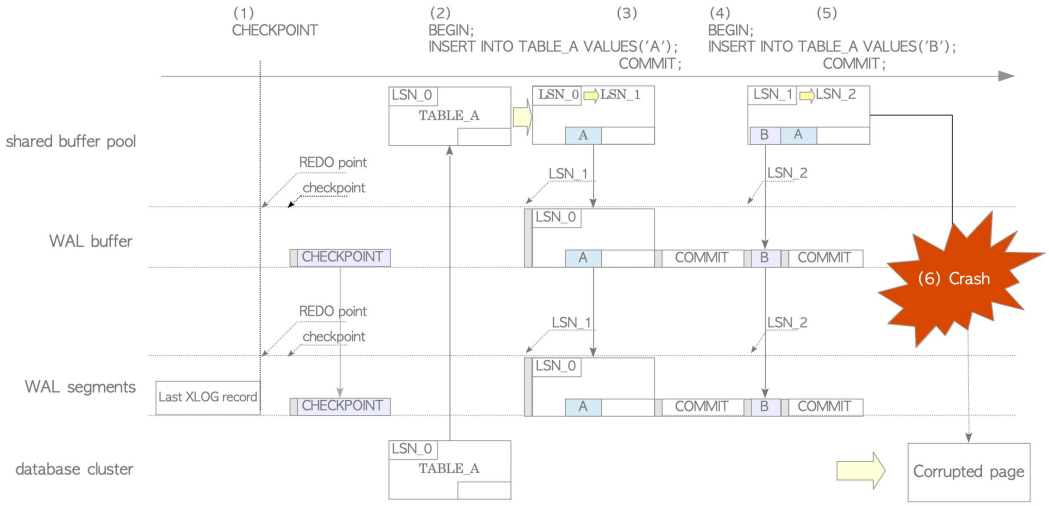

图 8.4 在带有全页写功能的 WAL 日志机制下插入数据

- 第(1)步：checkpointer 线程启动一个检查点。
- 第(2)步：在第 1 个 INSERT 语句插入时，与前面过程的不同之处在于，此时 XLOG 记录的是“头数据+整个页面”的备份块（backup block）（包含整个页面而不仅是插入的元组），因为这是在最新的检查点之后该页面的第一次写入。
- 第(3)步：此时该事务提交，操作方式与前面对应步骤相同。
- 第(4)步：在第 2 个 INSERT 语句插入时，操作方式与之前对应步骤完全相同，因为此时 XLOG 记录只是“头数据+插入元组”，而不是备份块。
- 第(5)步：此时该事务提交，操作方式与前面对应步骤相同。
- 第(6)步：为了说明全页写功能的效果，考虑当写入磁盘文件过程中，发生操作系统故障，TABLE_A 在磁盘上的页面发生部分写问题而被破坏。

openGauss 重启时，恢复过程如图 8.5 所示。

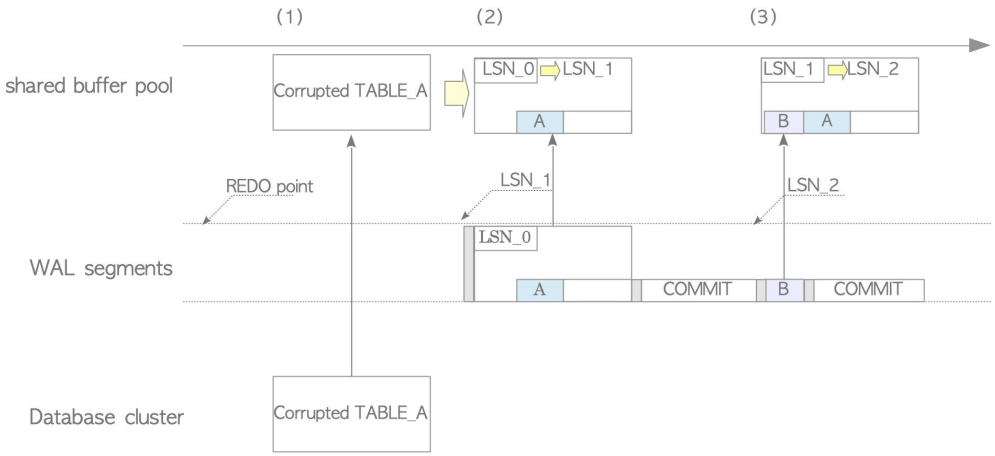

图 8.5 在带有全页写功能的 WAL 日志机制下进行故障恢复

- 第(1)步：读取第 1 个 INSERT 语句的 XLOG 记录，并将损坏的 TABLE_A 页面从磁盘文件加载到共享缓冲区中。在本例中，XLOG 记录是备份块，因为根据全页写规则，每个页面的第 1 个 XLOG 记录总是备份块。
- 第(2)步：当 XLOG 记录是备份块时，使用另一个重做规则，即 XLOG 记录的数据部分（即页面本身）会直接覆盖当前页面而不去比较 LSN 值，并且页面的 LSN 更新为 XLOG 记录的 LSN。

  在本例中，使用 XLOG 中记录的数据部分覆盖了损坏的页面，并将 TABLE_A 的 LSN 更新为 LSN_1。通过这种方式，损坏的页面通过其对应的备份块恢复了。
- 第(3)步：第 2 个 XLOG 记录是非备份块，操作方式与前面对应步骤相同。

此时，即使发生了一些由于操作系统原因而引起的数据写入错误，数据库也可以恢复。当然，如果发生的是文件系统或介质故障（即磁盘坏了），就不能仅通过 WAL 日志机制恢复了。通过备份（backup）和复制（replication）机制，再结合 WAL 日志，是可以保证即使在介质故障发生时数据库也可以恢复的。

在逻辑上，openGauss 将 XLOG 记录写入一个地址空间长度为 64 位（8B）的事务日志虚拟文件中（最大可达 16EB，即 2^{64}B）。在物理上，事务日志默认切分为 16 MB 大小的文件，每个文件称为 WAL 段（WAL segment）。

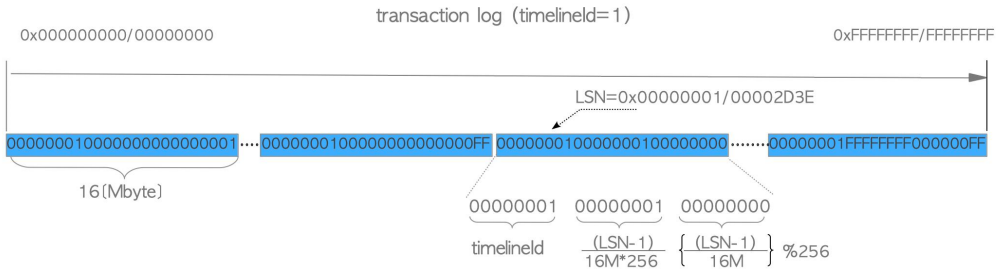

图 8.6 WAL 日志文件的命名编码方式

图 8.6 给出了 WAL 日志文件的命名编码方式。WAL 段文件的名称是一个 24 位的十六进制数，每 8 位是一个部分，即分为三部分：

- 前 8 位：时间线 id（timelineid），用于 PITR 恢复，这里固定为 00000001。
- 中间 8 位：WAL 日志的逻辑 id（logid），每个逻辑 id 对应的日志文件总容量默认大小为 256*16M（逻辑段数量*WAL 段文件大小），即每个逻辑 id 包含 256 个 16M

的物理 WAL 文件。

对于给定的 LSN，logid 的计算公式为：$\frac{LSN-1}{16\text{M}\times 256}$

- 最后 8 位：当前 WAL 段文件是本逻辑 id 的第几个（逻辑段号）（logseg），由于最大是 256 个，只需要使用最后 2 位。

  对于给定的 LSN，logseg 的计算公式为：$\frac{LSN-1}{16\text{M}}\%256$

因而，给定一个 LSN，其所在的 WAL 段文件的名称由下面公式给出：

$$\text{WAL 段文件名} = \text{timelineid} + \frac{LSN-1}{16\text{M}\times 256} + \frac{LSN-1}{16\text{M}}\%256$$

WAL 段文件默认是一个 16MB 的文件，内部切分为若干 8192 字节（8KB）的页面。第 1 个页面的头数据为结构体 XLogLongPageHeaderData，其余页面的头数据为结构体 XLogPageHeaderData。在页面头数据之后，则是写入页面中的一条条 XLOG 记录。如图 8.7 所示。

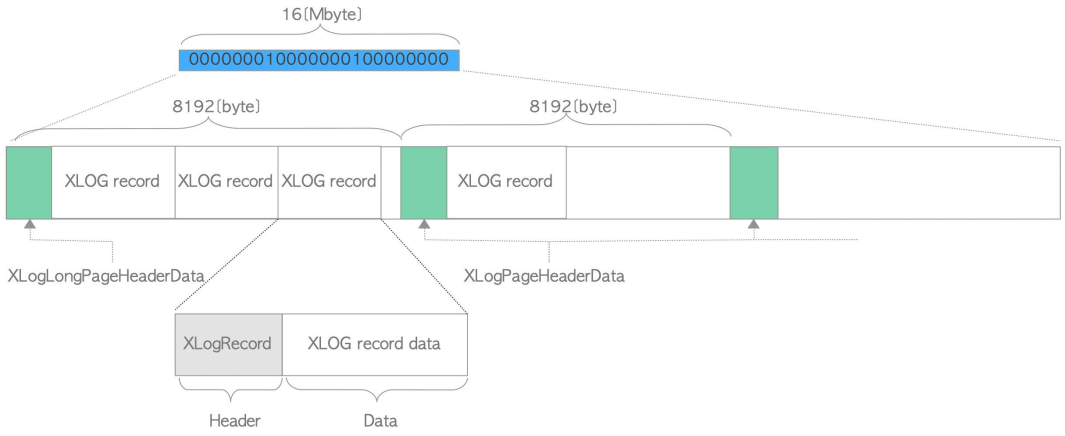

图 8.7 WAL 段文件的内部结构

3. 结构体 XLogPageHeaderData

【源码】src/include/access/xlog_basic.h

```
typedef struct XLogPageHeaderData {
        uint16 xlp_magic;        /* magic value for correctness checks */
        uint16 xlp_info;         /* flag bits, see below */
        TimeLineID xlp_tli;      /* TimeLineID of first record on page */
        XLogRecPtr xlp_pageaddr; /* XLOG address of this page */

        /*
         * When there is not enough space on current page for whole record, we
         * continue on the next page.  xlp_rem_len is the number of bytes
         * remaining from a previous page.
         *
         * Note that xl_rem_len includes backup-block data; that is, it tracks
         * xl_tot_len not xl_len in the initial header.  Also note that the
         * continuation data isn't necessarily aligned.
         */
```

```
        uint32 xlp_rem_len; /* total len of remaining data for record */
        uint32 xlp_total_len;
} XLogPageHeaderData;

typedef XLogPageHeaderData* XLogPageHeader;
```

4. 结构体 XLogLongPageHeaderData

【源码】src/include/access/xlog_basic.h

```
/*
 * When the XLP_LONG_HEADER flag is set, we store additional fields in the
 * page header.  (This is ordinarily done just in the first page of an
 * XLOG file.)  The additional fields serve to identify the file accurately.
 */
typedef struct XLogLongPageHeaderData {
    XLogPageHeaderData std; /* standard header fields */
    uint64 xlp_sysid;       /* system identifier from pg_control */
    uint32 xlp_seg_size;    /* just as a cross-check */
    uint32 xlp_xlog_blcksz; /* just as a cross-check */
} XLogLongPageHeaderData;

typedef XLogLongPageHeaderData* XLogLongPageHeader;
```

11.3.2 XLOG 日志记录

一条 XLOG 记录由通用头部分（XLogRecord 结构体）和数据部分组成。XLOG 记录的数据部分又有自己的头部分和数据部分。

1. 通用头部分：结构体 XLogRecord

字段说明：

- xl_tot_len：该 XLOG 记录的总长度
- xl_xid：事务 ID
- xl_prev：指向前一个 XLOG 记录的指针
- xl_info：标志位
- xl_rmid：该 XLOG 记录的资源管理器（resource manager）
- xl_crc：CRC 校验

其中，x_rmid 和 x_info 字段均与资源管理器相关。对于 XLOG 来说，不同的资源管理器决定着日志写入与恢复时的具体操作。表 7.1 给出了 openGauss 中不同类型的资源管理器。

表 7.1 openGauss 中不同类型的资源管理器

操作	资源管理器
堆表（Heap tuple ）操作	RM_HEAP, RM_HEAP2
索引（Index）操作	RM_BTREE, RM_HASH, RM_GIN, RM_GIST, RM_SPGIST, RM_BRIN
序列（Sequence）操作	RM_SEQ
事务（Transaction）操作	RM_XACT, RM_MULTIXACT, RM_CLOG, RM_XLOG,

	RM_COMMIT_TS
表空间（Tablespace ）操作	RM_SMGR, RM_DBASE, RM_TBLSPC, RM_RELMAP
复制和热备（replication and hot standby）操作	RM_STANDBY, RM_REPLORIGIN, RM_GENERIC_ID, RM_LOGICALMSG_ID

