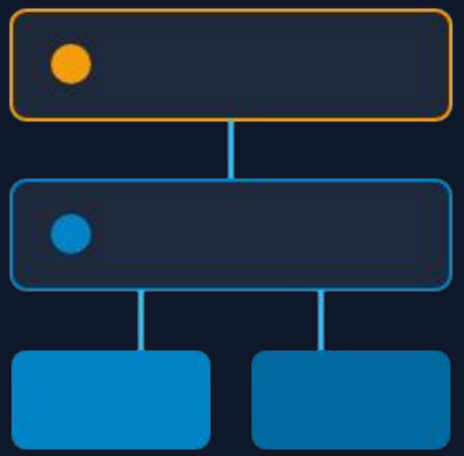
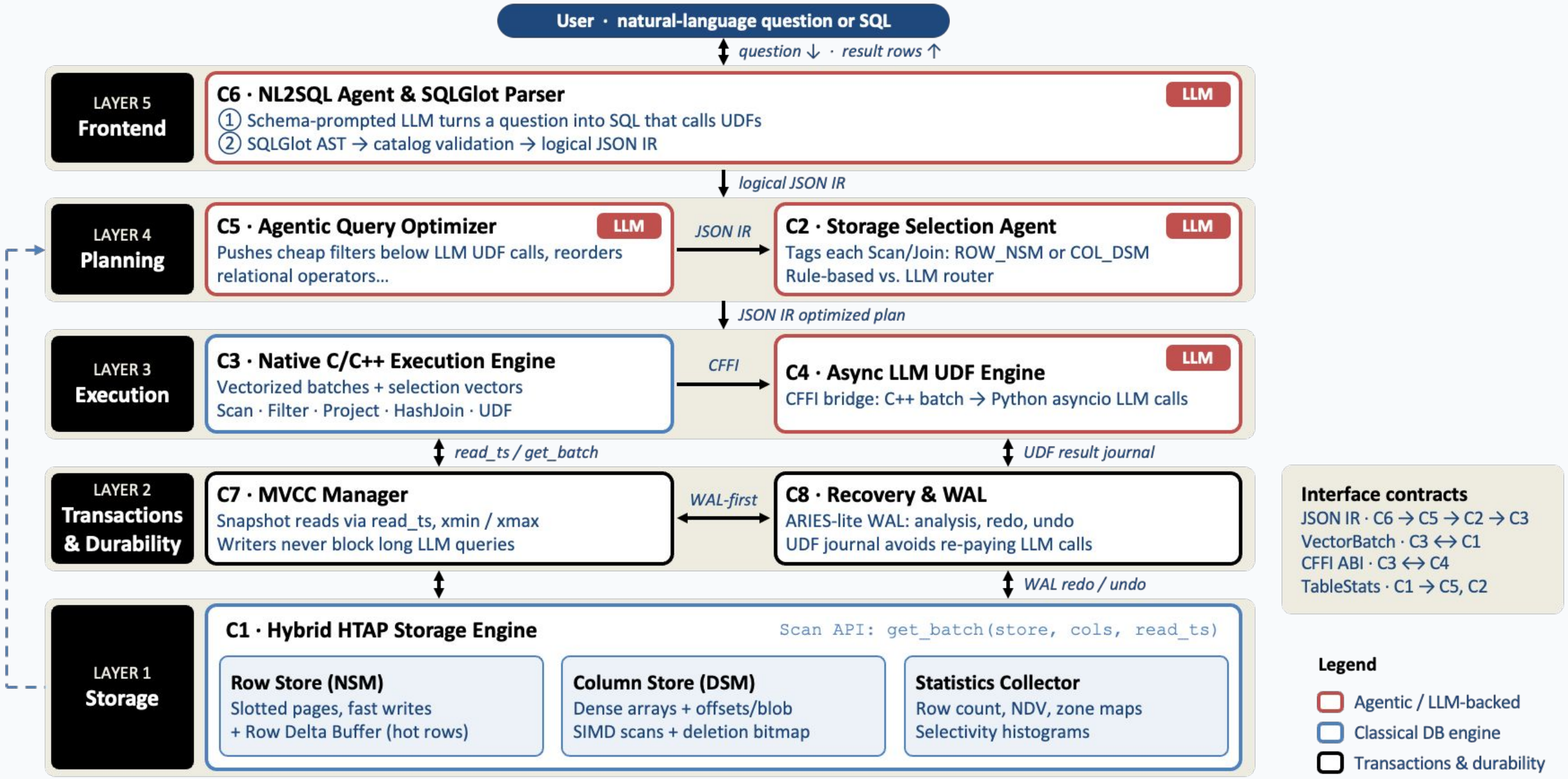
Agentic Hybrid DBMS

