Artifact for "Translytical Processing via DB-OS Co-designed Buffer: Cross-Engine Isolation and Instant Update Visibility for HTAP" (VISTA)

This repository contains the artifact for our paper, "Translytical Processing via DB-OS Co-designed Buffer: Cross-Engine Isolation and Instant Update Visibility for HTAP".

Our System, VISTA, is a DB-OS co-designed buffer virtualization framework that enables efficient, single-node Hybrid Transactional/Analytical Processing (HTAP). It provides strong performance isolation between engines and instant update visibility, eliminating the need for traditional ETL or log shipping pipelines.

VISTA features:

- OS-assisted virtual memory remapping for real-time consistent view
- Virtual memory segmentation for performance isolation
- Runtime for data format conversion and caching

In this artifact, you could find the code and scripts for VISTA and its evaluation.

Directory Structure

The repository is organized as follows:

- `vista-kernel/`: Contains the source code for the **VISTA OS module**.
- `vista/`: Contains the **user-space** components of VISTA.
  - `postgres/`: The modified PostgreSQL source code that integrates with VISTA.
  - `duckdb-ex/`: The DuckDB source code with VISTA extension.
  - `lib-scan/`: The upper-level library used by the OLAP engine to scan and convert data from remapped segments.
  - `libvista/`: The lower-level library providing the core VISTA runtime for the both engines.
- `evaluation/`: Scripts and configurations for running the performance evaluations.
  - `olap-only/`: Scripts for running the OLAP-only baseline experiments.
  - `olap-with-oltp/`: Scripts for running the full HTAP experiments with both OLTP and OLAP workloads.
- `scripts/`: Helper scripts for building the dataset, setting up the different system configurations (VISTA, vanilla PostgreSQL, etc.), and managing the server.

Code Overview

This section provides a detailed walkthrough of the VISTA implementation, highlighting the key components and logic in the OS module, the user-level library and its integration with the OLTP engine (PostgreSQL) and the OLAP engine (DuckDB).

VISTA OS Module (`vista-kernel/`)

The core of VISTA's OS-level support resides in `vista-kernel/linux/vista/vista.c`. This module manages 1 GB huge pages (VMSegs) and provides system calls for memory mapping and remapping.

Key System Calls

VISTA introduces several system calls to manage VMSegs from user space.

```
// vista-kernel/linux/vista/vista.c
vista_mmap(count, flag, ...);/* map reserved hugepages to user address
space
    specified with the `flag`(OLTP or OLAP) */
vista_remap(idx);/* remap the `idx` hugepage to the OLAP engine's address
space */
vista_unmap(addr);/* unmap a hugepage addressed with the `addr` */
vista_munmap();/* free hugepages allocated with `vista_mmap` */
```

IPI Handler

The most critical operation is the remapping of a sealed OLTP segment to the OLAP engine's address space, which is handled by an Inter-Processor Interrupt (IPI).

```
// IPI handler for remapping (simplified workflow)
static void vista_remap_handler(void *arg)
{
    // 1. Identify the OLAP task on the current CPU.
    // 2. Find the virtual address (remap_window) for the OLAP task.
    // 3. Locate the Page Upper Directory (PUD) entry for this address.
    pgd = pgd_offset(mm, addr);
    p4d = p4d_offset(pgd, addr);
    pud = pud_offset(p4d, addr);

    // 4. Clear the old mapping.
    pud_clear(pud);

    // 5. Get the physical address (PFN) of the sealed OLTP huge page.
    pfn = folio_pfn(vista_hugepage_mgr.oltp_folios[idx]);

    // 6. Insert the new mapping into the PUD as a read-only entry.
    vista_insert_pfn_pud(vma, addr, pud, pfn_to_pfn_t(pfn), 0);

    // 7. Alert the user-space OLAP runtime that the remapping is complete.
    atomic_set((atomic_t *)remap_alert, 1);
}
```

When the OLTP engine seals a segment, it invokes the `vista_remap` system call (See OLTP-side integration below). This triggers `smp_call_function_many`, which sends an IPI to all CPU cores running OLAP threads, executing `vista_remap_handler` on each of them. This ensures that the page table updates are propagated quickly and consistently across all OLAP cores.