【源码】src/include/access/xlog_basic.h

```
/*
 * The overall layout of an XLOG record is:
 *      Fixed-size header (XLogRecord struct)
 *      XLogRecordBlockHeader struct
 *      XLogRecordBlockHeader struct
 *      ...
 *      XLogRecordDataHeader[Short|Long] struct
 *      block data
 *      block data
 *      ...
 *      main data
 *
 * There can be zero or more XLogRecordBlockHeaders, and 0 or more bytes of
 * rmgr-specific data not associated with a block.  XLogRecord structs
 * always start on MAXALIGN boundaries in the WAL files, but the rest of
 * the fields are not aligned.
 *
 * The XLogRecordBlockHeader, XLogRecordDataHeaderShort and
 * XLogRecordDataHeaderLong structs all begin with a single 'id' byte. It's
 * used to distinguish between block references, and the main data structs.
 */
typedef struct XLogRecord {
    uint32 xl_tot_len; /* total len of entire record */
    uint32 xl_term;
    TransactionId xl_xid; /* xact id */
    XLogRecPtr xl_prev;   /* ptr to previous record in log */
    uint8 xl_info;        /* flag bits, see below */
    RmgrId xl_rmid;       /* resource manager for this record */
    uint2  xl_bucket_id;    /* stores bucket id */
    pg_crc32c xl_crc; /* CRC for this record */

    /* XLogRecordBlockHeaders and XLogRecordDataHeader follow, no padding */
} XLogRecord;
```

2. XLOG 记录的数据部分

XLOG 记录的数据部分又分为：头部分和数据部分。

XLOG 记录数据部分的头部分包含 0 或多个 XLogRecordBlockHeader 结构体和 0 或 1 个 XLogRecordDataHeaderShort 或 XLogRecordDataHeaderLong 结构体；且要包含至少 1 个这些结构体。当 XLOG 记录存储整页镜像（full-page image）（即备份块 backup block）时，XLogRecordBlockHeader 中包括 XLogRecordBlockImageHeader。如图 8.8 所示。

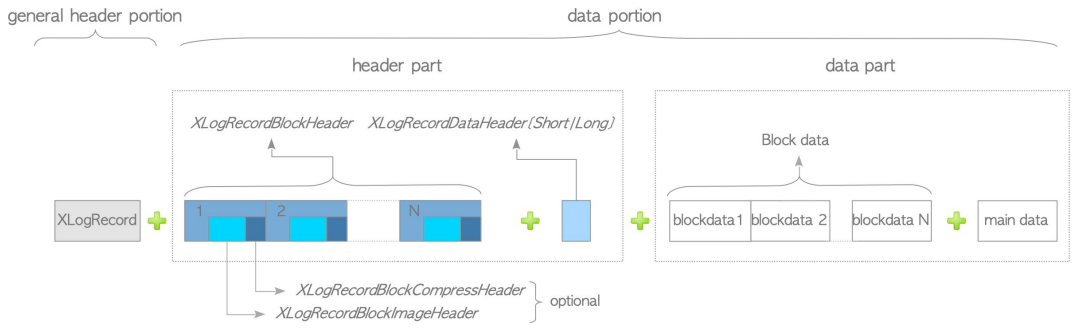

图 8.8 XLOG 记录的数据结构

- 结构体 XLogRecordBlockHeader

【源码】src/include/access/xlogrecord.h

```
/*
 * Header info for block data appended to an XLOG record.
 *
 * Note that we don't attempt to align the XLogRecordBlockHeader struct!
 * So, the struct must be copied to aligned local storage before use.
 * 'data_length' is the length of the payload data associated with this,
 * and includes the possible full-page image, and rmgr-specific data. It
 * does not include the XLogRecordBlockHeader struct itself.
 */
typedef struct XLogRecordBlockHeader {
    uint8 id;           /* block reference ID */
    uint8 fork_flags;   /* fork within the relation, and flags */
    uint16 data_length; /* number of payload bytes (not including page
                         * image) */

    /* If BKPBLOCK_HAS_IMAGE, an XLogRecordBlockImageHeader struct follows */
    /* If !BKPBLOCK_SAME_REL is not set, a RelFileNode follows */
    /* BlockNumber follows */
} XLogRecordBlockHeader;
```

- 结构体 XLogRecordBlockImageHeader

【源码】src/include/access/xlogrecord.h

```

/*
 * Additional header information when a full-page image is included
 * (i.e. when BKPBLOCK_HAS_IMAGE is set).
 *
 * As a trivial form of data compression, the XLOG code is aware that
 * PG data pages usually contain an unused "hole" in the middle, which
 * contains only zero bytes.  If hole_length > 0 then we have removed
 * such a "hole" from the stored data (and it's not counted in the
 * XLOG record's CRC, either).  Hence, the amount of block data actually
 * present is BLCKSZ - hole_length bytes.
```

```
 */
typedef struct XLogRecordBlockImageHeader {
    uint16 hole_offset; /* number of bytes before "hole" */
    uint16 hole_length; /* number of bytes in "hole" */
} XLogRecordBlockImageHeader;
```

- 结构体 XLogRecordDataHeaderShort 和 XLogRecordDataHeaderLong

【源码】src/include/access/xlogrecord.h

```
/*
 * XLogRecordDataHeaderShort/Long are used for the "main data" portion of
 * the record. If the length of the data is less than 256 bytes, the short
 * form is used, with a single byte to hold the length. Otherwise the long
 * form is used.
 *
 * (These structs are currently not used in the code, they are here just for
 * documentation purposes).
 */
typedef struct XLogRecordDataHeaderShort {
    uint8 id;          /* XLR_BLOCK_ID_DATA_SHORT */
    uint8 data_length; /* number of payload bytes */
} XLogRecordDataHeaderShort;

typedef struct XLogRecordDataHeaderLong {
    uint8 id; /* XLR_BLOCK_ID_DATA_LONG */
              /* followed by uint32 data_length, unaligned */
} XLogRecordDataHeaderLong;
```

XLOG 记录的数据部分由 0 或多个块数据和 0 或 1 个主数据组成，但两者至少要有一个。它们的头部分分别对应于 XLogRecordBlockHeader 和 XLogRecordDataHeader。

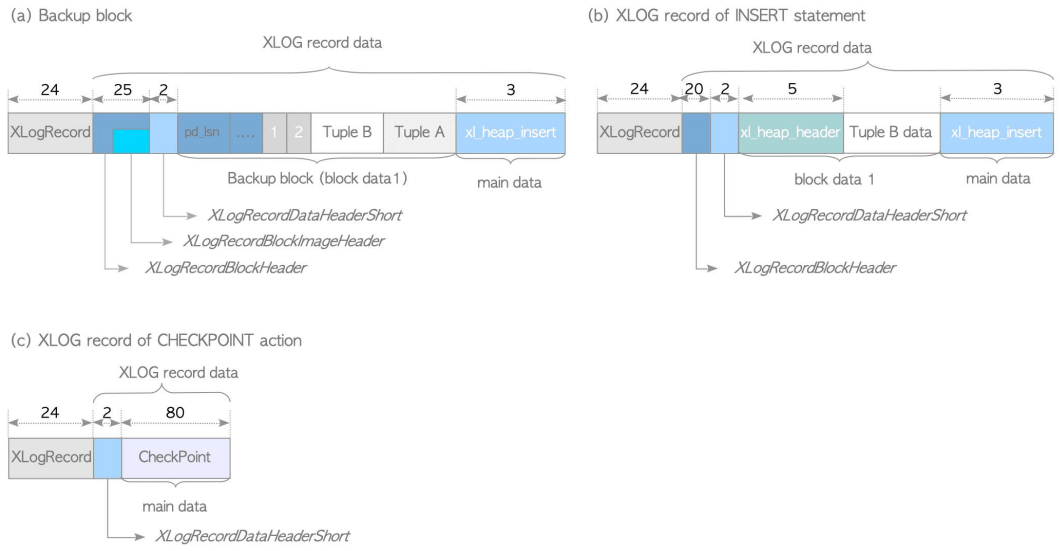

图 8.9 XLOG 记录数据部分的不同类型

- 备份块（backup block）

由 INSERT 语句引发的备份块如图 8.9(a)所示，由以下 5 部分组成：

(1) 结构体 XLogRecord（通用头部分）

(2) 结构体 XLogRecordBlockHeader，包含一个 XLogRecordBlockImageHeader

(3) 结构体 XLogRecordDataHeaderShort

(4) 备份块（块数据）

(5) 结构体 xl_heap_insert（主数据）

XLogRecordBlockHeader 包含用于在数据库集群中定位区块的变量。

XLogRecordImageHeader 包含此块的长度和偏移号。

XLogRecordDataHeaderShort 存储 xl_heap_insert 结构的长度，该结构是记录的主数据。

结构体 xl_heap_insert：

【源码】src/include/access/htup.h

```
/* This is what we need to know about insert */
typedef struct xl_heap_insert {
    OffsetNumber offnum; /* inserted tuple's offset */
    uint8 flags;

    /* xl_heap_header & TUPLE DATA in backup block 0 */
} xl_heap_insert;
```

- 非备份块（non-backup block）

由 INSERT 语句引发的非备份块如图 8.9(b)所示，由以下 5 部分组成：

(1) 结构体 XLogRecord（通用头部分）

(2) 结构体 XLogRecordBlockHeader

(3) 结构体 XLogRecordDataHeaderShort

(4) 一条被插入的元组（结构体 xl_heap_header+完整的插入数据）

(5) 结构体 xl_heap_insert（主数据）

XLogRecordBlockHeader 包含用于在数据库集群中定位区块的变量，用于指定该元组被插入哪个块中，以及要插入元组的数据部分的长度。XLogRecordDataHeaderShort 包含新的 xl_heap_insert 结构的长度。新的 xl_heap_insert 只包含两个值：新插入元组在块内的偏移量，以及可见性标志。

结构体 xl_heap_header：

【源码】src/include/access/htup.h

```
/*
 * We don't store the whole fixed part (HeapTupleHeaderData) of an inserted
 * or updated tuple in WAL; we can save a few bytes by reconstructing the
 * fields that are available elsewhere in the WAL record, or perhaps just
 * plain needn't be reconstructed.  These are the fields we must store.
 * NOTE: t_hoff could be recomputed, but we may as well store it because
 * it will come for free due to alignment considerations.
 */
typedef struct xl_heap_header {
    uint16 t_infomask2;
    uint16 t_infomask;
    uint8 t_hoff;
} xl_heap_header;
```

- 检查点记录（checkpoint）

检查点记录如图 8.9(c)所示，由 3 个数据结构组成：

(1) 结构体 XLogRecord（通用头部分）

(2) 结构体 XLogRecordDataHeaderShort，包含主数据长度

(3) 结构体 CheckPoint（主数据）

结构体 CheckPoint：

【源码】src/include/catalog/pg_control.h

```
/*
 * Body of CheckPoint XLOG records.  This is declared here because we keep
 * a copy of the latest one in pg_control for possible disaster recovery.
 * Changing this struct requires a PG_CONTROL_VERSION bump.
 */
typedef struct CheckPoint {
    XLogRecPtr        redo;            /* next RecPtr available when we began to
                                        * create CheckPoint (i.e. REDO start point) */
    TimeLineID        ThisTimeLineID;  /* current TLI */
    bool              fullPageWrites;  /* current full_page_writes */
    TransactionId     nextXid;         /* next free XID */
    Oid               nextOid;         /* next free OID */
    MultiXactId       nextMulti;       /* next free MultiXactId */
    MultiXactOffset   nextMultiOffset; /* next free MultiXact offset */
    TransactionId     oldestXid;       /* cluster-wide minimum datfrozenxid */
    Oid               oldestXidDB;     /* database with minimum datfrozenxid */
    pg_time_t         time;            /* time stamp of checkpoint */
    XLogSegNo         remove_seg;      /*the xlog segno we keep during this checkpoint*/
    /*
     * Oldest XID still running. This is only needed to initialize hot standby
     * mode from an online checkpoint, so we only bother calculating this for
     * online checkpoints and only when wal_level is hot_standby. Otherwise
     * it's set to InvalidTransactionId.
     */
    TransactionId     oldestActiveXid;
} CheckPoint;
```

11.3.3 日志写入过程

以执行一条 INSERT 语句为例，解释 XLOG 记录的写入过程：

创建表 table_a：

```
railway=# CREATE TABLE table_a ( a CHAR(1) );
CREATE TABLE
```

执行 INSERT 语句，插入一条元组：

```
railway=# INSERT INTO table_a VALUES ('A');
INSERT 0 1
```

openGauss 处理 INSERT 语句的执行，会调用函数 exec_simple_query()，该函数是的作用是将传入的 SQL 语句作为一个简单查询执行，适用于执行那些不返回结果集的查询，例如 INSERT、UPDATE、DELETE 等语句。在执行查询时，该函数将阻塞直到查询完成并返回一个表示执行结果的状态码。该函数的源代码位于：

【源码】src/gausskernel/process/tcop/postgres.cpp

```
static void exec_simple_query(const char* query_string, MessageType messageType,
    StringInfo msg = NULL)
{
...
}
```

在函数 exec_simple_query 的执行过程中，将直接或间接调用以下与日志写入相关的函数：

```
exec_simple_query() @postgres.cpp
  (1) ExtendCLOG() @clog.cpp              /* 将当前事务的状态"IN_PROGRESS" 写入 CLOG */
  (2) heap_insert()@heapam.cpp            /* 插入元组，创建一条 XLOG 记录并调用函数 XLogInsert */
  (3) XLogInsert() @xloginsert.cpp      /* 将插入元组的 XLOG 记录写入 WAL 缓冲区，更新页面的 pd_lsn */
  (4) finish_xact_command() @postgres.cpp      /* 执行提交*/
      XLogInsert() @xloginsert.cpp        /* 将该提交行为的 XLOG 记录写入 WAL 缓冲区 */
  (5) XLogWrite() @xlog.cpp                /* 将 WAL 缓冲区中所有的 xlog 写入 WAL 文件 */
  (6) TransactionIdCommitTree() @transam.cpp
                                /* 在 CLOG 中将事务状态由"IN_PROGRESS"改为"COMMITTED" on the CLOG */
```