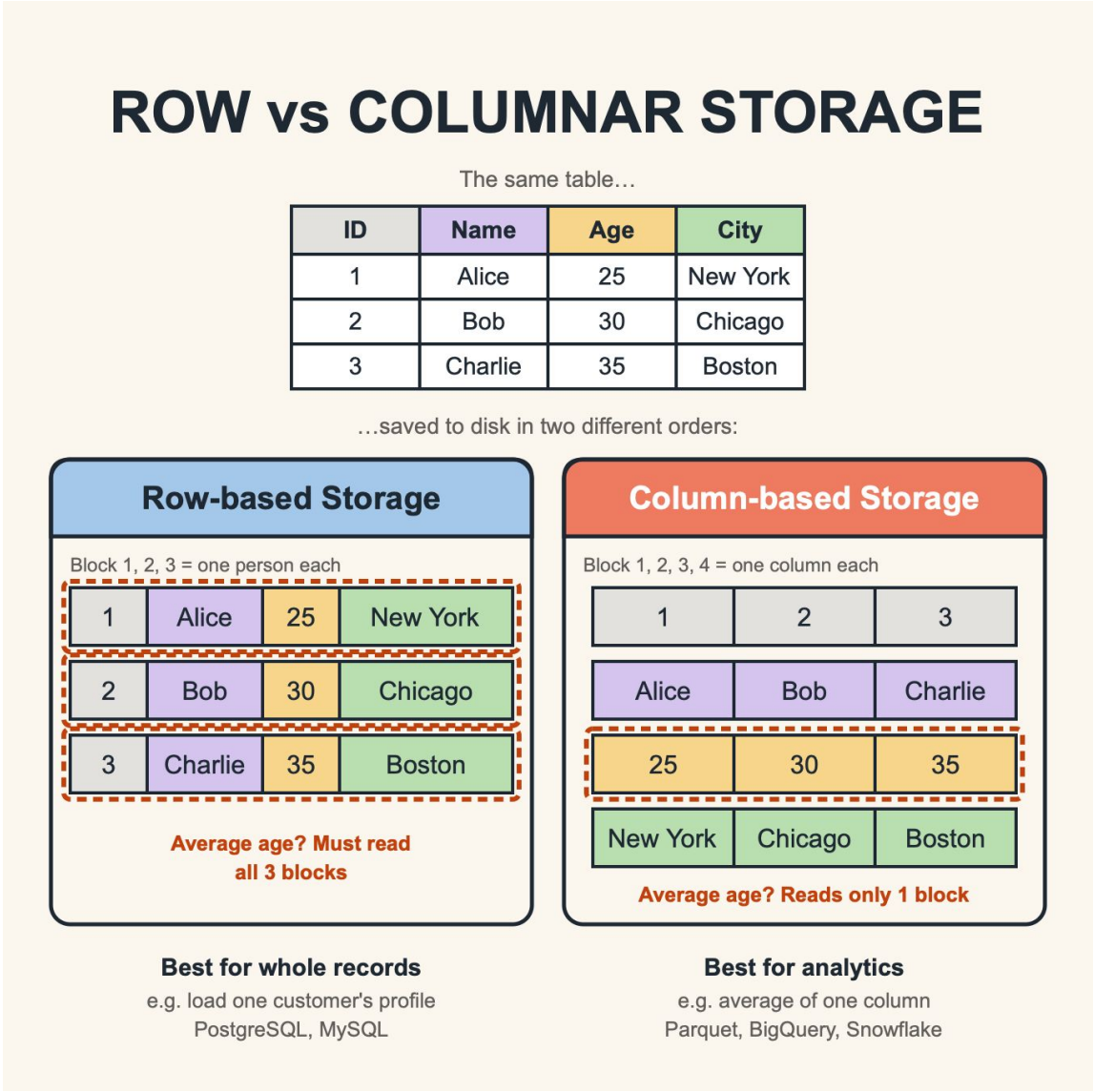
Yannis Foufoulas
johnfouf@athenarc.gr



Layered Architecture of the Agentic Hybrid DBMS



Row Store (NSM) vs. Column Store (DSM) Internals



Row Store (NSM) vs. Column Store (DSM) Internals

Row Store: N-ary Storage Model (NSM)

Physical Layout: All attributes of tuple i sit contiguously in a single struct/slotted page.