OLTP-Side Integration (`vista/postgres/` and `vista/libvista/`)

The OLTP engine is a modified version of PostgreSQL. The core logic for VISTA integration is in `postgres/src/backend/storage/vista_buffer/`, which is a modified fork of the default PostgreSQL buffer manager(`postgres/src/backend/storage/buffer/`).

Buffer Pool Initialization (`buf_init.c`)

VISTA's integration begins at the earliest stage of PostgreSQL's startup: buffer pool initialization. The modified `vista/postgres/src/backend/storage/vista_buffer/buf_init.c` is responsible for setting up the VMSeg-backed buffer pools.

During initialization, it performs the following key steps:

1. **VMSeg Allocation:** It calls the `vista_register` and `vista_init_vmseg` functions, which are thin wrappers around the VISTA system calls. This step requests the VISTA OS module to allocate and map several 1 GB huge pages (VMSegs) into the PostgreSQL process's address space. These VMSegs will house the OLTP buffer pools.
2. **Buffer Pool Initialization:** VISTA creates multiple buffer pools (`N_POOLS`, typically 3 in our current implementation) within these VMSegs. A custom allocator (`vista_vmseg_allocator`) carves out memory from each VMSeg for essential structures.
3. **Shared Metadata and Offsets:** A dedicated shared region created by POSIX shared memory is used for shared metadata accessible by both the OLTP and OLAP engines. Crucially, `buf_init.c` initializes a `vista_bufferpool_offsets` structure in this shared region. This structure stores the relative offsets of all key data structures (like buffer descriptors and blocks) within each VMSeg. When a segment is remapped, the OLAP engine uses these offsets to locate and interpret the data pages and their metadata correctly. Also, `vista_BufferPoolControl`, which is the main control structure, resides in this region.

Buffer Allocation and Copy-on-Seal

VISTA intercepts PostgreSQL's buffer allocation logic to direct data page allocations to VMSegs. The `vista_BufferAlloc_forOLTP` function handles this, implementing a "copy-on-seal" policy.

```
// vista/postgres/src/backend/storage/vista_buffer/bufmgr.c

FUNCTION vista_BufferAlloc_forOLTP()
    // Determine current operation mode (NORMAL or FLUSH)
    is_flushing = (ctl->vista_op_mode == VISTA_MODE_FLUSH);

    IF !is_flushing THEN
        // NORMAL mode: Allocate from the active segment.
        buf = vista_BufferAlloc_forOLTP_NonFlushing(active_segment, ...);
    ELSE
        // FLUSH mode: A segment is being sealed.
        // Modifications to pages in the sealed segment require copy-on-
seal.
        buf = vista_BufferAlloc_forOLTP_Flushing(sealed_segment,
active_segment, ...);
    ENDIF
```

```
    // Check if the active segment has breached the flush threshold.
    IF usage > OLTP_FLUSH_THRESHOLD THEN
        // Trigger the flush worker to seal and flush the active segment.
        SetLatch();
    ENDIF

    RETURN buf;
END FUNCTION
```

When a transaction needs to modify a page in a segment that is being sealed (`VISTA_MODE_FLUSH`), `vista_BufferAlloc_forOLTP_Flushing` first copies the page from the sealed segment to the new active segment before returning the buffer.