上述这些函数调用完成了日志写入的主要步骤，如图 8.10 所示，解释如下：

(1) 函数 ExtendCLOG()在内存 CLOG 中将当前事务状态置为 IN_PROGRESS

(2) 函数 heap_insert()在共享缓冲器插入元组，创建 XLOG 记录，调用函数 XLogInsert()

(3) 函数 XLogInsert()将 heap_insert()函数中创建的 XLOG 记录写入 WAL 缓冲区的 LSN_1 位置，将插入元组的页面的 pd_lsn 由 LSN_0 更新为 LSN_1

(4) 函数 finish_xact_command()提交当前事务，创建提交动作的 XLOG 记录，调用函数 XLogInsert()，将该 XLOG 记录写入 WAL 缓冲区的 LSN_2 位置

(5) 函数 XLogWrite()将所有在 WAL 缓冲区的 XLOG 记录写入 WAL 段文件

(6) 函数 TransactionIdCommitTree() 将 CLOG 中该事务的状态由 IN_PROGRESS 置为 COMMITTED

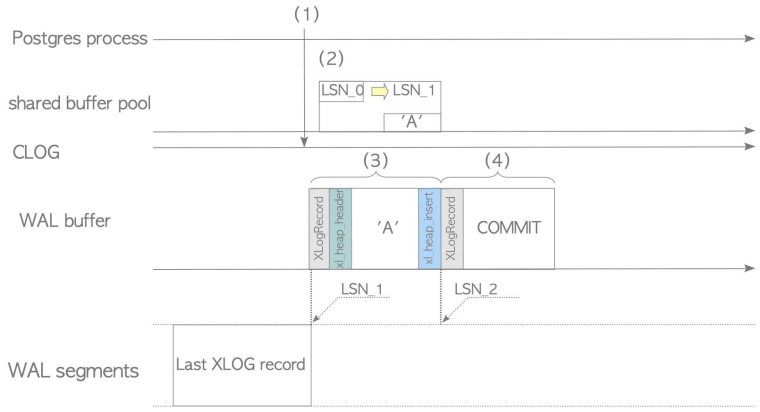

图 8.10 XLOG 记录写入过程

能够引起 XLOG 记录写入 WAL 段文件的操作包括:

- 一个事务提交（commit）或终止（abort）
- WAL 缓冲区满
- WAL 写线程周期性写入

日志写线程即 WalWriter 线程，是一个后台背景线程，负责周期性地检查 WAL 缓冲区，将所有未写入的 XLOG 记录写入 WAL 段文件，其目的是避免 XLOG 记录的大量集中写入。

11.3.4 检查点机制

当下列情形之一发生时，检查点后台线程 checkpointer 会执行检查点操作:

- 自上次检查点执行时间已超过 checkpoint_timeout 设置（默认 300 秒）。
- pg_xlog 目录中的 WAL 段文件总大小超过了 max_wal_size 设置（默认 1GB，64 个文件）。
- 以 fast 模式关闭数据库服务器。
- 手动执行 CHECKPOINT 命令。

下面介绍检查点的执行流程和保存当前检查点元数据的 pg_control 文件。

1. 检查点的执行流程

执行检查点有两个目的:

- 为数据库恢复做准备
- 将共享缓冲区中的脏页写盘

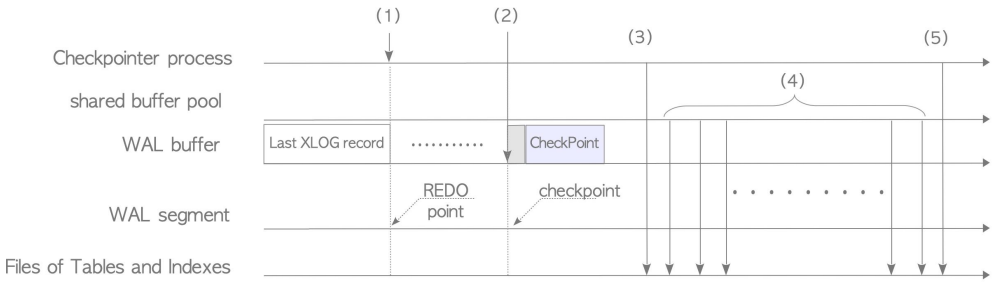

图 8.11 XLOG 记录写入过程

(1) 检查点过程启动后，会将重做点存储在内存中。重做点是最新的检查点开始时 XLOG 记录写入的位置，也是数据库恢复的起点。

(2) 将此检查点对应的 XLOG 记录（即检查点记录）写入 WAL 缓冲区。该记录的数据部分由 CheckPoint 结构体定义，其中包括步骤(1)中重做点的位置。此外，写入检查点记录的位置按照字面意思也称作检查点。

(3) 刷新共享缓冲区中的所有数据（例如 CLOG 的内容等）到磁盘。

(4) 更新 pg_control 文件，该文件包含一些基本信息，例如检查点记录的写入位置。

从数据库恢复的角度看，检查点过程在日志文件中创建了一个包含重做点的检查点记录，将检查点位置和更多的信息存储到 pg_control 文件中。数据库恢复时，通过 pg_control 文件获取检查点记录内的重做点，从该重做点开始重做 WAL 日志。

2. pg_control 文件

pg_control 文件包含检查点的基本信息，它对于数据库恢复是必须的。如果它损坏了，系统将无法得知恢复起点，故无法进行恢复。

pg_control 文件中存储了 3 项与数据库恢复相关的信息：

- State：最新检查点开始时数据库服务器的状态。共有 7 种状态：start up 表示数据库正在启动；shutdown 表示数据库正常关闭；in production 表示数据库正在运行，等等。
- Latest checkpoint location：最新检查点在日志文件中的 LSN 位置。
- Prior checkpoint location：前一个检查点在日志文件中的 LSN 位置。

11.3.5 数据库恢复

openGauss 实现了基于 REDO 日志的恢复功能。如果数据库服务器崩溃，openGauss 会从重做点按顺序重做 WAL 段文件中的 XLOG 记录，以恢复数据库。

1. 如何启动恢复过程

当 openGauss 启动时，它会首先读取 pg_control 文件，开始进行恢复过程。如图 8.12 所示。

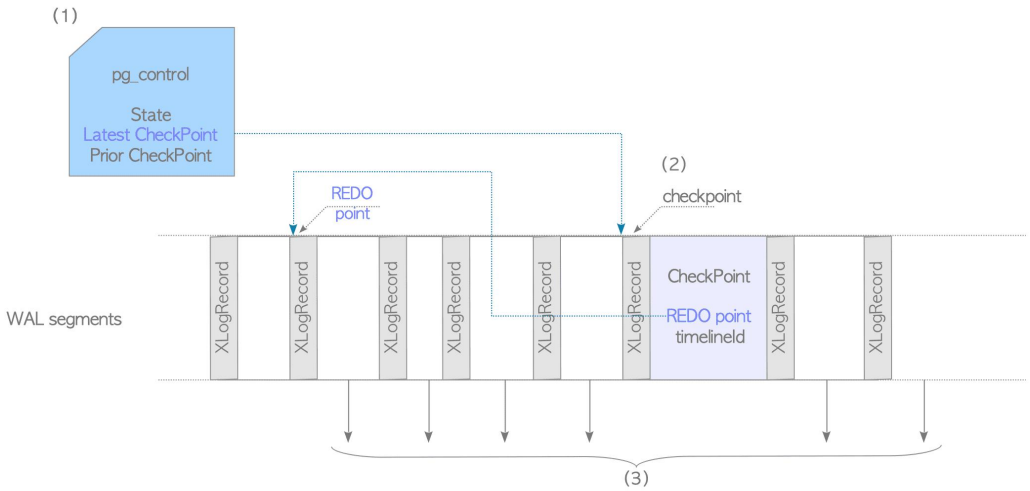

图 8.12 数据库恢复过程

(1) 数据库服务器启动时读取 pg_control 文件内容。如果 state 为 in production，将进入恢复模式（recovery-mode），因为这意味着数据库没有正常停止；如果为 shutdown，将进入正常启动模式。

(2) 从相应的 WAL 段文件中读取最新的检查点记录（pg_control 文件中保存了最新的检查点记录在 WAL 段文件中的位置），从该记录中获取到重做点。如果最新的检查点记录无效（invalid），将读取前一个检查点记录。如果两个记录都不可读，将放弃恢复。

(3) 使用合适的资源管理器从重做点开始按顺序读取和重做 XLOG 记录，直到最新 WAL 段文件的最后位置。当遇到备份块时，无论其 LSN 如何，都会用其覆盖相应表的页面；否则仅当此 XLOG 记录的 LSN 大于相应页面的 pd_lsn 时，才会重做该 XLOG 记录。

2. LSN 的比较

为什么应该比较非备份块 XLOG 记录的 LSN 与相应页面的 pd_lsn 呢？这里使用一个示例来解释比较两个 LSN 的必要性。注意，这里省略了 WAL 缓冲区以简化描述。

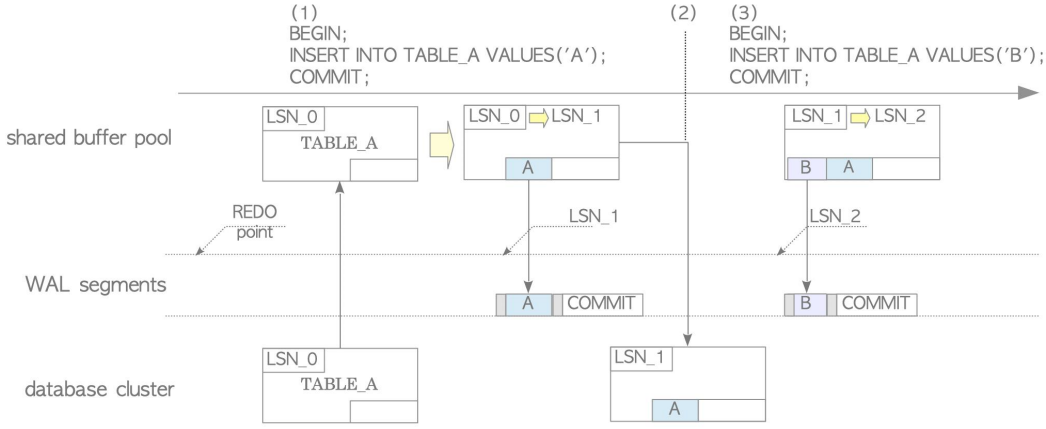

图 8.13 在后台写入线程工作过程中插入数据

(1) 向 TABLE_A 中插入一个元组，并在 LSN_1 处写入一条 XLOG 记录。

(2) 后台写入线程（BgWriter）将 TABLE_A 的页面写入磁盘。此时，该页面的 pd_lsn 为 LSN_1。

(3) 向 TABLE_A 中再插入一个元组，并在 LSN_2 处写入一条 XLOG 记录，修改后的页面还没有写入磁盘。

这时，使用 immediate 模式关闭数据库服务器（这会引起数据库服务器重启时进入恢复过程）；然后再次启动数据库服务器，恢复过程如图 8.14 所示。

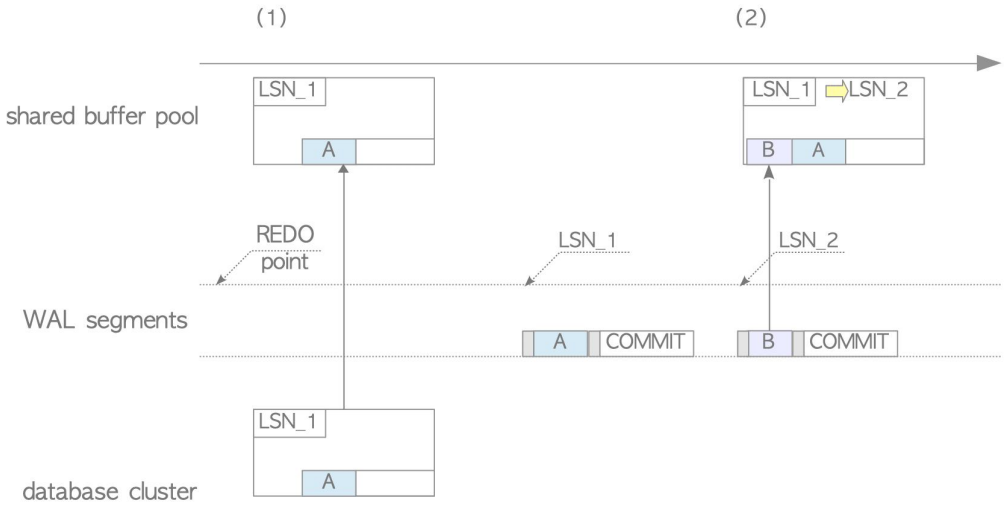

图 8.14 数据库恢复示例

(1) 加载第 1 个 XLOG 记录和 TABLE_A 的页面，但不重做该 XLOG 记录，因为该 XLOG 记录的 LSN 不大于 TABLE_A 的 LSN（都是 LSN_1）。