CPU Cache Pollution: Scanning only review pulls unused id, sentiment, and ts bytes into L1/L2 cache lines.

Strength: Single memory seek for INSERT, full-row UPDATE, or SELECT * WHERE id = 42.

Column Store: Decomposition Model (DSM)

Physical Layout: Each column is stored as a separate dense array

Zero I/O Waste: SELECT is_positive(review) touches only Col_Rev; vectorizes filters.

Weakness: Point updates and multi-column tuple selection

How Hybrid Storage Works

- **Implementation Language: C/C++**
- **Row Store Engine:** Slotted-page or contiguous array of RowTuple structs supporting fast append and in-place updates.
- **Column Store Engine:** Fixed-width arrays for integers/booleans plus an Arrow-style uint32_t offsets[] + char data[] buffer for variable-length review text.
- **Statistics Collector:** Maintains TableStats (row count N, distinct values NDV, avg string byte length, min/max zone maps, and selectivity histograms).
- **Unified Scan API:** Implements ScanIterator over both layouts.
- **Optimizations:** Compression (e.g., dictionary encoding, RLE encoding), cache friendly block based columnar storage (PAX https://www.vldb.org/conf/2001/P169.pdf)

Storage Agent

Decision Where Each Operator Runs

- **Input:** Logical JSON IR plan + TableStats from Storage Component .
- **Projectivity Ratio (k / M):** If an operator accesses k=1 column (review) out of M, route scan to COL_DSM.
- **Selectivity & Reconstruction Cost:** If a query fetches most columns for few filtered tuples (or reads hot uncommitted delta rows), route to ROW_NSM.
- **Deliverable:** Compare a Rule-Based approach against an LLM Agent Router that annotates every Scan and Join node with "store": "COL_DSM" | "ROW_NSM".

NL-to-SQL Translation, SQLGlot AST & JSON IR

3-Stage Pipeline

- **Stage 1 (NL-to-SQL Agent):** Schema-prompted LLM converts natural language questions into SQL containing registered UDFs (e.g., is_positive(review)).
- **Stage 2 (SQLGlot Parser):** Parses SQL via sqlglot.parse_one(sql), validates table/column names against Catalog, and extracts syntax tree.
- **Stage 3 (Logical JSON IR Lowering):** Translates the AST into a bottom-up operator tree (Scan, Filter, Project, HashJoin, EvalUDF).
- **Schema Validation:** Enforces strict JSON Schema before passing IR to Optimizer.
- **Supported queries:** Select, project, filter, join, UDF, insert, update

```
// Standardized JSON IR Contract
{
   "plan_id": "q_104",
   "root": {
     "op": "Project",
     "proj": ["review", "udf_out_1"],
     "child": {
       "op": "Filter",
       "pred": "udf:is_positive(review) == 1",
       "child": {
         "op": "Scan",
         "table": "reviews",
         "cols": ["id", "review"],
         "store": "UNDECIDED"
       }
     }
   }
 }
```

Query Verification & Semantic Validation Phase

NL-to-SQL Verification Protocol

- **1. Schema Conformity Check:** Verification that all referenced tables and columns exist in the active database Catalog.
- **2. UDF Registry Validation:** Ensures custom functions (e.g., is_positive(review)) are registered with valid signatures and expected return types.
- **3. Type Consistency & Binding:** Validates expression types, operator compatibility, and implicit casting rules across JOINs and WHERE clauses.
- **4. Semantic Error Feedback Loop:** If verification fails, static analysis diagnostic logs feed back into the LLM Agent to re-generate correct SQL or fallback to unoptimized plan.

```
// Verification Engine Result
STATUS: VERIFIED_VALID
{ "query_id": "q_104", "verification": {
"schema_check": "PASSED", "tables": ["reviews"],
"columns": ["id", "review"], "udf_check":
"PASSED", "udfs_validated": [ { "name":
"is_positive", "arg_type": "TEXT", "return_type":
"INT" } ], "type_binding": "OK" } }
```

Plan Rewriting & Optimization

RULE 1 • CSE MEMOIZATION

Eliminate common subexpressions

Rewrite SELECT udf(x) WHERE udf(x) into a single EvalUDF(x) AS tmp operator below Filter(tmp).