Flush Worker

A dedicated background worker, `VistaFlushWorkerMain`, manages the segment sealing and remapping process. It is triggered by a latch when the active segment is nearly full.

```
// vista/postgres/src/backend/storage/vista_buffer/vista_flushworker.c

FUNCTION flush_segment_and_update_bitmap()
    // 1. Transition system to FLUSH mode and increase generation.
    ctl->vista_op_mode = VISTA_MODE_FLUSH;
    ctl->global_generation++;

    // 2. Wait for OLTP to finish with the sealed segment
    // and then collect all dirty buffers from the sealed segment.
    for (i = 0; i < NBuffers; i++) {
        dirty_buffers[num_dirty].buf = ...;
    }
    // 3. Update the format cache invalidation bitmap.
    for (i = 0; i < num_dirty; i++) {
        set_bit(bitmap, idx);
    }

    // 4. Trigger the OS to remap the sealed segment to OLAP processes.
    vista_remap(pool_id);

    // 5. Flush dirty buffers to disk asynchronously using io_uring.
    vista_flush_buffers_uring_with_info(dirty_buffers, num_dirty);

    // 6. Transition system back to NORMAL mode.
    ctl->vista_op_mode = VISTA_MODE_NORMAL;

    // 7. Wait for OLAP to finish with the remapped segment, then clean it
up.
    clean_old_segment(old_seg);
END FUNCTION
```

OLAP-Side Integration (`vista/duckdb-ex/`, `vista/lib-scan/`, and `vista/libvista`)

The OLAP engine is DuckDB with a custom extension that can scan data from PostgreSQL via VISTA. The scan logic integrates a format cache to avoid repeated data conversion.

Scan Flow

The main entry point for a scan is `PostgresScan`. The core logic resides in `ScanChunk`, which coordinates reading from the format cache or directly from PostgreSQL data pages (via remapped VMSegs or prefetched from storage).

```
// vista/duckdb-ex/src/postgres_scanner_libscan.cpp

FUNCTION PostgresLocalState::ScanChunk()
    (...)
    // Check the format cache for valid SubGroups within this BlockGroup.
    uncached_subgroup_mask = cache->getInvalidSubGroupMask(...);

    // Process one SubGroup at a time
    IF !(uncached_subgroup_mask) THEN
        // SubGroup is in cache: load it.
        subgroup_chunks = cache->loadSubGroup(...);
    ELSE
        // SubGroup is not in cache: read from data pages.
        // This will read from remapped VMSegs or prefetched pages.
        LibScan_FillChunkFromBlockGroup(chunk, libscan_state);

        // If the data is stable, store the converted chunk in the cache.
        IF subgroup_all_valid THEN
            cache->storeSubGroup(sg_id, subgroup_chunks);
        ENDIF
    ENDIF
    (...)
END FUNCTION
```

On-the-fly Transformation

When a SubGroup is not in the format cache, `LibScan_FillChunkFromBlockGroup` is called. This function reads raw PostgreSQL data pages and converts them to DuckDB's columnar format.

```
// vista/lib-scan/src/tuple_to_duckdb.cpp

FUNCTION FillChunkFromBlockGroup_Vista()
    (...)
    // 1. Get the PostgreSQL data page.
    // Try to read from a remapped VMSeg first for recent data.
    remapped_page = vista_try_read_remapped_page(...);
    // Also retrieve the data page prefetched by libvista
    vista_page = vista_get_page(...);
```

```
    // If not in a remapped segment, get from prefetched I/O.
    current_page = remapped_page ? remapped_page : vista_page;

    // 2. Process the tuples on the page.
    tuples_in_block = ProcessBlock(duckdb_chunk, ..., current_page, ...);

    // 3. Release the page.
    vista_put_page(...);
    (...)
END FUNCTION

FUNCTION ProcessBlock(...)
    FOR each tuple on page:
        // Perform MVCC visibility check using the exported snapshot.
        IF VistaHeapTupleSatisfiesMVCC(tuple, snapshot) THEN
            // Convert to columnar format and insert into DuckDB chunk.
            InsertTupleIntoChunk(chunk, slot, ...);
        ENDIF
    END FOR
END FUNCTION
```

Accessing Remapped VMSeg

The bridge between the OLAP engine and the freshly remapped OLTP data is `vista_try_read_remapped_page`. It orchestrates the process of attaching to, reading from, and detaching from the remapped VMSeg.