(2) 会重做第 2 个 XLOG 记录，因为该记录的 LSN（LSN_2）大于当前 TABLE_A 的 LSN（LSN_1）。

可以看出，如果非备份块的重做顺序不正确或者被多次重做，将导致数据库中数据的不一致状态。也就是说，非备份块的重做操作不是幂等的（idempotent）。因此，为了保留正确的重做顺序，当且仅当非备份块 XLOG 记录的 LSN 大于相应页面的 pd_lsn 时，才重放该 XLOG 记录。另一方面，由于备份块的重做操作是幂等的，备份块可以重做任意多次，而不管其 LSN 如何。（注：在数学中，幂等是指函数执行任意多次所产生的效果与执行一次相同，即$f\big(f(x)\big)=f(x)$。）

11.3.6 数据库备份与 PITR 恢复

数据库备份是数据库管理中的一个重要方面，它用于保护数据库免受数据丢失和损坏的风险。数据库备份可以分为物理备份和逻辑备份两种方式：

物理备份：物理备份是指将数据库的物理文件完全复制到备份存储设备上的过程。物理备份通常包括数据库的所有数据文件、控制文件、日志文件、配置文件等，它们在备份过程中被复制到备份设备上。物理备份速度较快，恢复时也比较方便，但是备份文件较大，不便于跨平台恢复。

逻辑备份：逻辑备份是指将数据库中的逻辑数据和对象导出到备份文件中的过程。逻辑备份通常包括数据库中的表、视图、存储过程、触发器等逻辑对象，以及这些对象中的数据。逻辑备份的备份文件相对较小，便于跨平台恢复，但是备份和恢复的速度相对较慢。

物理备份适合备份整个数据库以及大型数据库，而逻辑备份适合备份指定的逻辑对象和数据。在实际应用中，根据实际需要可以综合使用物理备份和逻辑备份，以达到最佳的备份效果。

openGauss 中的 PITR（Point-In-Time Recovery）恢复实际上是物理备份和 WAL 日志恢复的结合。在 openGauss 中，物理备份是数据库集群在运行时的一个快照，也称为基础备份（base backup）。

PITR 恢复功能使用基础备份和归档日志数据可将数据库恢复到基础备份之后的任意时间点。PITR 的工作原理是在基础备份上重做归档日志中的 XLOG 记录。PITR 可以帮助数据库管理将数据库恢复到某个误操作之前的状态，比如一个误操作删除了表中的所有数据，可以通过 PITR 将数据库恢复到执行删除表数据操作之前的时刻。

进行 PITR 恢复，需要具备：

- 基于物理备份的全量数据文件
- 基于归档的 WAL 日志文件

PITR 恢复流程：

(1) 将物理备份的文件替换目标数据库目录。
(2) 删除数据库目录下 pg_xlog 中的所有文件。
(3) 将归档的 WAL 日志文件复制到 pg_xlog 文件中（通过配置 recovery.conf 恢复命令文件中的 restore_command 项，此步骤可以省略）。
(4) 在数据库目录下创建恢复命令文件 recovery.conf，指定数据库恢复的程度。
(5) 启动数据库。
(6) 连接数据库，查看是否恢复到预期状态。
(7) 若已经恢复到预期状态，通过 pg_xlog_replay_resume()指令使主节点对外提供服务。

11.4 实验步骤

11.4.1 查看 WAL 日志文件

1. 查看默认日志大小

```
railway=# SHOW wal_segment_size;
 wal_segment_size
------------------
 16MB
(1 row)
```

2. 查看当前日志的 LSN

```
railway=# SELECT pg_current_xlog_insert_location();
 pg_current_xlog_insert_location
---------------------------------
 0/1497E7B0
(1 row)
```

3. 查看当前日志文件名

```
railway=# SELECT pg_xlogfile_name('0/1497E7B0');
     pg_xlogfile_name
--------------------------
 000000010000000000000014
(1 row)
```

4. 查看当前日志文件名和 LSN 在 WAL 段文件中的字节偏移量

```
railway=# SELECT pg_xlogfile_name_offset('0/1497E7B0');
      pg_xlogfile_name_offset
-----------------------------------
```

```
 (0000000100000000000000014,9955248)
(1 row)
```

5. 查看 openGauss 的 WAL 日志文件目录
找到了“000000010000000000000014”WAL 段文件。

```
[dblab@eduog pg_xlog]$ ll $GAUSSHOME/data/pg_xlog
total 305M
-rw------- 1 dblab dbgrp  16M Feb  1 19:19 000000010000000000000003
-rw------- 1 dblab dbgrp  16M Feb  3 02:44 000000010000000000000004
-rw------- 1 dblab dbgrp  16M Feb  5 04:00 000000010000000000000005
-rw------- 1 dblab dbgrp  16M Feb 10 23:21 000000010000000000000006
-rw------- 1 dblab dbgrp  16M Feb 11 04:20 000000010000000000000007
-rw------- 1 dblab dbgrp  16M Feb 12 08:12 000000010000000000000008
-rw------- 1 dblab dbgrp  16M Feb 13 18:59 000000010000000000000009
-rw------- 1 dblab dbgrp  16M Feb 14 12:35 00000001000000000000000A
-rw------- 1 dblab dbgrp  16M Feb 14 14:53 00000001000000000000000B
-rw------- 1 dblab dbgrp  16M Feb 14 17:03 00000001000000000000000C
-rw------- 1 dblab dbgrp  16M Feb 15 15:51 00000001000000000000000D
-rw------- 1 dblab dbgrp  16M Feb 15 19:09 00000001000000000000000E
-rw------- 1 dblab dbgrp  16M Feb 16 14:09 00000001000000000000000F
-rw------- 1 dblab dbgrp  16M Feb 17 18:01 000000010000000000000010
-rw------- 1 dblab dbgrp  16M Feb 18 16:19 000000010000000000000011
-rw------- 1 dblab dbgrp  16M Feb 19 20:01 000000010000000000000012
-rw------- 1 dblab dbgrp  16M Feb 20 23:54 000000010000000000000013
-rw------- 1 dblab dbgrp  16M Feb 21 14:27 000000010000000000000014
-rw------- 1 dblab dbgrp  16M Jan 30 20:40 000000010000000000000015
drwx------ 2 dblab dbgrp 4.0K Jan 29 15:07 archive_status
```

11.4.2 验证数据库恢复

1. 通过 immediate 关闭模式触发数据库恢复
停止数据库服务器的命令为:

```
gs_ctl stop -D DATADIR -m shutdown-mode
```

其中，-D 选项指定数据文件目录为 DATADIR，-m 选项指定关闭模式（shutdown-mode）。关闭模式有两种:

- 快速（fast）：快速关闭数据库，断开客户端的连接，让当前未提交事务回滚，然后正常关闭数据库。
- 立即（immediate）：立即关闭数据库，立即停止数据库进程，直接退出；下次启动时会进行数据库恢复。

确保数据库服务器已启动。
新建一个 SSH 会话，用 gsql 客户端连接数据库服务器。执行下面语句:

```
railway=# CREATE TABLE table_b (id INT, name VARCHAR(5));
CREATE TABLE
railway=# BEGIN;
BEGIN
```

```
railway=# INSERT INTO table_b VALUES (1, 'abc');
INSERT 0 1
```

上述语句开启一个事务，执行一条 INSERT 语句，但没有提交该事务。

用 fast 模式关闭数据库服务器：

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl stop -D $GAUSSHOME/data -m fast
waiting for server to shut down..... done
server stopped
```

查看 gaussdb 进程输出日志文件（位于$GAUSSHOME/data/pg_log 目录），其中部分日志行如下：

```
LOG:  received fast shutdown request
LOG:  aborting any active transactions
FATAL:  terminating connection due to administrator command
LOG:  shutting down
LOG:  will do full checkpoint, need flush 0 pages.
LOG:  pagewriter thread shut down, id is 2
LOG:  pagewriter thread shut down, id is 1
LOG:  pagewriter thread shut down, id is 4
LOG:  pagewriter thread shut down, id is 3
LOG:  pagewriter thread shut down, id is 0
LOG:  will update control file (create checkpoint), shutdown:1
LOG:  database system is shut down
LOG:  Gaussdb exit(0)
```

重启数据库服务器：

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl start -D $GAUSSHOME/data -Z single_node
-l logfile
```

查看表数据，发现未提交的事务被回滚，INSERT 语句插入的数据没有出现在表中。

```
railway=# SELECT * FROM table_b;
 id | name
----+------
 (0 row)
```

下面通过实验观察 immediate 关闭模式。

新建一个 SSH 会话，用 gsql 客户端连接数据库服务器。执行下面语句：

```
railway=# INSERT INTO table_b VALUES (1, 'abc');
INSERT 0 1
```

用 immediate 模式关闭数据库服务器：

```
dblab@eduog openGauss-server-v3.0.0]$ gs_ctl stop -D $GAUSSHOME/data -m immediate
waiting for server to shut down..... done
server stopped
```

查看 gaussdb 进程的输出日志文件（位于$GAUSSHOME/data/pg_log 目录），其中部分日志行如下：

```
LOG:  received immediate shutdown request
LOG:  Gaussdb exit(0)
```

重启数据库服务器：

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl start -D $GAUSSHOME/data -Z single_node
```

```
-l logfile
```

查看 gaussdb 进程输出日志文件（位于$GAUSSHOME/data/pg_log 目录），其中部分日志行如下：

```
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/15E87F98
[REDO] LOG:  pagerepair started
[BACKEND] LOG:  redo done at 0/15E88170, end at 0/15E881B8
[REDO]      LOG:            [PR]:      Recoverying      elapsed:      100631      us,
redoTotalBytes:544,EndRecPtr:367559096,      redoStartPtr:367558552,speed:0      MB/s,
totalTime:100631
[BACKEND] LOG:  last completed transaction was at log time 2023-02-23 00:25:32.978453+08
LOG: database system is ready to accept connections
[BACKEND] LOG:  database first startup and recovery finish,so do checkpointer
```

查看表数据，发现 INSERT 语句插入的数据在表中，表明数据库已恢复到正常状态。

```
railway=# SELECT * FROM table_b;
 id | name
----+------
  1 | abc
(1 row)
```

2. 查看 pg_control 文件

   pg_control 文件位于在数据目录的 global 子目录中。

```
[dblab@eduog openGauss-server-v3.0.0]$ cd $GAUSSHOME/data/global
[dblab@eduog global]$ ll pg_control
-rw------- 1 dblab dbgrp 8.0K Feb 22 21:10 pg_control
```

   使用 pg_controldata 命令可查看 pg_control 文件内容。

```
[dblab@eduog global]$ pg_controldata $GAUSSHOME/data
pg_control version number:            923
Catalog version number:               201611171
Database system identifier:           18020660039215399202
Database cluster state:               in production
pg_control last modified:             Wed 22 Feb 2023 09:11:05 PM CST
Latest checkpoint location:           0/15E7A2C8
Prior checkpoint location:            0/15E7A1A8
Latest checkpoint's REDO location:    0/15E7A248
Latest checkpoint's TimeLineID:       1
Latest checkpoint's full_page_writes: off
Latest checkpoint's NextXID:          31334
Latest checkpoint's NextOID:          82069
Latest checkpoint's NextMultiXactId:  2
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        12772
Latest checkpoint's oldestXID's DB:   15552
Latest checkpoint's oldestActiveXID:  31334
Latest checkpoint's remove lsn:          0/5
```

```
Time of latest checkpoint:            Wed 22 Feb 2023 09:11:05 PM CST
Minimum recovery ending location:     0/0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
Current wal_level setting:            hot_standby
Current max_connections setting:      250
Current max_prepared_xacts setting:   200
Current max_locks_per_xact setting:   256
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Database system TimeLine:             55
```

11.4.3 分析代码：数据库恢复过程

openGauss 数据库发生非正常停止后，重启时会进入到恢复模式。这时，会自动重做 XLOG 日志记录。恢复过程调用的主要函数是 StartupXLOG，该函数非常长，我们只关注关键步骤。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
/*
 * This must be called ONCE during postmaster or standalone-backend startup
 */
void StartupXLOG(void)
{
    XLogCtlInsert *Insert = NULL;
    CheckPoint checkPoint;
    CheckPointNew checkPointNew;   /* to adapt update and not to modify the storage format */
    CheckPointPlus checkPointPlus; /* to adapt update and not to modify the storage format for global
clean */
    CheckPointUndo checkPointUndo;
    uint32 recordLen = 0;
    bool wasShutdown = false;
    bool DBStateShutdown = false;
    bool reachedStopPoint = false;
    bool haveBackupLabel = false;
```

```
    bool haveTblspcMap = false;
    XLogRecPtr RecPtr, checkPointLoc, EndOfLog;
    XLogSegNo endLogSegNo;
   XLogRecord *record = NULL;
... 省略
```

调用函数 ReadControlFile，读取 pg_control 文件。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
    /*
     * Read control file and check XLOG status looks valid.
     *
     * Note: in most control paths, *ControlFile is already valid and we need
     * not do ReadControlFile() here, but might as well do it to be sure.
     */
    ReadControlFile();
```

输出数据库的 timeline，每次执行恢复时，timeline 会增加 1。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
    /* description: timeline > MAX_INT32 */
    if (IsUnderPostmaster) {
        t_thrd.shemem_ptr_cxt.ControlFile->timeline =
           t_thrd.shemem_ptr_cxt.ControlFile->timeline + 1;
        ereport(LOG, (errmsg("database system timeline: %u",
            t_thrd.shemem_ptr_cxt.ControlFile->timeline)));
        if (t_thrd.shemem_ptr_cxt.ControlFile->timeline == 0) {
            t_thrd.shemem_ptr_cxt.ControlFile->timeline = 1;
        }
    }
```