RULE 2 • FILTER & JOIN ORDERING

Push Down & Reordering

Order predicates and joins so that the total query cost is minimized.

RULE 3 • SAFETY & FALLBACK

Verify Semantic Equivalence

Validate agent output against JSON schema; fall back to deterministic plan on LLM hallucination.

```
Unoptimized: Scan(50K) → Filter[LLM_UDF: 50K calls] → Project[LLM_UDF: 25K calls] = 75,000 LLM Calls
```

```
Optimized: Scan(50K) → SQL_Filter(10K) → EvalUDF(10K) → Filter(tmp) = 10,000 LLM Calls (7.5× speedup)
```

Vectorized Execution Protocol

- **VectorBatch Model:** Instead of row-at-a-time next(), operators process chunks of BATCH_SIZE = 256..1024 tuples per call.
- **Selection Vectors:** FilterOperator updates an active index array without copying string payloads in memory.
- **In-Memory Hash Join:** Builds an index on the one input table and probes batches from the outer table.
- **ScalarUDF Operator:** Passes a full batch of active string pointers to the UDF Engine so LLM calls execute concurrently.

UDF Engine:
CFFI Bridge & Async LLM Batching Protocol

- **What is a UDF:** UDF stands for User Defined Functions, and this is a user code written in a procedural language (here in Python) that participates in a query plan and is called by the execution engine.

- **SQL Example:**

  select is_positive(review) from reviews;

  Python UDF:

  def is_positive(arg):
  # procedural logic
  return true

```
extern "C" int eval_llm_udf_batch(
  const char* udf_name,
  int32_t batch_size,
  const char** review_ptrs,
  uint8_t* out_bool_results,
  uint8_t* out_null_bitmask
);

# Inside Python CFFI callback:
# asyncio.run(dispatch_batch(reviews, sem=K))
```

UDF Engine:
CFFI Bridge & Async LLM Batching Protocol

Solving the C++ to Python LLM Bottleneck

- **CFFI ABI/API Bridge:** Exposes a C-compatible function pointer registered into the C++ ScalarUDFOperator at startup.
- **Asyncio Pool:** Receives tuple strings from C++ and dispatches concurrent async LLM requests.
- **Prompt Concurrency Benchmark:** Empirically sweep K in {1, 4, 16, 64, 128} to find the LLM server's saturation point.

```
extern "C" int eval_llm_udf_batch(
    const char* udf_name,
    int32_t batch_size,
    const char** review_ptrs,
    uint8_t* out_bool_results,
    uint8_t* out_null_bitmask
);

# Inside Python CFFI callback:
# asyncio.run(dispatch_batch(reviews, sem=K))
```

Concurrency Control

The Problem: Slow LLM Queries vs. Fast Updates

- **Long-Lived Analytical Reads:** A query running is_positive(review) over thousands of rows takes seconds or minutes due to LLM latency.
- **Concurrent OLTP Writes:** Simultaneously, users execute INSERT, DELETE, or UPDATE reviews SET sentiment = ...
- **Why Locks (2PL) is not the best here:** Shared read locks would block all updates for minutes; exclusive write locks would stall LLM pipelines

The Solution: Multi-Version Concurrency (MVCC)

- **Readers Never Block Writers:** Updates create a new tuple version instead of overwriting active memory in place.
- **Consistent Snapshot:** Each query receives a monotonic timestamp read_ts at start and sees only data committed before read_ts.
- **Hybrid Sync:** Updates land in the Row Store + Column Deletion Bitmap immediately, and merge asynchronously into the Column Store.

