CUBRID Query Processing

Parser, Optimizer, XASL, Executor, Access Methods

2026-05 · Code Analysis Seminar

© 2026 CUBRID Corporation. All rights reserved.

Agenda

1. **Why a query pipeline** — SQL is declarative; the engine materializes a **plan**.
2. **The full pipeline** — one diagram of the four stages and the runtime helpers.
3. **Front-end** — parser, semantic check, query rewrite.
4. **Middle-end** — optimizer, XASL generator, XASL cache.
5. **Back-end** — Volcano executor, scan manager, list-file spool.
6. **Runtime helpers** — predicate evaluator, post-processing.

> Closing: **Beyond CUBRID** — research frontiers.

© 2026 CUBRID Corporation. All rights reserved.

Why a query pipeline at all

SQL is **declarative** — **"give me these rows"**. The engine has to invent a plan that produces them. That invention is the query pipeline.

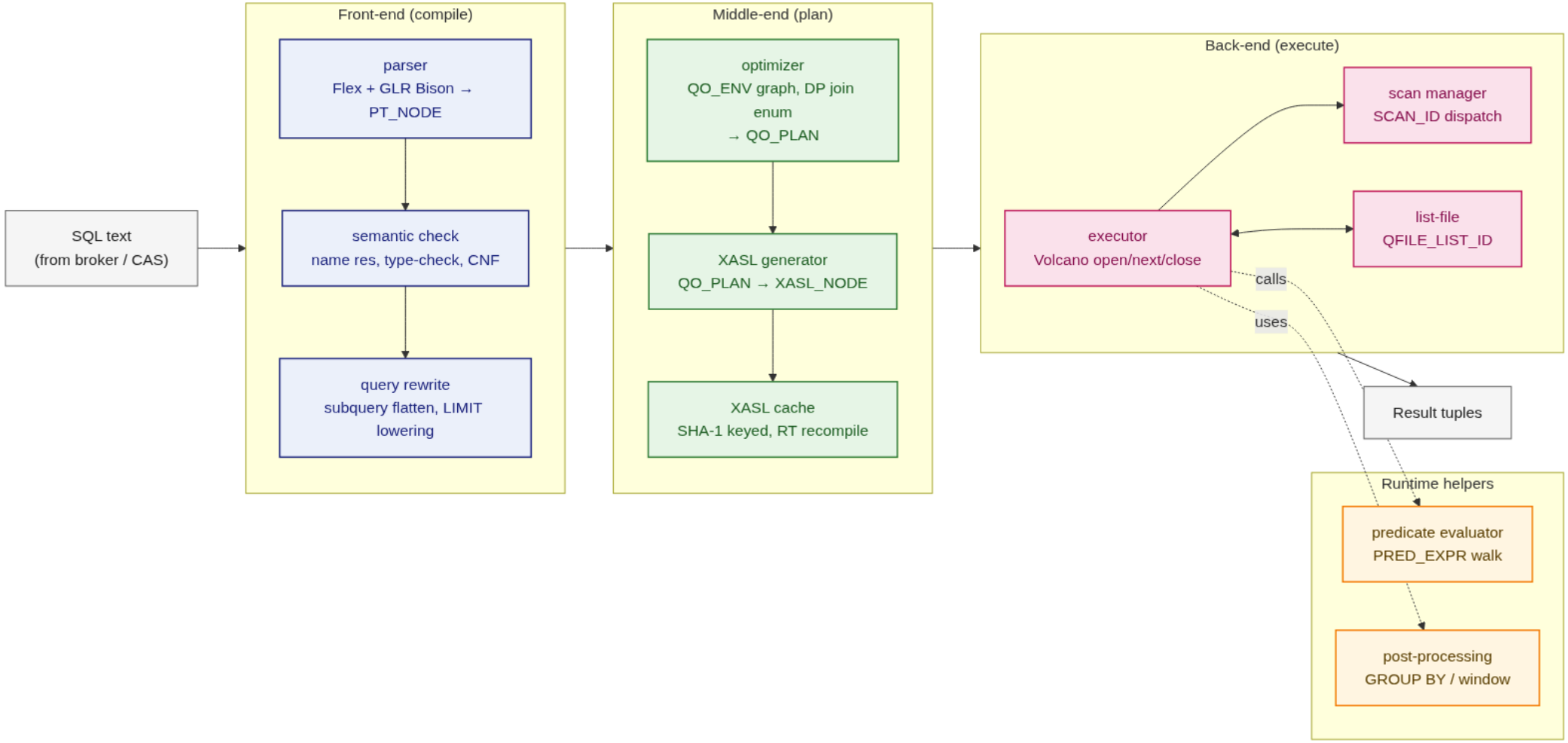
Question	Stage that answers it
Is this even a SELECT?	parser → `PT_NODE`
Does `t.col` exist? Is it the right type?	semantic check
Can the subquery become a semijoin?	query rewrite
Which join order is cheapest?	optimizer (DP enumeration)
In what shape does the executor want this?	XASL generator → `XASL_NODE`
Have we seen this SQL before?	XASL cache (SHA-1 keyed)
How do I actually fetch row 1?	executor → scan manager → access method
Where do I put the intermediate result?	list-file (`QFILE_LIST_ID`)