输出上次数据库服务器非正常停止时数据库的状态。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
     if (t_thrd.shemem_ptr_cxt.ControlFile->state == DB_SHUTDOWNED) {
        DBStateShutdown = true;
        AbnormalShutdown = false;
        ereport(LOG,
               (errmsg("database system was shut down at %s",
                    str_time(t_thrd.shemem_ptr_cxt.ControlFile->time))));
      } else if (t_thrd.shemem_ptr_cxt.ControlFile->state == DB_SHUTDOWNED_IN_RECOVERY) {
        DBStateShutdown = true;
        ereport(LOG, (errmsg("database system was shut down in recovery at %s",
                             str_time(t_thrd.shemem_ptr_cxt.ControlFile->time))));
    } else if (t_thrd.shemem_ptr_cxt.ControlFile->state == DB_SHUTDOWNING) {
        ereport(LOG, (errmsg("database system shutdown was interrupted; last known up at %s",
                             str_time(t_thrd.shemem_ptr_cxt.ControlFile->time))));
    } else if (t_thrd.shemem_ptr_cxt.ControlFile->state == DB_IN_CRASH_RECOVERY) {
        ereport(LOG, (errmsg("database system was interrupted while in recovery at %s",
                             str_time(t_thrd.shemem_ptr_cxt.ControlFile->time)),
```

```
                    errhint("This probably means that some data is corrupted and"
                            " you will have to use the last backup for recovery.")));
    } else if (t_thrd.shemem_ptr_cxt.ControlFile->state == DB_IN_ARCHIVE_RECOVERY) {
        ereport(LOG, (errmsg("database system was interrupted while in recovery at log time %s",
                             str_time(t_thrd.shemem_ptr_cxt.ControlFile->checkPointCopy.time)),
                    errhint("If this has occurred more than once some data might be corrupted"
                            " and you might need to choose an earlier recovery target.")));
    } else if (t_thrd.shemem_ptr_cxt.ControlFile->state == DB_IN_PRODUCTION) {
        ereport(LOG, (errmsg("database system was interrupted; last known up at %s",
                             str_time(t_thrd.shemem_ptr_cxt.ControlFile->time))));
    }
```

创建 XLogReaderState 对象。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
    xlogreader = XLogReaderAllocate(&XLogPageRead, &readprivate);
    if (xlogreader == NULL) {
        ereport(ERROR, (errcode(ERRCODE_OUT_OF_MEMORY), errmsg("out of memory"),
                        errdetail("Failed while allocating an XLog reading processor")));
    }
    xlogreader->system_identifier = t_thrd.shemem_ptr_cxt.ControlFile->system_identifier;
```

调用函数 ReadCheckpointRecord，获取检查点记录。如果第一次没有获取到（record 为 null），就使用控制文件 ControlFile 中的 prevCheckPoint 来获取。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
    /*
     * Get the last valid checkpoint record.  If the latest one according
     * to pg_control is broken, try the next-to-last one.
     */
    checkPointLoc = t_thrd.shemem_ptr_cxt.ControlFile->checkPoint;
    t_thrd.xlog_cxt.RedoStartLSN =
          t_thrd.shemem_ptr_cxt.ControlFile->checkPointCopy.redo;
    g_instance.comm_cxt.predo_cxt.redoPf.redo_start_ptr = t_thrd.xlog_cxt.RedoStartLSN;
    record = ReadCheckpointRecord(xlogreader, checkPointLoc, 1);
    if (record != NULL) {
        ereport(DEBUG1,
                  (errmsg("checkpoint record is at %X/%X", (uint32)(checkPointLoc >> 32),
                          (uint32)checkPointLoc)));
    } else if (t_thrd.xlog_cxt.StandbyMode) {
        /*
         * The last valid checkpoint record required for a streaming
         * recovery exists in neither standby nor the primary.
         */
        ereport(PANIC, (errmsg("could not locate a valid checkpoint record")));
    } else {
        checkPointLoc = t_thrd.shemem_ptr_cxt.ControlFile->prevCheckPoint;
        record = ReadCheckpointRecord(xlogreader, checkPointLoc, 2);
```

```
            if (record != NULL) {
                ereport(LOG, (errmsg("using previous checkpoint record at %X/%X",
                     (uint32)(checkPointLoc >> 32), (uint32)checkPointLoc)));
                t_thrd.xlog_cxt.InRecovery = true; /* force recovery even if SHUTDOWNED */
            } else {
                ereport(PANIC, (errmsg("could not locate a valid checkpoint record")));
            }
        }
        rcm = memcpy_s(&checkPoint, sizeof(CheckPoint), XLogRecGetData(xlogreader),
            sizeof(CheckPoint));
        securec_check(rcm, "", "");

        recordLen = record->xl_tot_len;
        if (record->xl_tot_len == CHECKPOINTNEW_LEN) {
            rcm = memcpy_s(&checkPointNew, sizeof(checkPointNew), XLogRecGetData(xlogreader),
                           sizeof(checkPointNew));
            securec_check(rcm, "", "");
        } else if (record->xl_tot_len == CHECKPOINTPLUS_LEN) {
            rcm = memcpy_s(&checkPointPlus, sizeof(checkPointPlus), XLogRecGetData(xlogreader),
                           sizeof(checkPointPlus));
            securec_check(rcm, "", "");
        } else if (record->xl_tot_len == CHECKPOINTUNDO_LEN) {
            rcm = memcpy_s(&checkPointUndo, sizeof(checkPointUndo), XLogRecGetData(xlogreader),
                           sizeof(checkPointUndo));
            securec_check(rcm, "", "");
        }

        UpdateTermFromXLog(record->xl_term);

        wasShutdown = (record->xl_info == XLOG_CHECKPOINT_SHUTDOWN);
        wasCheckpoint = wasShutdown || (record->xl_info == XLOG_CHECKPOINT_ONLINE);
```

将 RecPtr 设置为检查点位置 checkPointLoc。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
t_thrd.xlog_cxt.LastRec = RecPtr = checkPointLoc;
ereport(LOG, (errmsg("redo record is at %X/%X; shutdown %s",
        (uint32)(checkPoint.redo >> 32), (uint32)checkPoint.redo,
        wasShutdown ? "TRUE" : "FALSE")));
```

判断是否需要恢复。如果数据库状态不是正常关闭（DB_SHUTDOWNED）就需要恢复。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
/*
 * Check whether we need to force recovery from WAL.  If it appears to
 * have been a clean shutdown and we did not have a recovery.conf file,
 * then assume no recovery needed.
```

```
     */
    if (XLByteLT(checkPoint.redo, RecPtr)) {
#ifdef ENABLE_MULTIPLE_NODES
        if (wasShutdown) {
            ereport(PANIC, (errmsg("invalid redo record in shutdown checkpoint")));
        }
#endif
        t_thrd.xlog_cxt.InRecovery = true;
    } else if (t_thrd.shemem_ptr_cxt.ControlFile->state != DB_SHUTDOWNED) {
        t_thrd.xlog_cxt.InRecovery = true;
    } else if (t_thrd.xlog_cxt.ArchiveRecoveryRequested) {
        /* force recovery due to presence of recovery.conf */
        t_thrd.xlog_cxt.InRecovery = true;
        if (ArchiveRecoveryByPending) {
            RecoveryByPending = true;
        }
    }
```

进入实际的重做日志（REDO）过程。判断数据库是否为 InArchiveRecovery，如果是，则将数据库状态标为 DB_IN_ARCHIVE_RECOVERY，否则标为 DB_IN_CRASH_RECOVERY 并输出：database system was not properly shut down; automatic recovery in progress。

调用函数 PrintCkpXctlControlFile 输出控制文件中的检查点信息，然后更新了 pg_control 中的检查点，该检查点就是进行恢复的起始点。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
     /* REDO */
    if (t_thrd.xlog_cxt.InRecovery) {
        /* use volatile pointer to prevent code rearrangement */
        volatile XLogCtlData *xlogctl = t_thrd.shemem_ptr_cxt.XLogCtl;

        /*
         * Update pg_control to show that we are recovering and to show the
         * selected checkpoint as the place we are starting from. We also mark
         * pg_control with any minimum recovery stop point obtained from a
         * backup history file.
         */
        dbstate_at_startup = t_thrd.shemem_ptr_cxt.ControlFile->state;
        if (t_thrd.xlog_cxt.InArchiveRecovery) {
            t_thrd.shemem_ptr_cxt.ControlFile->state = DB_IN_ARCHIVE_RECOVERY;
        } else {
            ereport(LOG, (errmsg("database system was not properly shut down; "
                                 "automatic recovery in progress")));
            t_thrd.shemem_ptr_cxt.ControlFile->state = DB_IN_CRASH_RECOVERY;
        }
        PrintCkpXctlControlFile(t_thrd.shemem_ptr_cxt.ControlFile->checkPoint,
               &(t_thrd.shemem_ptr_cxt.ControlFile->checkPointCopy),            checkPointLoc,
```

```
&checkPoint,
                              t_thrd.shemem_ptr_cxt.ControlFile->prevCheckPoint,
"StartupXLOG");
        t_thrd.shemem_ptr_cxt.ControlFile->prevCheckPoint =
           t_thrd.shemem_ptr_cxt.ControlFile->checkPoint;
        t_thrd.shemem_ptr_cxt.ControlFile->checkPoint = checkPointLoc;
        t_thrd.shemem_ptr_cxt.ControlFile->checkPointCopy = checkPoint;
        if (t_thrd.xlog_cxt.InArchiveRecovery) {
            /* initialize minRecoveryPoint if not set yet */
           if (XLByteLT(t_thrd.shemem_ptr_cxt.ControlFile->minRecoveryPoint, checkPoint.redo))
{
                t_thrd.shemem_ptr_cxt.ControlFile->minRecoveryPoint = checkPoint.redo;
           }
        }
```

调用函数 PrintCkpXctlControlFile。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
/*
 * Log checkPoint
 */
inline  void  PrintCkpXctlControlFile(XLogRecPtr  oldCkpLoc,  CheckPoint  *oldCkp,  XLogRecPtr
newCkpLoc, CheckPoint *newCkp,
                                      XLogRecPtr preCkpLoc, char *name)
{
    ereport(LOG,
            (errmsg("%s PrintCkpXctlControlFile: [checkPoint] oldCkpLoc:%X/%X, oldRedo:%X/%X,
                    newCkpLoc:%X/%X, "
                    "newRedo:%X/%X, preCkpLoc:%X/%X",
                    name, (uint32)(oldCkpLoc >> 32), (uint32)oldCkpLoc,
                  (uint32)(oldCkp->redo >> 32),
                    (uint32)oldCkp->redo, (uint32)(newCkpLoc >> 32),
                   (uint32)newCkpLoc, (uint32)(newCkp->redo >> 32),
                    (uint32)newCkp->redo, (uint32)(preCkpLoc >> 32), (uint32)preCkpLoc)));
}
```

获取第一条 XLOG 记录。如果 checkPoint.redo < RecPtr（即重做点小于检查点），则获取重做点对应的 XLOG 记录；否则，获取检查点之后的一条 XLOG 记录。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
        // Find the first record that logically follows the checkpoint --- it
        //  might physically precede it, though.
        if (XLByteLT(checkPoint.redo, RecPtr)) {
            /* back up to find the record */
            record = ReadRecord(xlogreader, checkPoint.redo, PANIC, false);
        } else {
            /* just have to read next record after CheckPoint */
            record = ReadRecord(xlogreader, InvalidXLogRecPtr, LOG, false);
```

```
        }
```

函数 ReadRecord 读取一条 XLOG 记录。在 for 循环中，调用函数 XLogReadRecord 进行 XLOG 记录读取。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
/*
 * Attempt to read an XLOG record.
 *
 * If RecPtr is not NULL, try to read a record at that position.  Otherwise
 * try to read a record just after the last one previously read.
 *
 * If no valid record is available, returns NULL, or fails if emode is PANIC.
 * (emode must be either PANIC, LOG). In standby mode, retries until a valid
 * record is available.
 *
 * The record is copied into readRecordBuf, so that on successful return,
 * the returned record pointer always points there.
 */
static XLogRecord *ReadRecord(XLogReaderState *xlogreader, XLogRecPtr RecPtr, int emode, bool
fetching_ckpt)
{
    XLogRecord *record = NULL;
    uint32 streamFailCount = 0;
    XLogPageReadPrivate *readprivate = (XLogPageReadPrivate *)xlogreader->private_data;

    /* Pass through parameters to XLogPageRead */
    readprivate->fetching_ckpt = fetching_ckpt;
    readprivate->emode = emode;
    readprivate->randAccess = !XLByteEQ(RecPtr, InvalidXLogRecPtr);

    /* This is the first try to read this page. */
    t_thrd.xlog_cxt.failedSources = 0;

    for (;;) {
        char *errormsg = NULL;

        record = XLogReadRecord(xlogreader, RecPtr, &errormsg);
        t_thrd.xlog_cxt.ReadRecPtr = xlogreader->ReadRecPtr;
        t_thrd.xlog_cxt.EndRecPtr = xlogreader->EndRecPtr;
        g_instance.comm_cxt.predo_cxt.redoPf.read_ptr = t_thrd.xlog_cxt.ReadRecPtr;
...
    }
```

函数 XLogReadRecord。

【源码】src/bin/pg_basebackup/xlogreader.cpp

