IndicatorX

High-Performance ProgramIndicator Analytics for DHIS2
Bypassing the service layer — reducing hours-long queries to minutes

DHIS2 Annual Conference 2026
Oslo, Norway · June 2026

Raja Zubair Asghar | HISP Pakistan

Hours → Minutes	~5×	5	Zero	2
QUERY TIME REDUCTION	FEWER EVENT TABLE SCANS (LATERAL JOINS)	PARALLEL QUERIES, BOUNDED CONCURRENCY	DHIS2 SERVICE LAYER DEPENDENCY AT QUERY TIME	EXECUTION MODES — DIRECT SQL + SQL VIEW FALLBACK

THE PROBLEM

DHIS2's built-in analytics API evaluates ProgramIndicators by running correlated subqueries against event tables. On national-scale deployments with millions of tracker events:

30 min – 3+ hr per query · Blocked reporting workflows · Correlated subquery per DE

Each referenced data element in a ProgramIndicator expression generates a separate correlated subquery — scanning the event table once per element, per enrollment, per period. At national scale, this becomes untenable.

THE INDICATORX APPROACH

IndicatorX pre-translates PI expressions to optimised SQL **once**, caches the result in SQLite, and executes directly against DHIS2's PostgreSQL `analytics_enrollment_*` and `analytics_event_*` tables — bypassing the DHIS2 application layer entirely.

Read-only DB connection · SHA-256 change detection · No DHIS2 restart needed

After the first sync, the expensive SQL translation step never runs again unless a PI expression actually changes — detected automatically via content hashing.

DUAL EXECUTION MODES

Mode	Condition	Query Path
Direct SQL	PostgreSQL pool connected	SQL runs directly against analytics tables, up to 5 parallel
SQL View	Pool is `None` — DB unreachable or not configured	Queries routed via DHIS2 `/api/sqlViews` — same cached SQL

Both modes use the same SQL cache. The active mode is shown as a badge in the app header. All features remain available in both modes.

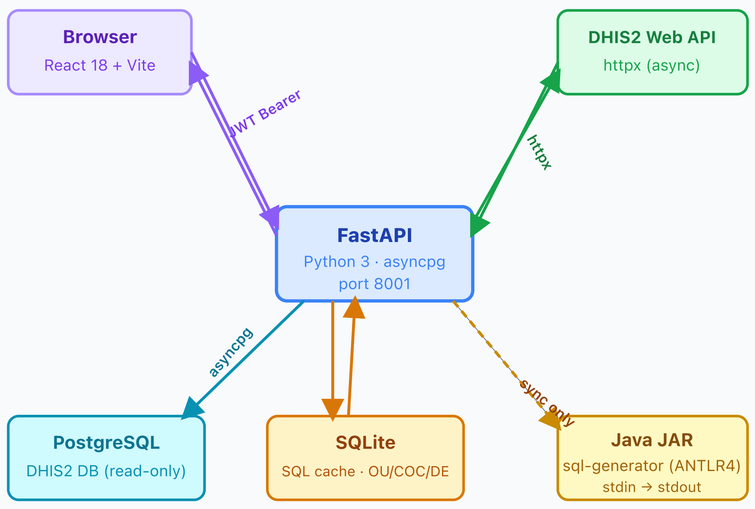
EXPORT & VALIDATE

Export to DHIS2: Push query results back as DHIS2 aggregate data values (`POST /api/dataValueSets`). Maps PI → DataElement via a configurable attribute UID; resolves COC from SQLite cache — no live DB read at export time.

Compare with DHIS2: Run the same query against DHIS2's own `/api/analytics` and compare row-by-row (tolerance 1e-9). Reports matched, value differences, and rows unique to each system.

Benchmark: Times the equivalent DHIS2 analytics API call and displays a race-bar comparison against IndicatorX speed — giving a concrete, reproducible performance figure.

SYSTEM ARCHITECTURE



One long-running process — the Python app — serves both the React SPA and all API endpoints from a single port. The Java JAR is a short-lived subprocess invoked only during sync.