- **Classical 7-stage architecture** — Selinger 1979, System R. Every major RDBMS shares this skeleton.
- The pipeline **materialises SQL into an iterator plan** plus a cache that skips compile on re-execute.

> Scope: broker/CAS in; storage stack (heap, B+Tree, page buffer, MVCC) out.

© 2026 CUBRID Corporation. All rights reserved.

The full pipeline



On re-execute the cache short-circuits FE + ME entirely; the executor reads the cached plan directly.

© 2026 CUBRID Corporation. All rights reserved.

Five common DBMS patterns

Five dials almost every modern DBMS sets in some way:

1. **AST → plan IR → executor IR.** Three IRs, one per pipeline stage. CUBRID's are `PT_NODE` → `QO_PLAN` → `XASL_NODE`.
2. **Cost-based join enumeration with pruning.** Dynamic-programming over join orderings, costed against statistics. Selinger 1979 set the template.
3. **Plan cache keyed by parameterized-SQL hash.** Compile once, execute many. The compile half is the expensive half.
4. **Iterator pull executor (Volcano).** Each operator exposes `open / next / close`; the consumer pulls tuples down through the tree. Back-pressure is automatic.
5. **Spill-to-temp for intermediate results.** Sort runs, hash builds, group-by accumulators, the final result for cursor read-back — all materialised through one substrate.

> CUBRID is **one point** in this dial-space. The same five patterns reappear in Postgres, MySQL, SQL Server, Oracle.

© 2026 CUBRID Corporation. All rights reserved.

Part II

Stage by stage

© 2026 CUBRID Corporation. All rights reserved.

Parser — Flex + Bison → `PT_NODE`

The front door. Turn a string into a tree.

- **Lexer** — `Flex` tokenizer. Keywords, identifiers, literals, operators.
- **Parser** — `GLR Bison` grammar. GLR handles SQL's ambiguities — same prefix can mean different productions.
- **Output IR** — `PT_NODE`, a polymorphic tagged-union tree. Each node: a `node_type` tag, a dispatch into three function-pointer arrays (init / apply / free), sub-info via `union info`, and `next` for sibling lists.
- **Allocator** — `PT_NODE` allocated from a per-`PARSER_CONTEXT` block allocator; freed on context destroy.

`node_type`	What it represents
`PT_SELECT`	a SELECT statement (FROM, WHERE, GROUP BY, ORDER BY)
`PT_EXPR`	an expression node — `a + b`, `f(x)`, `a > b`
`PT_NAME`	a column reference, function name, or alias
`PT_VALUE`	a literal — int, string, null

> The whole front-end and middle-end **mutates this tree in place**. Its shape is the prerequisite for reading any later stage.