```
/*
```

```
 * Attempt to read an XLOG record.
 *
 * If RecPtr is not NULL, try to read a record at that position.  Otherwise
 * try to read a record just after the last one previously read.
 *
 * If the page read callback fails to read the requested data, NULL is
 * returned.  The callback is expected to have reported the error; errormsg
 * is set to NULL.
 *
 * If the reading fails for some other reason, NULL is also returned, and
 * *errormsg is set to a string with details of the failure.
 *
 * The returned pointer (or *errormsg) points to an internal buffer that's
 * valid until the next call to XLogReadRecord.
 */
XLogRecord *XLogReadRecord(XLogReaderState *state, XLogRecPtr RecPtr, char **errormsg,
            bool doDecode, char* xlog_path)
{
...
```

继续回到函数 StartupXLOG 中。输出 redo starts at 重做点 LSN。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
       ereport(LOG, (errmsg("redo starts at %X/%X", (uint32)(t_thrd.xlog_cxt.ReadRecPtr >> 32),
                            (uint32)t_thrd.xlog_cxt.ReadRecPtr)));
```

进入执行 XLOG 记录的 do … while 循环：

```
            do {
                TermFileData term_file;
...
             // 调用函数 DispatchRedoRecord，将 XLOG 记录分派给各种资源管理器，进行实际的日志重做
             GetRedoStartTime(t_thrd.xlog_cxt.timeCost[TIME_COST_STEP_4]);
                DispatchRedoRecord(xlogreader, t_thrd.xlog_cxt.expectedTLIs, xtime);
                CountRedoTime(t_thrd.xlog_cxt.timeCost[TIME_COST_STEP_4]);
...省略
               // 读取下一条 XLOG 记录
                GetRedoStartTime(t_thrd.xlog_cxt.timeCost[TIME_COST_STEP_1]);
                if (xlogreader->isPRProcess && IsExtremeRedo()) {
                    record = ReadNextXLogRecord(&xlogreader, LOG);
                } else {
                    xlogreader = newXlogReader;
                    record = ReadRecord(xlogreader, InvalidXLogRecPtr, LOG, false);
                }
                CountRedoTime(t_thrd.xlog_cxt.timeCost[TIME_COST_STEP_1]);
            } while (record != NULL);  // end of main redo apply loop
```

对函数 DispatchRedoRecord 进行探索。该函数又调用了函数 ApplyRedoRecord。

【源码】src/gausskernel/storage/access/transam/multi_redo_api.cpp

```
void DispatchRedoRecord(XLogReaderState *record, List *expectedTLIs, TimestampTz recordXTime)
{
    if (IsExtremeRedo()) {
        extreme_rto::DispatchRedoRecordToFile(record, expectedTLIs, recordXTime);
    } else if (IsParallelRedo()) {
        parallel_recovery::DispatchRedoRecordToFile(record, expectedTLIs, recordXTime);
    } else {
        g_instance.startup_cxt.current_record = record;
        uint32 term = XLogRecGetTerm(record);
        if (term > g_instance.comm_cxt.localinfo_cxt.term_from_xlog) {
            g_instance.comm_cxt.localinfo_cxt.term_from_xlog = term;
        }

        long readbufcountbefore = u_sess->instr_cxt.pg_buffer_usage->local_blks_read;
        ApplyRedoRecord(record);
        record->readblocks     =     u_sess->instr_cxt.pg_buffer_usage->local_blks_read     -
readbufcountbefore;
        CountXLogNumbers(record);
        if (XLogRecGetRmid(record) == RM_XACT_ID)
            SetLatestXTime(recordXTime);
        SetXLogReplayRecPtr(record->ReadRecPtr, record->EndRecPtr);
        CheckRecoveryConsistency();
    }
}
```

继续进入函数 ApplyRedoRecord。该函数中有这样一行:

```
RmgrTable[XLogRecGetRmid(record)].rm_redo(record);
```

该行就是我们要找的一条 XLOG 记录对应的某个资源管理器执行 REDO 操作的地方。

【源码】src/gausskernel/storage/access/transam/multi_redo_api.cpp

```
void ApplyRedoRecord(XLogReaderState *record)
{
    ErrorContextCallback errContext;
    errContext.callback = rm_redo_error_callback;
    errContext.arg = (void *)record;
    errContext.previous = t_thrd.log_cxt.error_context_stack;
    t_thrd.log_cxt.error_context_stack = &errContext;
    if (module_logging_is_on(MOD_REDO)) {
        DiagLogRedoRecord(record, "ApplyRedoRecord");
    }
    RmgrTable[XLogRecGetRmid(record)].rm_redo(record);

    t_thrd.log_cxt.error_context_stack = errContext.previous;
}
```

RmgrTable 设计的非常精妙，它使用 C 语言实现了类似 C++多态性。其定义如下:

【源码】src/gausskernel/storage/access/transam/rmgr.cpp

```
/* must be kept in sync with RmgrData definition in xlog_internal.h */
#define PG_RMGR(symname, name, redo, desc, startup, cleanup, safe_restartpoint, undo, undo_desc,
type_name) \
        {name, redo, desc, startup, cleanup, safe_restartpoint, undo, undo_desc, type_name},

const RmgrData RmgrTable[RM_MAX_ID + 1] = {
#include "access/rmgrlist.h"
};
```

可见，RmgrTable 即资源管理器表，是 RmgrData 类型的数组。这里的巧妙之处在于，RmgrTable 数组的初始值是通过#include "access/rmgrlist.h"引入的。

先来看 RmgrData 类型定义：

【源码】src/include/access/xlog_internal.h

```
/*
 * Method table for resource managers.
 *
 * This struct must be kept in sync with the PG_RMGR definition in
 * rmgr.cpp.
 *
 * RmgrTable[] is indexed by RmgrId values (see rmgrlist.h).
 */
typedef struct RmgrData {
    const char* rm_name;
    void (*rm_redo)(XLogReaderState* record);
    void (*rm_desc)(StringInfo buf, XLogReaderState* record);
    void (*rm_startup)(void);
    void (*rm_cleanup)(void);
    bool (*rm_safe_restartpoint)(void);
    bool (*rm_undo)(URecVector *urecvector, int startIdx, int endIdx,
                    TransactionId xid, Oid reloid, Oid partitionoid,
                    BlockNumber blkno, bool isFullChain);
    void (*rm_undo_desc) (StringInfo buf, UndoRecord *record);
    const char* (*rm_type_name)(uint8 subtype);
} RmgrData;
```

再来看 access/rmgrlist.h 文件的内容：

【源码】dest/include/postgresql/server/access/rmgrlist.h

```
/*
 * List of resource manager entries.  Note that order of entries defines the
 * numerical values of each rmgr's ID, which is stored in WAL records.  New
 * entries should be added at the end, to avoid changing IDs of existing
 * entries.
 *
 * Changes to this list possibly need an XLOG_PAGE_MAGIC bump.
```

```
 */

/* symbol name, textual name, redo, desc, identify, startup, cleanup, undo, undo_desc info_type_name
*/
PG_RMGR(RM_XLOG_ID, "XLOG", xlog_redo, xlog_desc, NULL, NULL, NULL, NULL, NULL, xlog_type_name)
PG_RMGR(RM_XACT_ID, "Transaction", xact_redo, xact_desc, NULL, NULL, NULL, NULL, NULL,
xact_type_name)
PG_RMGR(RM_SMGR_ID, "Storage", smgr_redo, smgr_desc, NULL, NULL, NULL, NULL, NULL, smgr_type_name)
PG_RMGR(RM_CLOG_ID, "CLOG", clog_redo, clog_desc, NULL, NULL, NULL, NULL, NULL, clog_type_name)
PG_RMGR(RM_DBASE_ID, "Database", dbase_redo, dbase_desc, NULL, NULL, NULL, NULL, NULL,
dbase_type_name)
PG_RMGR(RM_TBLSPC_ID, "Tablespace", tblspc_redo, tblspc_desc, NULL, NULL, NULL, NULL, NULL,
tblspc_type_name)
PG_RMGR(RM_MULTIXACT_ID, "MultiXact", multixact_redo, multixact_desc, NULL, NULL, NULL, NULL, NULL,
multixact_type_name)
PG_RMGR(RM_RELMAP_ID, "RelMap", relmap_redo, relmap_desc, NULL, NULL, NULL, NULL, NULL,
relmap_type_name)

PG_RMGR(RM_STANDBY_ID,    "Standby",    standby_redo,    standby_desc,    StandbyXlogStartup,
StandbyXlogCleanup, \
    StandbySafeRestartpoint, NULL, NULL, standby_type_name)

PG_RMGR(RM_HEAP2_ID, "Heap2", heap2_redo, heap2_desc, NULL, NULL, NULL, NULL, NULL,
heap2_type_name)
PG_RMGR(RM_HEAP_ID, "Heap", heap_redo, heap_desc, NULL, NULL, NULL, NULL, NULL, heap_type_name)
PG_RMGR(RM_BTREE_ID, "Btree", btree_redo, btree_desc, btree_xlog_startup, btree_xlog_cleanup,
btree_safe_restartpoint,
    NULL, NULL, btree_type_name)
    ... 省略
```

可以看到，有若干种不同的资源管理器，每种都有自己的日志处理函数。我们关注的是重做函数。对于 INSERT 语句，执行 XLOG 记录恢复时会调用 RM_HEAP_ID 对应的重做函数 heap_redo。

找到函数 heap_redo 的定义：

【源码】src/gausskernel/storage/access/heap/heapam.cpp

```
void heap_redo(XLogReaderState* record)
{
    uint8 info = XLogRecGetInfo(record) & ~XLR_INFO_MASK;

    /*
     * These operations don't overwrite MVCC data so no conflict processing is
     * required. The ones in heap2 rmgr do.
     */
    switch (info & XLOG_HEAP_OPMASK) {
        case XLOG_HEAP_INSERT:
```

```
            heap_xlog_insert(record);
            break;
        case XLOG_HEAP_DELETE:
            heap_xlog_delete(record);
            break;
        case XLOG_HEAP_UPDATE:
            heap_xlog_update(record, false);
            break;
        case XLOG_HEAP_BASE_SHIFT:
            heap_xlog_base_shift(record);
            break;
        case XLOG_HEAP_HOT_UPDATE:
            heap_xlog_update(record, true);
            break;
        case XLOG_HEAP_NEWPAGE:
            heap_xlog_newpage(record);
            break;
        case XLOG_HEAP_LOCK:
            heap_xlog_lock(record);
            break;
        case XLOG_HEAP_INPLACE:
            heap_xlog_inplace(record);
            break;
        default:
            ereport(PANIC, (errmsg("heap_redo: unknown op code %hhu", info)));
    }
}
```

可见，不同的表修改 SQL 语句对应不同的 XLOG 处理函数。INSERT 语句对应函数 heap_xlog_insert。

函数 heap_xlog_insert 定义：

【源码】src/gausskernel/storage/access/heap/heapam.cpp

```
static void heap_xlog_insert(XLogReaderState* record)
{
    Pointer rec_data = (Pointer)XLogRecGetData(record);
    bool isinit = (XLogRecGetInfo(record) & XLOG_HEAP_INIT_PAGE) != 0;
    bool tde = XLogRecGetTdeInfo(record);
    xl_heap_insert* xlrec = NULL;
    RedoBufferInfo buffer;
    Size freespace = 0;
    XLogRedoAction action;
    RelFileNode target_node;
    BlockNumber blkno;

    if (isinit) {
```

```
        rec_data += sizeof(TransactionId);
    }
    xlrec = (xl_heap_insert*)rec_data;

    XLogRecGetBlockTag(record, HEAP_INSERT_ORIG_BLOCK_NUM, &target_node, NULL, &blkno);

    /*
     * The visibility map may need to be fixed even if the heap page is
     * already up-to-date.
     */
    if (xlrec->flags & XLH_INSERT_ALL_VISIBLE_CLEARED) {
        heap_xlog_allvisiblecleared(record, HEAP_INSERT_ORIG_BLOCK_NUM);
    }

    /*
     * If we inserted the first and only tuple on the page, re-initialize
     * the page from scratch.
     */
    if (isinit) {
        XLogInitBufferForRedo(record, HEAP_INSERT_ORIG_BLOCK_NUM, &buffer);
        action = BLK_NEEDS_REDO;
    } else {
        action = XLogReadBufferForRedo(record, HEAP_INSERT_ORIG_BLOCK_NUM, &buffer);
    }

    if (action == BLK_NEEDS_REDO) {
        char* maindata = XLogRecGetData(record);
        TransactionId recordxid = XLogRecGetXid(record);
        Size blkdatalen;
        char* blkdata = NULL;
        blkdata = XLogRecGetBlockData(record, HEAP_INSERT_ORIG_BLOCK_NUM, &blkdatalen);

        HeapXlogInsertOperatorPage(
            &buffer, (void*)maindata, isinit, (void*)blkdata, blkdatalen, recordxid, &freespace,
tde);

        MarkBufferDirty(buffer.buf);
    }
    if (BufferIsValid(buffer.buf)) {
        UnlockReleaseBuffer(buffer.buf);
    }
    /*
     * If the page is running low on free space, update the FSM as well.
     * Arbitrarily, our definition of "low" is less than 20%. We can't do much
```

```
     * better than that without knowing the fill-factor for the table.
     *
     * XXX: Don't do this if the page was restored from full page image. We
     * don't bother to update the FSM in that case, it doesn't need to be
     * totally accurate anyway.
     */
    if (action == BLK_NEEDS_REDO && freespace < BLCKSZ / 5) {
        XLogRecordPageWithFreeSpace(target_node, blkno, freespace);
    }
}
```