1. **Attach (`vista_remapped_open`):** When the OS signals a successful remap (via `remap_alert_ptr`), the first OLAP thread to notice initiates the attach process.
    - It increments a reference count (`remap_count`) in the shared control structure to prevent the OLTP flush worker from cleaning up the VMSeg while OLAP threads are reading it.
    - The thread then calls `vista_reconstruct_buffer_pool` to calculate the absolute pointers to the buffer descriptors, data blocks, and other control structures within the remapped window. This makes the raw memory block a navigable buffer pool.
2. **Read:** Once attached, threads can use `vista_read_buffer` to look up specific pages within the reconstructed buffer pool.
3. **Unmap (`vista_unmap_logic`):** When the OLTP flush worker finishes its cycle, OLAP threads detect this change by reading the state and trigger the unmap logic.
    - It waits for all local OLAP readers to finish, then decrements the shared `remap_count`, signaling to the OLTP engine that the OLAP side is finished with the VMSeg.
    - It invokes the `vista_unmap` system call to release the mapping.

Evaluation

Setup and Installation

1. Build and Install the VISTA OS Module

VISTA uses the Linux kernel v6.12.

```
cd vista-kernel/linux

make oldconfig
make menuconfig
# Ensure following configs are enabled
#  CONFIG_VISTA
#  CONFIG_TRANSPARENT_HUGEPAGE (madvise)
make -j$(nproc)
make modules -j$(nproc)
make modules_install -j$(nproc)
make headers -j$(nproc)
make headers_install INSTALL_HDR_PATH=/usr
```

Note: You will need to reboot your system to use the new kernel.

2. Build Systems and Dataset

```
# Install dependencies
./scripts/requirements.sh

# Build each system
./scripts/setup_vista.sh
./scripts/setup_duckdb_pg_ext.sh
./scripts/setup_pg_duckdb.sh
./scripts/setup_pgonly.sh

# Create dataset (1000 warehouses)
./scripts/build_dataset.sh --warehouse 1000
```

3. System Configuration

Before running experiments, create cgroups:

```
sudo ./evaluation/.config_scripts/create_eval_cgroup.sh
```

This creates:

- `vista_base`: 32 CPUs (NUMA0), 64GB - for pgonly, duckdb_ext, pg_duckdb
- `vista_nohp`: 32 CPUs (16 NUMA0 + 16 NUMA1), 60GB - for vista

4. Running Experiments

olap-only/

OLAP-only workload: Runs 22 CH-Benchmark queries without concurrent OLTP.

Usage:

```
./run-olap_only.py --data <data-dir> --baseline
<pgonly|duckdb_ext|pg_duckdb|vista>
```

Example:

```
./run-olap_only.py --baseline vista --iter 1 --data
../../dataset/warehouse-1000 \
    --repeat 2 --excluding "21" --output vista.json --timeout 600 --no-
random
```

- `--repeat 2`: First run is cold (no cache), second run is warm (cached)
- `--iter`: Number of iterations
- `--excluding`: Skip specific queries (e.g., "21" or "5,21")
- `--timeout`: Query timeout in seconds

olap-with-oltp/

HTAP workload: Runs OLAP queries with concurrent TPC-C workload.

Usage:

```
./run_eval.sh --data <data-dir> --baseline
<pgonly|duckdb_ext|pg_duckdb|vista>
```

Example:

```
./run_eval.sh --data ../../dataset/warehouse-1000 --baseline vista
```

To modify OLTP workers or iterations, edit the script:

- Line 11: `ITERATION=3` - Number of iterations
- Line 84: `WORKER_COUNTS=(4 12)` - OLTP worker count

Baseline Ports

- `pgonly`: PostgreSQL only (port 5003)
- `duckdb_ext`: DuckDB postgres_scanner extension (port 5001)
- `pg_duckdb`: pg_duckdb extension (port 5002)
- `vista`: VISTA (port 5004)