© 2026 CUBRID Corporation. All rights reserved.

Semantic check — resolve, type, CNF

The parser produces shape, not meaning. Semantic check annotates the tree with meaning.

Four passes, all driven by `pt_check_with_info`:

1. **Name resolution.** Every `PT_NAME` must bind to something — a column in scope, a function, a parameter. Consults the catalog. Failed binding → `error 213, "Unknown column ..."` at this stage.
2. **Where-clause aggregate check.** Aggregates (`SUM`, `COUNT`) belong in `SELECT` / `HAVING`, not `WHERE`. Catches `WHERE COUNT(*) > 0` early.
3. **Host-variable replacement.** Bound parameters (`?` placeholders) get their type info from the supplied bind.
4. **`semantic_check_local`** — the heavy pass. Recurses with `pt_semantic_type` to check operand types, insert implicit coercions, and constant-fold literal-only sub-expressions (`1 + 2` → `3`).

Then **`pt_cnf`** rewrites every WHERE clause into **conjunctive normal form** — `(A OR B) AND (C OR D)`. CNF is the shared language: rewriter pushes conjuncts, optimizer costs each, executor evaluates in `eval_pred`.

> Every node carries type / OID / coercion annotations; the rewriter inherits a **typed** tree.

© 2026 CUBRID Corporation. All rights reserved.

Query rewrite — flatten, inline, lower

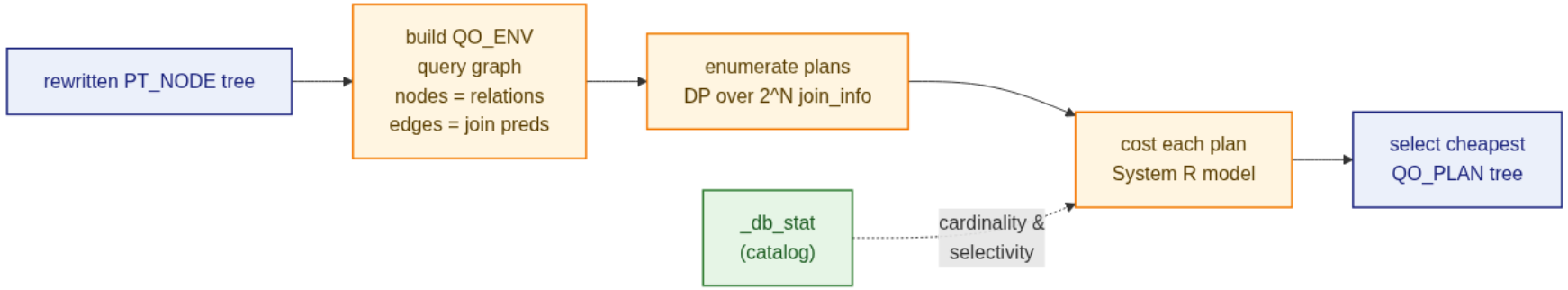
`mq_rewrite` and friends — the algebraic transform pass. Operates on a fully-resolved, type-checked, CNF'd tree. The goal is to **expose work to the optimizer** that was hidden inside the source SQL.

Transform	Example before	After	Why
Subquery flattening	`WHERE EXISTS (SELECT 1 FROM s WHERE s.x = t.x)`	semi-join `t ⋉ s`	Subqueries are opaque to join enumeration; semijoins are not.
View inlining	`SELECT * FROM v WHERE ...`	inline body of `v`	Push predicates **into** the view's FROM list.
Predicate reduce	`WHERE A AND TRUE AND B AND B`	`WHERE A AND B`	Smaller CNF → faster optimizer.
Outer-join reduction	LOJ with null-rejecting predicate on right side	inner join	More join-order options.
Redundant-join elimination	`t JOIN t2 ON t.id = t2.id` where `t2.id` is PK and no `t2` columns read	drop `t2`	Saves a join.
Auto-parameterization	`WHERE x = 7`	`WHERE x = ?` (with bound `7`)	One cached plan per SQL shape.
LIMIT lowering	`... LIMIT 10`	`INST_NUM / ORDERBY_NUM / GROUPBY_NUM` predicate	Lets the executor **stop** at row 10 instead of producing all then trimming.