CORE WORKFLOWS

① Sync (SQL Generation)

1. **Fetch PI metadata**
   PostgreSQL (DB mode) or DHIS2 REST API (SQL View mode)
2. **SHA-256 change detection**
   Hash expression + filter + name + type → compare with `pi_sql_store`
3. **Invoke Java JAR (new / changed PIs only)**
   ANTLR4 parses expression → LATERAL JOIN rewriting → EXISTS boundary SQL
4. **Store + validate in SQLite**
   `SELECT … WHERE 1=0` validation · refresh OU/COC/DE caches

② Query Execution

1. **Resolve periods & org units**
   `2026Q1` → `(2026-01-01, 2026-04-01)` · LEVEL/DESCENDANTS use JOIN (not large array)
2. **Load SQL fragments from SQLite**
   Retrieve `expr_sql` , `filter_sql` , `lateral_joins` , `boundary_sql` per PI
3. **Execute in parallel (asyncio semaphore = 5)**
   Boundary PIs: one query per period · Normal PIs: single bulk query for all periods

GENERATED SQL (SIMPLIFIED)

```sql
SELECT COUNT(enrollment) AS value, '2026Q1' AS period, ax.ou
FROM   analytics_enrollment_<programUid> ax
JOIN   organisationunit _ou
         ON _ou.uid = ax.ou AND _ou.hierarchylevel = 6
LEFT JOIN LATERAL (          ← rewritten by JAR
    SELECT "DE1", "DE2"
    FROM   analytics_event_<programUid>
    WHERE  enrollment = ax.enrollment
      AND  ps = 'stageUid'
    ORDER BY occurreddate DESC LIMIT 1
) _lj0 ON true
WHERE  (EXISTS (          ← EXISTS boundary
    SELECT 1 FROM analytics_event_<programUid>
    WHERE  enrollment = ax.enrollment
      AND  cast("dateDE" as date) >= '2026-01-01'
      AND  cast("dateDE" as date) <  '2026-04-01'))
  AND  (_lj0."DE1" = 1 AND _lj0."DE2" = 'VALUE')
GROUP BY ax.ou
```

THREE KEY SQL OPTIMISATIONS

LATERAL JOIN Rewriting

`N correlated subqueries` → `1 LEFT JOIN LATERAL per stage`

The Java JAR groups all correlated subqueries by **(eventTable, stageUid)** and rewrites them into a single `LEFT JOIN LATERAL` . All referenced data elements are read in one pass. Result: **~5× fewer event table scans** per PI.

EXISTS-Based Boundary SQL

`LIMIT 1 scalar subquery` → `EXISTS (… AND date IN range)`

The old approach checked only the *most recent* event — giving wrong results for historical periods. The EXISTS pattern checks whether **any** event in the period qualifies. Sentinel dates (`'1970-01-01'` , `'2099-12-31'`) are substituted by Python at query time.

JOIN-Based OU Resolution

`ANY(ARRAY[…UIDs])` → `JOIN organisationunit ON level=N`

Large `ARRAY` parameters for LEVEL and DESCENDANTS OU types exhausted PostgreSQL's shared lock table under parallel load. Replacing with a `JOIN` eliminates the lock overhead entirely.

TECHNOLOGY STACK

BACKEND	FRONTEND
Python 3 / FastAPI asyncpg, aiosqlite, httpx, PyJWT, bcrypt, Fernet	**React 18 + Vite** DHIS2 UI components, served from same FastAPI process
SQL GENERATOR	**DATABASES**
Java / ANTLR4 DHIS2's own expression parser — no reimplementation. Invoked as stdin→stdout subprocess.	**PostgreSQL + SQLite** PostgreSQL: DHIS2 DB (read-only). SQLite: SQL cache, credentials, OU/COC/DE caches.
AUTH	**DEPLOYMENT**
JWT + bcrypt App-level auth independent of DHIS2. 12-hour token TTL. Passwords AES-128-GCM encrypted at rest.	**bash + systemd** Single-command install script. Systemd service for auto-start. nginx reverse-proxy template included.

KNOWN LIMITATIONS

Analytics tables must be up to date: IndicatorX queries the pre-aggregated analytics tables; results reflect the last DHIS2 analytics export run, not live event data.

PI expression coverage: Complex expressions using functions not covered by the Java JAR's ANTLR4 grammar fall back gracefully to the SQL View mode but may require manual review.

Read-only by design: The PostgreSQL connection is read-only. All write operations (export, benchmark) go through DHIS2's REST API, not directly to the DB.

Single-server deployment: No horizontal scaling or distributed mode — designed for a single application server co-located with or proxying the DHIS2 instance.

Tracker programs only: Targets `analytics_enrollment_*` and `analytics_event_*` tables. Aggregate data values and event (non-tracker) analytics are out of scope.

FUTURE WORK

Scheduled exports: Cron-driven automatic pushes of PI results to DHIS2 aggregate data elements on a configurable schedule.

Expanded expression coverage: Extend the Java JAR grammar to handle a wider set of DHIS2 PI expression functions currently handled only by the fallback path.

Multi-instance support: Manage connections to multiple DHIS2 instances from a single IndicatorX deployment — useful for federated or multi-country setups.

Visualisation layer: Built-in charting for time-series PI results and OU-level comparison, removing the need to export to an external BI tool for quick analysis.

Raja Zubair Asghar · rajazubair.asghar@gmail.com

Source code
QR code