函数 heap_xlog_insert 调用函数 XLogReadBufferForRedo，该函数定义为：

【源码】src/gausskernel/storage/access/transam/xlogutils.cpp

```
/*
 * XLogReadBufferForRedo
 *       Read a page during XLOG replay
 *
 * Reads a block referenced by a WAL record into shared buffer cache, and
 * determines what needs to be done to redo the changes to it.  If the WAL
 * record includes a full-page image of the page, it is restored.
 *
 * 'lsn' is the LSN of the record being replayed.  It is compared with the
 * page's LSN to determine if the record has already been replayed.
 * 'block_id' is the ID number the block was registered with, when the WAL
 * record was created.
 *
 * Returns one of the following:
 *
 *   BLK_NEEDS_REDO - changes from the WAL record need to be applied
 *   BLK_DONE       - block doesn't need replaying
 *   BLK_RESTORED   - block was restored from a full-page image included in
 *                        the record
 *   BLK_NOTFOUND   - block was not found (because it was truncated away by
 *                        an operation later in the WAL stream)
 *
 * On return, the buffer is locked in exclusive-mode, and returned in *buf.
 * Note that the buffer is locked and returned even if it doesn't need
 * replaying.  (Getting the buffer lock is not really necessary during
 * single-process crash recovery, but some subroutines such as MarkBufferDirty
 * will complain if we don't have the lock.  In hot standby mode it's
 * definitely necessary.)
 *
 * Note: when a backup block is available in XLOG, we restore it
 * unconditionally, even if the page in the database appears newer.  This is
 * to protect ourselves against database pages that were partially or
```

```
 * incorrectly written during a crash.  We assume that the XLOG data must be
 * good because it has passed a CRC check, while the database page might not
 * be.  This will force us to replay all subsequent modifications of the page
 * that appear in XLOG, rather than possibly ignoring them as already
 * applied, but that's not a huge drawback.
 */
XLogRedoAction XLogReadBufferForRedo(XLogReaderState *record, uint8 block_id, RedoBufferInfo
*bufferinfo)
{
    return XLogReadBufferForRedoExtended(record, block_id, RBM_NORMAL, false, bufferinfo);
}
```

继续调用 XLogReadBufferForRedoExtended。

【源码】src/gausskernel/storage/access/transam/xlogutils.cpp

```
XLogRedoAction XLogReadBufferForRedoExtended(XLogReaderState *record, uint8 block_id,
   ReadBufferMode mode, bool get_cleanup_lock, RedoBufferInfo *bufferinfo,
   ReadBufferMethod readmethod)
{
... 省略
        redoaction = XLogReadBufferForRedoBlockExtend(&blockinfo, mode, get_cleanup_lock,
               bufferinfo,record->EndRecPtr, record->blocks[block_id].last_lsn,
               willinit, readmethod, tde);
... 省略
}
```

继续调用 XLogReadBufferForRedoBlockExtend。

其中，XLByteLE(xloglsn, PageGetLSN(page))比较 XLOG 记录中的 LSN（xloglsn）和页面中的 LSN（PageGetLSN(page)），如果 xloglsn 较小，则返回 BLK_DONE，表示页面中的数据写入较新，该 XLOG 记录不需要在页面中进行重做；否则返回 BLK_NEEDS_REDO，表示需要在页面中重做该 XLOG 记录。

【源码】src/gausskernel/storage/access/transam/xlogutils.cpp

```
XLogRedoAction XLogReadBufferForRedoBlockExtend(RedoBufferTag *redoblock, ReadBufferMode mode,
     bool get_cleanup_lock, RedoBufferInfo *redobufferinfo, XLogRecPtr xloglsn,
    XLogRecPtr last_lsn, bool willinit, ReadBufferMethod readmethod, bool tde)
{
...省略
    redobufferinfo->lsn = xloglsn;
    redobufferinfo->blockinfo = *redoblock;
    if (pageisvalid) {
        redobufferinfo->buf = buf;
        redobufferinfo->pageinfo.page = page;
        redobufferinfo->pageinfo.pagesize = pagesize;

        if (XLByteLE(xloglsn, PageGetLSN(page)))
            return BLK_DONE;
        else {
```

```
            if (SegmentNeedAdvancedLSNCheck(redoblock->rnode, redoblock->forknum, mode)) {
                /*
                 * For segment-page storage, before returning BLK_NEEDS_REDO, we need checking LSN.
Illegal LSN may be
                 * caused by dropping table and invalidating buffer. So the page can not be replayed
on. The xlog can
                 * be skipped, as later commit xlog will remove the invalid page
                 */
                /* If lsn check fails, return invalidate buffer */
                bool needRepair = false;    /* Cannot determine whether the segment page can be
repaired. */
                if (!DoLsnCheck(redobufferinfo, willinit, last_lsn, pblk, &needRepair)) {
                    redobufferinfo->buf = InvalidBuffer;
                    redobufferinfo->pageinfo = {0};
                    UnlockReleaseBuffer(buf);
                    return BLK_NOTFOUND;
                }
            }
            return BLK_NEEDS_REDO;
        }
    } else {
        redobufferinfo->buf = InvalidBuffer;
    }
    return BLK_NOTFOUND;
}
```

回到函数 heap_xlog_insert。在调用函数 XLogReadBufferForRedo 返回是否需要进行重做 XLOG 记录，保存在变量 action 中。如果需要重做（即 action 为 BLK_NEEDS_REDO），则调用函数 HeapXlogInsertOperatorPage 进行重做插入元组的实际页面动作。

函数 HeapXlogInsertOperatorPage 定义如下：

【源码】src/gausskernel/storage/access/redo/redo_heapam.cpp

```
void HeapXlogInsertOperatorPage(RedoBufferInfo *buffer, void *recorddata, bool isinit, void
*blkdata, Size datalen,
                                TransactionId recxid, Size *freespace, bool tde)
{
    Pointer rec_data = (Pointer)recorddata;
    char *data = (char *)blkdata;
    Page page = buffer->pageinfo.page;
    TransactionId pd_xid_base = InvalidTransactionId;
    xl_heap_insert *xlrec = NULL;
    ItemPointerData target_tid;
    errno_t rc = EOK;
    uint32 newlen;
    HeapTupleHeader htup;
    xl_heap_header xlhdr;
```

```
struct {
    HeapTupleHeaderData hdr;
    char data[MaxHeapTupleSize];
} tbuf;

if (isinit) {
    HeapPageHeader phdr;

    pd_xid_base = *((TransactionId *)rec_data);
    PageInit(page, buffer->pageinfo.pagesize, 0, true);
    phdr = (HeapPageHeader)page;
    phdr->pd_xid_base = pd_xid_base;
    phdr->pd_multi_base = 0;

    rec_data += sizeof(TransactionId);
    /*
     * When it comes to the TDE record, we prefer to remake the init page in TDE format.
     * And set TDE flag which on the PAGE to the enabled state.
     */
    if (tde) {
        phdr->pd_upper -= sizeof(TdePageInfo);
        phdr->pd_special -= sizeof(TdePageInfo);
        PageSetTDE(page);
    }
}
xlrec = (xl_heap_insert *)rec_data;

ItemPointerSetBlockNumber(&target_tid, buffer->blockinfo.blkno);
ItemPointerSetOffsetNumber(&target_tid, xlrec->offnum);
rc = memset_s(&tbuf, sizeof(tbuf), 0, sizeof(tbuf));
securec_check(rc, "\0", "\0");

OffsetNumber maxoff = PageGetMaxOffsetNumber(page);

if (maxoff + 1 < xlrec->offnum)
    ereport(PANIC, (errmsg("heap_insert_redo: invalid max offset number")));

newlen = datalen - SizeOfHeapHeader;
Assert(datalen > SizeOfHeapHeader && newlen <= MaxHeapTupleSize);
rc = memcpy_s((char *)&xlhdr, SizeOfHeapHeader, data, SizeOfHeapHeader);
securec_check(rc, "", "");
data += SizeOfHeapHeader;

htup = &tbuf.hdr;
```

```
    rc = memset_s((char *)htup, sizeof(HeapTupleHeaderData), 0, sizeof(HeapTupleHeaderData));
    securec_check(rc, "\0", "\0");
    /* PG73FORMAT: get bitmap [+ padding] [+ oid] + data */
    rc = memcpy_s((char *)htup + offsetof(HeapTupleHeaderData, t_bits), newlen, data, newlen);
    securec_check(rc, "\0", "\0");
    newlen += offsetof(HeapTupleHeaderData, t_bits);
    htup->t_infomask2 = xlhdr.t_infomask2;
    htup->t_infomask = xlhdr.t_infomask;
    htup->t_hoff = xlhdr.t_hoff;
    HeapTupleHeaderSetXmin(page, htup, recxid);
    HeapTupleHeaderSetCmin(htup, FirstCommandId);
    htup->t_ctid = target_tid;

    if (PageAddItem(page, (Item)htup, newlen, xlrec->offnum, true, true) == InvalidOffsetNumber)
        ereport(PANIC, (errmsg("heap_insert_redo: failed to add tuple")));

    if (freespace != NULL) {
        *freespace = PageGetHeapFreeSpace(page);
    }

    PageSetLSN(page, buffer->lsn);

    if (xlrec->flags & XLH_INSERT_ALL_VISIBLE_CLEARED)
        PageClearAllVisible(page);
}
```

11.4.4 添加代码：在数据库恢复过程中输出信息

在本节中，我们进行代码实验，通过注释掉 xlog.cpp 文件中的一些条件编译宏变量 WAL_DEBUG 和代码，打开一些日志调试信息输出代码。

1. 注释掉条件编译，打开函数 xlog_outrec
   注释掉两处条件编译 WAL_DEBUG，打开函数 xlog_outrec。该函数用于输出一条 XLOG 记录的相关信息。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
/* [ DBLAB ========================================================== */
// #ifdef WAL_DEBUG
/* DBLAB ] ========================================================== */
static void xlog_outrec(StringInfo buf, XLogReaderState *record);
/* [ DBLAB ========================================================== */
// #endif
/* DBLAB ] ========================================================== */
```

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
/* [ DBLAB ========================================================== */
```

```
// #ifdef WAL_DEBUG
/* DBLAB ] ============================================================ */

static void xlog_outrec(StringInfo buf, XLogReaderState *record)
{
    int block_id;

       appendStringInfo(buf, "prev %X/%X; xid %u", (uint32)(XLogRecGetPrev(record) >> 32),
(uint32)XLogRecGetPrev(record),
                        XLogRecGetXid(record));

    appendStringInfo(buf, "; len %u", XLogRecGetDataLen(record));

    /* decode block references */
    for (block_id = 0; block_id <= record->max_block_id; block_id++) {
        RelFileNode rnode;
        ForkNumber forknum;
        BlockNumber blk;

        if (!XLogRecHasBlockRef(record, block_id)) {
            continue;
        }

        XLogRecGetBlockTag(record, block_id, &rnode, &forknum, &blk);
        if (forknum != MAIN_FORKNUM) {
              appendStringInfo(buf, "; blkref #%u: rel %u/%u/%u, fork %u, blk %u", block_id,
rnode.spcNode, rnode.dbNode,
                                rnode.relNode, forknum, blk);
        } else {
           appendStringInfo(buf, "; blkref #%u: rel %u/%u/%u, blk %u", block_id, rnode.spcNode,
rnode.dbNode,
                                rnode.relNode, blk);
        }
        if (XLogRecHasBlockImage(record, block_id)) {
            appendStringInfo(buf, " FPW");
        }
    }
}
/* [ DBLAB ============================================================ */
// #endif /* WAL_DEBUG */
/* DBLAB ] ============================================================ */
```

2. 注释掉 StartupXLOG 中的条件编译和部分代码
   在函数 StartupXLOG 中，注释掉 do … while 循环内的 WAL_DEBUG 条件编译和部分代码，输出进行恢复时重做每条 XLOG 记录的相关信息。

【源码】src/gausskernel/storage/access/transam/xlog.cpp

```
...
/* [ DBLAB ========================================================== */
// #ifdef WAL_DEBUG
   // if (u_sess->attr.attr_storage.XLOG_DEBUG ||
   //    (rmid == RM_XACT_ID && u_sess->attr.attr_common.trace_recovery_messages <= DEBUG2) ||
   //    (rmid != RM_XACT_ID && u_sess->attr.attr_common.trace_recovery_messages <= DEBUG3)) {
/* DBLAB ] ========================================================== */

   StringInfoData buf;

   initStringInfo(&buf);
   appendStringInfo(&buf, "REDO @ %X/%X; LSN %X/%X: ", (uint32)(t_thrd.xlog_cxt.ReadRecPtr >>
                    32),(uint32)t_thrd.xlog_cxt.ReadRecPtr,(uint32)(t_thrd.xlog_cxt.EndRecPtr
                    >> 32),(uint32)t_thrd.xlog_cxt.EndRecPtr);
   xlog_outrec(&buf, xlogreader);
   appendStringInfo(&buf, " - ");

   RmgrTable[record->xl_rmid].rm_desc(&buf, xlogreader);
   ereport(LOG, (errmsg("%s", buf.data)));
   pfree_ext(buf.data);

/* [ DBLAB ========================================================== */
// }
// #endif
/* DBLAB ] ========================================================== */
...
```