> LIMIT lowering ripples across rewrite, plan generation, and runtime — one feature, three stages.

© 2026 CUBRID Corporation. All rights reserved.

Optimizer — `QO_ENV` graph + DP join enumeration



- `QO_ENV` — the **query graph**. One node per relation in FROM; one edge per join predicate. Anything not bound to a join edge stays as a residual filter.
- **Dynamic-programming join enumeration.** A `join_info` vector of size `2^N` indexed by **which subset of relations is joined so far**. For each subset, keep the cheapest plan. Build up from singletons to the full N-set. **Partial-then-total** to keep memory bounded.
- **System R cost model.** Each plan node costs as `fixed_cpu + fixed_io + variable_cpu * card + variable_io * card`. Cardinalities flow from input statistics through per-operator selectivity estimates.
- **Output IR** — `QO_PLAN`, a tree of plan nodes. Each node has type (`seq scan`, `idx scan`, `NL join`, `merge join`, `sort`, `groupby`, …), input plans, applied predicates, and an estimated cost.

> Bad statistics → bad cost → bad plan. The RT recompile guard is the main mitigation.

© 2026 CUBRID Corporation. All rights reserved.

Cost model basics

A cost-based optimizer is exactly as good as its cost model — which is exactly as good as its statistics.

- **Cardinality** — how many rows does this operator emit?
  - Comes from `_db_stat` for base relations; from per-operator estimates for derived ones.
  - `seq scan(R)` → `|R|`.
  - `σ(R, p)` → `|R| * selectivity(p)`.
  - `R ⋈ S` on equi-key → `|R| * |S| / max(distinct(R.key), distinct(S.key))`.
- **Selectivity** — fraction of rows that pass a predicate.
  - Per-column histograms when available; back-off to **default constants** (e.g. `0.1` for `=`, `0.333` for `<`) when not.
  - `qo_equal_selectivity`, `qo_range_selectivity` are the consumer APIs.
- **IO term** — pages read. Index access costs `tree_height + leaf_pages_scanned`; sequential costs `pages(R)`.
- **CPU term** — per-tuple work. Scaled by an internal constant; rarely the dominant factor on a real workload.

> Same SQL + same catalog → same plan. Data growth causing cardinality drift is what the RT recompile check watches.

© 2026 CUBRID Corporation. All rights reserved.

XASL generation — `QO_PLAN` → `XASL_NODE` tree

XASL is the **executor IR**. Where `QO_PLAN` says **"merge-join R with S"**, XASL says **"open scan over R, open scan over S, drive merge with `OUTPTR_LIST`, evaluate `pred` per probe."**

- **One `XASL_NODE` per relational operator**, with three child-slot conventions:

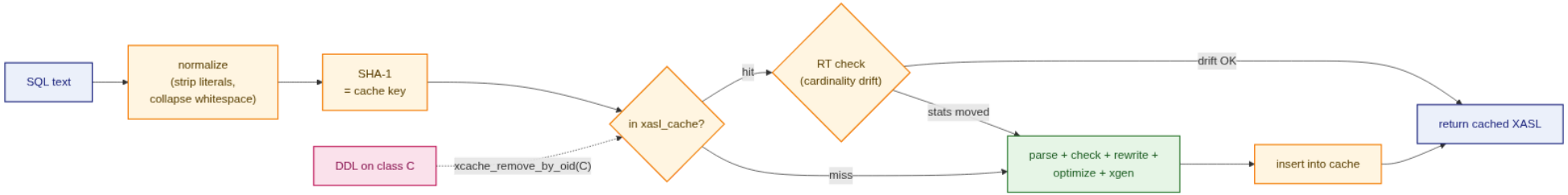
Slot	Holds	Meaning
`aptr_list`	sub-XASL tree	**uncorrelated sub-queries** materialised before this node runs
`dptr_list`	sub-XASL tree	**correlated sub-queries** re-evaluated per outer row
`scan_ptr`	sub-XASL tree	child operator feeding **this** one