Concrete MVCC Protocol

Protocol Rules & Architecture Integration

- **Tuple Version Header:** Every tuple stores xmin (creation commit timestamp), xmax (deletion/invalidation timestamp, \infty if active), and prev_ptr.
- **Snapshot Visibility Rule:** In Component 1's get_batch(read_ts), tuple T is visible iff $\text{xmin} \le \text{read_ts} < \text{xmax}.
- **Write-Write Conflict (First-Committer-Wins):** To update T, transaction uses atomic CAS(&T.xmax, INF, txn_id); if xmax != INF, abort transaction.
- **Hybrid Store Sync:** An UPDATE sets bit i in Column Store's DeletionBitmap, appends new version to Row Delta Buffer, and background thread compacts into Column Store.

```
// MVCC Transaction & Visibility (Team 7)
struct VersionHeader {
   std::atomic xmin;
   std::atomic xmax; // INF = 0
   RowTuple* prev_version;
 };


 bool is_visible(const VersionHeader& h,
                 uint64_t read_ts) {
   uint64_t b = h.xmin.load();
   uint64_t e = h.xmax.load();
   return (b ⩽ read_ts) &&
          (e == 0 || read_ts < e);
 }
```

Recovery

- **Volatile RAM Problem:** Both Row Store and Column Store buffers live in memory; a process crash wipes uncommitted and uncheckpointed updates.
- **Dual-Layout Desync Risk:** If a crash occurs after updating the Row Store but before updating the Column Store, the database becomes corrupted.
- **Write-Ahead Rule:** Every modification must be appended and flushed (fsync) to an append-only disk log before commit returns.

UDF Fault Tolerance (LLM Query Resume)

- **Expensive Computation Loss:** Suppose a query evaluates is_positive(review) on 50,000 rows and crashes at row 45,000.
- **Wasted Money & Time:** Re-running the query naively re-invokes 45,000 paid LLM API calls from scratch.
- **Durable UDF Result Log:** Persisting deterministic (udf_id, text_hash) → output batches on disk lets C4 skip already-evaluated tuples on restart.

* Recovery will be implemented either for UDFs or for SQL transactions according the the components will be selected. Not for both

Concrete Recovery Protocol

3-Phase Recovery Protocol

- **Binary WAL Record Format:** Fixed header [LSN, txn_id, prev_lsn, type, table_id, tuple_id] + payload [before_img, after_img].

- **ARIES Restart* (3 Passes): 1. Analysis:** Scan WAL forward from last checkpoint to find winner & loser transactions. **2. Redo:** Replay all logged changes forward into Row & Column stores. **3. Undo:** Roll back uncommitted loser transactions in reverse order.

```
// WAL Record & Recovery API (Team 8)
enum LogType { BEGIN, INSERT, UPDATE,
               COMMIT, ABORT, CHKPT };


 struct WALRecord {
   uint64_t lsn;
   uint64_t txn_id;
   uint8_t log_type;
   uint32_t tuple_id;
   uint32_t payload_len;
 };


 void recover_database(HybridTable* tbl);
```

*https://medium.com/@vikas.singh_67409/algorithms-for-recovery-and-isolation-exploiting-semantics-aries-d904765fb9b8

Component Selection Matrix

ID	Component Name	Protocol / Mechanism	Core Deliverable & Interface
C1	**Hybrid Storage Engine**	NSM Row + DSM Column + Stats	C++ dual buffers, zone maps, get_batch() API
C2	Storage Router Agent	Cost Model + LLM Routing Agent	Per-operator ROW_NSM vs COL_DSM tagging
C3	**Execution Engine**	Vectorized Batch Volcano Model	C++ Scan, Filter, Project, HashJoin, UDF
C4	Async LLM UDF Engine	CFFI ABI + Python asyncio Pool	Semaphore-batched LLM calls & concurrency study
C5	**Agentic Query Optimizer**	CSE Memoization + Filter Pushdown	Cost-aware JSON IR rewriter & validator
C6	**NL2SQL & SQLGlot Parser**	LLM NL2SQL + AST-to-IR Lowering	Natural language & SQL to validated JSON IR
C7	Concurrency Control	Timestamp MVCC + Deletion Bitmap	xmin/xmax snapshot isolation & delta compaction
C8	Recovery & Fault Tolerance	ARIES-Lite WAL + UDF Hash Cache	Redo/Undo crash recovery & idempotent UDF log