3. 确保 openGauss 数据库服务器已停止，执行编译安装过程
4. 启动 openGauss 数据库服务器
5. 在新的 SSH 会话中打开 gsql 客户端，执行 SQL 语句

```
[dblab@eduog openGauss-server-v3.0.0]$ gsql railway -r
gsql ((openGauss 3.0.0 build ) compiled at 2023-02-23 19:56:43 commit 0 last mr  debug)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

railway=# drop table table_a;
DROP TABLE
railway=# CREATE TABLE table_a ( a CHAR(1) );
CREATE TABLE
railway=# INSERT INTO table_a VALUES ('A');
INSERT 0 1
```

6. 立即在启动数据库服务器的 SSH 会话中以 immediate 模式关闭数据库服务器

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl stop -D $GAUSSHOME/data -m immediate
[2023-02-23     20:33:47.398][2051468][][gs_ctl]:     gs_ctl     stopped     ,datadir     is
```

```
/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data
waiting for server to shut down.... done
server stopped
```

7. 重新启动数据库服务器

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl start -D $GAUSSHOME/data -Z single_node -l logfile
[2023-02-23 20:35:35.712][2051527][][gs_ctl]: gs_ctl started,datadir is
/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data
[2023-02-23 20:35:35.744][2051527][][gs_ctl]: waiting for server to start...
.
[2023-02-23 20:35:36.757][2051527][][gs_ctl]: done
[2023-02-23 20:35:36.757][2051527][][gs_ctl]: server started
(/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data)
```

8. 查看 pg_log 目录下的日志文件

用 vim 查看 pg_log 目录中最新的一个日志文件。此处需要将 postgresql-2023-02-23_203536.log 替换为实验时生产的最新日志文件名。

```
[dblab@eduog openGauss-server-v3.0.0]$ vim
$GAUSSHOME/data/pg_log/postgresql-2023-02-23_203536.log
```

如上述实验步骤无误，将产生以下日志输出（具体时间不同）。请对照 8.4.3 节中的代码，分析这里的日志输出。

```
...省略
LOG:  database system timeline: 66
LOG:  database system was interrupted; last known up at 2023-02-23 20:33:35 CST
...省略
LOG:  redo record is at 0/162B85E8; shutdown FALSE
...省略
LOG:  database system was not properly shut down; automatic recovery in progress
LOG: StartupXLOG PrintCkpXctlControlFile: [checkPoint] oldCkpLoc:0/162B8668, oldRedo:0/162B85E8,
newCkpLoc:0/162B8668, newRedo:0/162B85E8, preCkpLoc:0/162B6CD8
...省略
LOG:  redo starts at 0/162B85E8
...省略
...以下是通过本节实验操作所产生的重做 XLOG 记录的输出
LOG:  REDO @ 0/162B85E8; LSN 0/162B8618: prev 0/162B8408; xid 0; len 8 -  XLOG_STANDBY_CSN
LOG:  REDO @ 0/162B8618; LSN 0/162B8668: prev 0/162B85E8; xid 0; len 40 -  XLOG_RUNNING_XACTS
LOG: REDO @ 0/162B8668; LSN 0/162B8708: prev 0/162B8618; xid 0; len 120 - checkpoint: redo 0/162B85E8;
len 120; next_csn 2575; recent_global_xmin 34896; tli 1; fpw false; xid 34897; oid 90272; multi
2; offset 0; oldest xid 12772 in DB 15552; oldest running xid 34897; oldest xid with epoch having
undo 0; online at Thu Feb 23 20:33:35 2023; remove_seg 0/6
LOG:  REDO @ 0/162B8708; LSN 0/162B8760: prev 0/162B8668; xid 34897; len 11; blkref #0: rel
1663/57357/82080, blk 0 - XLOG_HEAP_INSERT insert(init): off 34894
LOG:  REDO @ 0/162B8760; LSN 0/162B87A0: prev 0/162B8708; xid 34897; len 24 -
XLOG_STANDBY_CSN_COMMITTING, xid 34897, csn 2575
LOG:  REDO @ 0/162B87A0; LSN 0/162B87E8: prev 0/162B8760; xid 34897; len 32 -
XLOG_XACT_COMMIT_COMPACT commit: 2023-02-23 20:33:45.17921+08; csn:2575; RecentXmin:34897
```

```
...通过本节实验操作所产生的重做 XLOG 记录的输出结束
...省略
LOG:  checkpointer started, CheckPointTimeout is 60
...省略
LOG:  redo done at 0/162B87A0, end at 0/162B87E8
LOG:     [PR]:   Recoverying   elapsed:   102717   us,   redoTotalBytes:512,EndRecPtr:371951592,
redoStartPtr:371951080,speed:0 MB/s, totalTime:102717
...省略
LOG:  last completed transaction was at log time 2023-02-23 20:33:45.17921+08
...省略
LOG:  redo done, nextXid: 35253, startupMaxXid: 35253, recentLocalXmin: 34898, recentGlobalXmin:
34898, PendingPreparedXacts: 0, NextCommitSeqNo: 2576, cutoff_csn_min: 0.
```

11.4.5 验证数据库备份与 PITR 恢复

openGauss 提供 gs_basebackup 脚本进行全量的数据文件备份。（需要在数据库启动状态下进行）

1. 首先启动数据库，并连接数据库，确定初始数据库中已有的数据库。

```
openGauss=# \d
                   List of relations
 Schema |  Name | Type  | owner |             Storage
 -------+-------+-------+-------+-----------------------------
 public | users | table | dblab | {orientation=row,compression=no}
(1 row)
```

此时数据库中只有一个 users 表，退出数据库，并对初始数据库进行备份。

2. 输入 gs_basebackup -D /home/dblab/backup -h 127.0.0.1 -p 5432（建议使用示例相同目录进行备份，防止出现权限问题）

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_basebackup -D /home/dblab/backup -h 127.0.0.1
-p 5432
INFO:  The starting position of the xlog copy of the full build is: 0/3000028.The slot
minimum LSN is: 0/0.
[2023-01-14 14:52:51]: begin build tablespace list
[2023-01-14 14:52:51]: finish build tablespace list
[2023-01-14 14:52:51]: begin get xlog by xlogstream
[2023-01-14 14:52:51]: check identify system success
[2023-01-14 14:52:51]: send START_REPLICATION 0/3000000 success
[2023-01-14 14:52:51]: keepalive message is received
[2023-01-14 14:52:51]: keepalive message is received
[2023-01-14 14:52:56]: gs_basebackup: base backup successfully
```

备份成功，进入到备份目录，可以查看到备份成功的文件。

```
[dblabaeduog backup]$ ls
asp_data         gs_profile           pg_csnlog            pg_ident.conf
pg_notify      pg_snapshots       PG_VERSION         postmaster.pid.lock  backup_label
gswlm_userinfo.cfg   pg_ctl.lock          pg_llog
```

```
pg_perf          pg_stat_tmp          pg_xlog              sql_monitor
base             pg_audit             pg_errorinfo         pg_logical
pg_replslot      pg_tblspc            postgresql.conf      undo
global          pg_clog             pg_hba.conf         pg_multixact        pg_serial
pg_twophase          postgresql.conf.lock
```

此时就拥有了两个前提条件中的第一个条件：物理备份的全量数据文件。

插入测试数据并拷贝 WAL 日志文件。

3. 在步骤 2 中我们已经将数据库备份了，接下来我们连接数据库，在其基础上插入测试数据。

```
openGauss=# CREATE TABLE testPITR1 AS SELECT * FROM pg_class;
INSERT 0 787
openGauss=#SELECT pg_create_restore_point('restore_point_1');
  pg_create_restore_point
--------------------------
  0/40416D8
(1 row)
```

4. 之后调用 pg_create_restore_point 函数在此时创建了一个还原点 1。

```
openGauss=# SELECT pg_create_restore_point('restore_point_1');
```

5. 接下来再创建一个测试表 testPITR2，并创建还原点 2:

```
openGauss=# CREATE TABLE testPITR2 AS SELECT * FROM pg_description;
INSERT 0 3517
openGauss=# SELECT pg_create_restore_point('restore_point_2');
  pg_create_restore_point
--------------------------
  0/40AA048
(1 row)
```

6. 退出数据库，并把 WAL 日志文件（存储在数据目录下的 dest/data/pg_xlog 文件夹中）复制到一个其它目录中，比如存储到/home/dblab/archive 目录下。

```
[dblab@eduog openGauss-server-v3.0.0]$ cp -r dest/data/pg_xlog/* /home/dblab/archive/
```

此时，我们就把 WAL 日志文件拷贝到了其它文件夹中，达成了两个前提条件的第二个。

接着实现 PITR 恢复。根据实验步骤 8.3.6 中的 PITR 步骤介绍，我们开始进行 PITR 恢复。

7. 首先关闭数据库。

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl stop -D $GAUSSHONE/data -Z single_node -l
logfile
[2023-01-14 15:05:06.382][727548][][gs_ctl): gs_ctl stopped ,datadir is /home/dblab/
opengauss-compile/openGauss-server-v3.0.0/dest/data
waiting for server to shut down..... done
server stopped
```

8. 然后清空当前数据目录下的文件，并把之前的全量备份文件复制过来。

```
[dblab@eduog data]$ rm -r *
[dblab@eduog data]$ cp -r /home/dblab/backup/*
/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data
```

```
[dblab@eduog data]$ ls
asp_data         gs_profile           pg_csnlog            pg_ident.conf
pg_notify       pg_snapshots       PG_VERSION         postmaster.pid.lock  backup_label
gswlm_userinfo.cfg   pg_ctl.lock          pg_llog
pg_perf          pg_stat_tmp          pg_xlog              sql_monitor
base             pg_audit             pg_errorinfo         pg_logical
pg_replslot      pg_tblspc            postgresql.conf      undo
global          pg_clog             pg_hba.conf         pg_multixact        pg_serial
pg_twophase          postgresql.conf.lock
```

9. 之后清空 data/pg_xlog 下的文件，并把之前拷贝好的 WAL 日志文件复制进去。

```
[dblab@eduog data]$ rm -r pg_xlog/*
[dblab@eduog data]$ cp -r /home/dblab/archive/*
/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data/pg_xlog/
```

10. 编辑 recovery.conf 文件，在里面输入 recovery_target_name = 'restore_point_1' 用来配置恢复进度

```
[dblab@eduog data]$ vi recovery.conf
[dblab@eduog data]$ cat recovery.conf
recovery_target_name = 'restore_point_1'
```

在配置文件中，指定把数据文件恢复到还原点 1。

11. 接下来启动数据库。

```
[dblab@eduog data]$ gs_ctl start -D $GAUSSHOME/data -Z single_node -l logfile
[2023-01-14 15:17:3.603][727926][][gs_ctl]: gs_ctl started,datadir is /home/dblab/
opengauss-compile/openGauss-server-v3.0.0/dest/data
[2023-01-14 15:17:33.636][727920][][gs_ctl]: waiting for server to start...
[2023-01-14 15:17:37.658][727920][][gs_ctl]: done
[2023-01-14 15:1737.658][727920][][gs_ctl]: server
started(/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data)
[dblab@eduog data]$ gsql postgres -r
gsql ((openGauss 3.0.0 build ) compiled at 2023-01-09 17:48:54 commit 0 last mr debug)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.
openGauss=# \d
                      List of relations
 Schema |    Name   | Type  | Owner |             Storage
--------+-----------+-------+-------+-----------------------------------
 public | testPITR1 | table | dblab | {orientation=row, compression=no}
 public | users     | table | dblab | {orientation=row, compression=no}
 (2 rows)
```

输入\d 可以看到，还原点 1 前的 testPITR1 表已经被恢复了，但是还原点 2 才有的 testPITR2 表没有被恢复。

12. 接下来我们退出并关闭数据库，修改配置文件为还原到还原点 2.

```
openGauss=# \q
[dblab@eduog data]$ gs_ctl stop -D $GAUSSHOME/data -Z single_node -l logfile
[2023-61-14 15:18:37.340][727974][][gs_ctl]: gs_ctl stopped ,datadir is
```

```
/home/dblab/opengauss-compil1e/openGauss-server-3.0.0/dest/data
waiting for server to shut down.... done
server stopped
[dblab@eduog data]$ vi recovery.conf
[dblab@eduog data]$ cat recovery.conf
recovery_target_name = 'restore_point_2'
```

13. 重新启动数据库，再查看表 testPITR2 是否被创建。

```
openGauss=# \d
                      List of relations
 Schema |    Name   | Type  | Owner |             Storage
--------+-----------+-------+-------+---------------------------------
 public | testPITR1 | table | dblab | {orientation=row, compression=no}
 public | testPITR2 | table | dblab | {orientation=row, compression=no}
 public | users     | table | dblab | {orientation=row, compression=no}
 (3 rows)
```

可以看到表 testPITR2，已经被创建了。至此，我们就利用 WAL 日志完成了数据库的恢复。

11.5 实验结果

【请按照要求完成实验操作】

1. 完成实验步骤 8.4.1、8.4.2 节并查看输出。

2. 完成实验步骤 8.4.3 节，绘制实验步骤中函数 StartupXLOG 涉及到的函数调用图。

3. 完成实验步骤 8.4.4 节，查看第 8 步的日志输出。

4. 完成实验步骤 8.4.5 节并查看输出。

11.6 讨论与总结

【请将实验中遇到的问题描述、解决办法与思考讨论列在下面，并对本实验进行总结。】