- **Sub-IRs** carried by every `XASL_NODE`:
  - `ACCESS_SPEC` — which relation, which index, which range, which heap.
  - `OUTPTR_LIST` — projection list — what tuple shape this node emits.
  - `PRED_EXPR` — the predicate tree for the WHERE / ON clause attached here.
  - `REGU_VARIABLE` — operand resolver (constant / column / arithmetic expression).
- **Serialisation.** `xts_*` fixes up internal pointers as integer offsets into one flat byte buffer for cache, ship, and checksum.

© 2026 CUBRID Corporation. All rights reserved.

> `aptr` runs once; `dptr` runs per outer tuple. Misreading those is the most common bug in XASL-level patches.

XASL cache — SHA-1 keyed plan cache



- **Key** = SHA-1 of the **rewritten** SQL with literals stripped — **"hash text"**. So `WHERE id = 7` and `WHERE id = 12` collide into one cache entry. Auto-parameterization (slide 9) is what makes this safe.
- **Latch-free, server-wide hashmap.** Multiple sessions reuse one entry concurrently via a `cache_flag` reference count.
- **RT recompile.** A drift-detection guard: if observed cardinality moves too far from the cardinality the plan was costed against, the cached plan is invalidated and recompiled. Defends against stat staleness.
- **Per-class invalidation.** DDL on class `C` fires `xcache_remove_by_oid(C)`. Every cached XASL that references `C` is dropped. Hooked from the DDL execution path.

> The cache is the **hot path**: on a prepared-statement re-execute, the front-end and the rest of the middle-end are skipped entirely. The executor reads the cached plan directly.

© 2026 CUBRID Corporation. All rights reserved.

Volcano executor — open / next / close

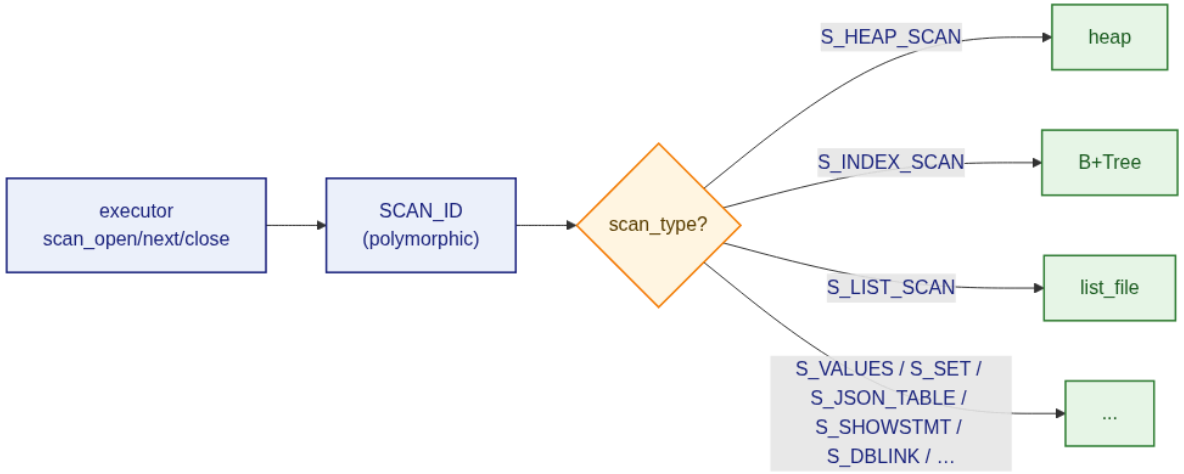
The back-end's executor is a **textbook Volcano iterator**.

- One method triple per operator: `open()` initialises state, `next()` produces the next tuple or EOF, `close()` tears down.
- Tuples are **pulled**, not pushed. Rows climb the tree on demand.
- **Back-pressure is automatic** — operators below stall until the next `next()` arrives. `LIMIT 10` stops the whole sub-tree after row 10.

```c
// qexec_execute_mainblock_internal — src/query/query_executor.c
int run (THREAD_ENTRY *th, XASL_NODE *xasl)
{
  scan_open (xasl, &sid);                  // open
  while (scan_next (&sid, &tuple) == OK)   // next
    {
      if (!eval_pred (xasl->where, tuple)) continue;
      list_file_append (xasl->list_id, project (xasl->outptr, tuple));
    }
  scan_close (&sid);                       // close
  return NO_ERROR;
}
```

© 2026 CUBRID Corporation. All rights reserved.

Scan manager — `SCAN_ID` dispatch



One uniform `open / next / close` interface upward.

© 2026 CUBRID Corporation. All rights reserved.

Access methods

One line each. All present the same `SCAN_ID` interface upward.

Scan type	Source	One-line job
`S_HEAP_SCAN`	`cubrid-heap-manager.md`	sequential scan over a class heap; honours MVCC visibility per row.
`S_INDEX_SCAN`	`cubrid-btree.md`	range scan over a B+Tree; key range + optional residual filter; OID set output.
`S_LIST_SCAN`	`cubrid-list-file.md`	re-scan a previously materialised list file (sub-query result, sort output, hash build side, group-by accumulator).
`S_VALUES_SCAN`	`query_executor.c`	iterate a literal `VALUES (...)` clause — no IO.
`S_SET_SCAN`	`query_executor.c`	iterate a SQL set / multiset value.
`S_JSON_TABLE_SCAN`	`cubrid-json-table.md`	iterate the rows produced by a `JSON_TABLE(...)` expression.
`S_SHOWSTMT_SCAN`	`cubrid-show-commands.md`	iterate the server-introspection rows produced by `SHOW <X>`.
`S_DBLINK_SCAN`	`cubrid-dbi-cci.md`	iterate rows fetched from a remote database over `dblink`.
`S_CLASS_ATTR_SCAN` / method	`query_executor.c`	iterate class attributes or invoke a stored method.

> Access method = how tuples enter the pipeline. Every other operator consumes from one of these; everything else is shaping.

© 2026 CUBRID Corporation. All rights reserved.

Predicate evaluation — `PRED_EXPR` + `REGU_VARIABLE`

The same evaluator runs at every site that needs **"does this tuple match?"** — scan filters, join conditions, post-processing.

- **`PRED_EXPR` tree.** Two leaf kinds: `T_PRED` (boolean operator: `AND`, `OR`, `NOT`) and `T_EVAL_TERM` (leaf comparison: `x = y`, `a < b`, `s LIKE p`).
- **`REGU_VARIABLE` operands.** Resolves to: a constant, a column reference, an arithmetic expression (via `qdata_*`), a function call, or a host parameter.
- **Three-valued logic.** `TRUE / FALSE / UNKNOWN`; Kleene's rules — `TRUE AND UNKNOWN = UNKNOWN`, `FALSE AND UNKNOWN = FALSE`.

```
// eval_pred — src/query/query_evaluator.c
DB_LOGICAL eval_pred (THREAD_ENTRY *th, PRED_EXPR *p, VAL_DESCR *vd, OID *o)
{
  switch (p->type)
    {
    case T_PRED:      return pred_combine (eval_pred (p->lhs), eval_pred (p->rhs), p->op);
    case T_EVAL_TERM: return eval_term (th, &p->term, vd, o); // fetch_peek_dbval
    case T_NOT_TERM:  return v_not (eval_pred (p->child));
    }
}
```

© 2026 CUBRID Corporation. All rights reserved.

One evaluator, four call sites: `key_pred` (index), `where_pred` (scan), `merge_pred` / `inner_pred` (join), post-processing.

Post-processing — group-by, aggregate, window, order-by

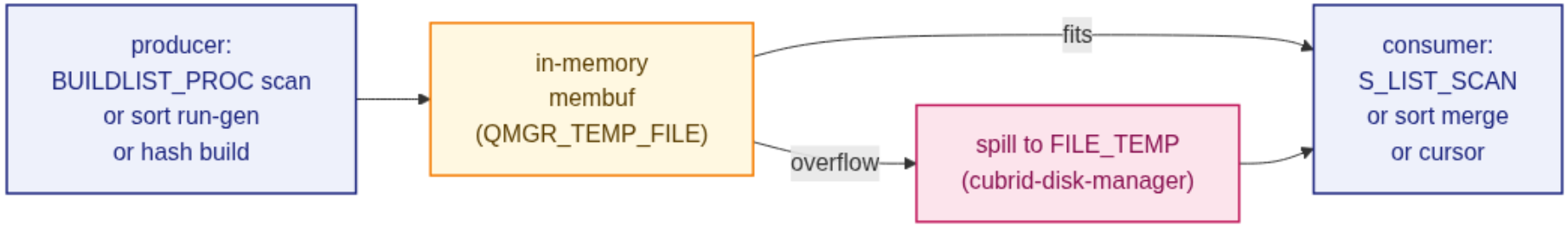
Operators that **force materialisation** — they need to see all input rows before they can emit anything. They sit on top of the Volcano spine; their input is some `BUILDLIST_PROC` that fed a list file.

- `qexec_groupby` — GROUP BY.
  - Strategy choice at plan time: **sort-based** vs **hash-based**.
  - Sort-based: sort on the group key (external sort over `FILE_TEMP`), then walk runs of equal keys.
  - Hash-based: build a `HASH_TABLE` keyed by the group expression, fold each row into its accumulator. Spills to external sort when memory exceeds `max_agg_hash_size`.
- **Aggregates.** `SUM`, `COUNT`, `MIN`, `MAX`, `AVG`, … run as per-group accumulators. `COUNT(DISTINCT x)` needs its own hash set per group.
- **Window / analytic functions.** `qexec_execute_analytic` materialises into list files per partition, sorts within each partition, then walks the rows applying the frame (`ROWS BETWEEN ...`, `RANGE BETWEEN ...`). One pass per analytic in the same SELECT.
- **ORDER BY.** A sort iterator on top of the executor, fed by an external-sort run-gen + k-way merge.

> Common substrate: `list_file` for materialisation; `external_sort` for ordering.

© 2026 CUBRID Corporation. All rights reserved.

`list_file` — `QFILE_LIST_ID` + FILE_TEMP spill



- Every materialised tuple stream in CUBRID — **sub-query result, sort run, hash build side, group-by accumulator, final result for the cursor** — is one `QFILE_LIST_ID`.
- **Linked-page abstraction.** A `QFILE_LIST_ID` is a chain of pages, each carrying a header and a sequence of tuples. The reader gets one page at a time; the writer appends.
- **Two-tier storage.**
  - Small intermediate? Stays in a **per-query membuf** — `QMGR_TEMP_FILE`.
  - Doesn't fit? Spills to **`FILE_TEMP`** — temp file allocated through `cubrid-disk-manager`. Transparent to the producer / consumer.
- **Reusable.** A list file written once can be `S_LIST_SCAN`'d many times — that's how the inner side of an NL join can be re-scanned per outer tuple without re-running the inner sub-tree.

> `list_file` is the single materialisation substrate — every per-stage spill, sort, and cursor goes through it.

© 2026 CUBRID Corporation. All rights reserved.

Source walkthrough — where to read next

Symbol names are the stable handle. Line numbers drift with refactors; `git grep -n '<symbol>' src/` is your friend.

Stage	Symbol(s)	File
Parser entry	`parser_parse_string_with_escapes` · `PT_NODE`	`src/parser/parser.h` · `parse_tree.h`
Semantic check	`pt_check_with_info` · `pt_semantic_type` · `pt_cnf`	`src/parser/semantic_check.c`
Query rewrite	`mq_rewrite` · `qo_rewrite_query`	`src/parser/query_rewrite.c`
Optimizer	`qo_optimize_query` · `QO_ENV` · `QO_PLAN`	`src/optimizer/query_planner.c`
XASL generation	`pt_to_xasl` · `gen_outer` · `XASL_NODE`	`src/query/xasl_generation.c` · `xasl.h`
XASL cache	`xcache_find_by_query_string` · `xcache_remove_by_oid`	`src/query/xasl_cache.c`
Executor	`qexec_execute_mainblock_internal` · `qexec_groupby`	`src/query/query_executor.c`
Scan manager	`scan_open` · `scan_next_scan` · `SCAN_ID`	`src/query/scan_manager.c`
Pred. evaluator · List file	`eval_pred` · `PRED_EXPR` · `qfile_open_list` · `QFILE_LIST_ID`	`src/query/query_evaluator.c` · `list_file.c`

> Each stage has a dedicated detail doc linked from `cubrid-overview-query-processing.md`.

© 2026 CUBRID Corporation. All rights reserved.

Beyond CUBRID — research frontiers

The four-stage Selinger pipeline is universal, but every layer has alternative formulations in the literature and in modern engines.

Direction	One line
Cascades (Graefe 1995)	Top-down memo-driven optimiser. SQL Server, Greenplum/Orca, Apache Calcite, CockroachDB. Generalises Volcano's bottom-up DP into pattern-driven rule firing over a shared memo.
Volcano-style executor	The base pattern CUBRID uses. Modern engines extend with code-generation: Hyper / Umbra / DuckDB compile pipelines to native code rather than interpret.
Vectorised executors	DuckDB, Velox, ClickHouse, Photon. Operate on column batches (1K rows) instead of one tuple at a time — amortises dispatch cost, enables SIMD.
Adaptive query execution	Redshift, Spark AQE, SQL Server. Re-plan mid-execution when observed cardinalities diverge from the optimizer's estimates. Closes the loop the RT-recompile mechanism only partially covers.
Learned cardinality estimation	Neo (MIT), Bao (MIT), MS Auto-CE. Replace selectivity constants with ML models trained on workload history.
LLM-assisted plan rewriting	DB-GPT (Microsoft), LLM-R2. Use LLMs as a rewrite rule discovery engine — produce candidate transformations, regression-test them on the workload.
Query compilation	LLVM-based JIT (Postgres JIT, Umbra). Compile a tight loop per XASL instead of interpreting the iterator tree.

> CUBRID: System R + DP + Volcano + SHA-1 cache — a defensible position with a clear roadmap for where the engine could move.

© 2026 CUBRID Corporation. All rights reserved.

Thank you

Q & A

- Analysis: `knowledge/code-analysis/cubrid/cubrid-overview-query-processing.md`
- Detail docs (per stage): `cubrid-parser.md`, `cubrid-semantic-check.md`, `cubrid-query-rewrite.md`, `cubrid-query-optimizer.md`, `cubrid-xasl-generator.md`, `cubrid-xasl-cache.md`, `cubrid-query-executor.md`, `cubrid-scan-manager.md`, `cubrid-list-file.md`, `cubrid-query-evaluator.md`
- Code: `src/parser/` · `src/optimizer/` · `src/query/`

© 2026 CUBRID Corporation. All rights reserved.