The Metrix Filter Language, Compared

WHITEPAPER

A technical whitepaper on query languages for engineering data.

T4Test · November 2026

The Metrix Filter Language, Compared

A technical whitepaper on query languages for engineering data

Published by T4Test Engineering Consulting · November 2026 **For:** tool-chain architects, ALM owners, functional-safety and quality leads **Companion to:** PROmeter Metrix v1.0

Abstract

Engineering teams work across many tools. Requirements live in one tool. Tests live in another. Defects and code live in two more. Every tool has its own query language.

Jira has JQL. Azure DevOps has WIQL. Polarion has Lucene. codebeamer has cbQL. Windchill RV&S has its own filter grammar. DOORS Classic has DXL. DOORS Next, ELM and ClearQuest speak OSLC. GitHub, GitLab and Jama expose parameter bags. Underneath them all sit SQL, openCypher and Gremlin.

So why another language? Because none of the languages above can answer the questions a cross-tool team actually needs. Each language leaves out something different. Together, four gaps show up again and again.

Gap 1: Multi-hop traceability. A real question is often a chain. For example: "*which customer requirements are tested?*" really means *customer requirement → system requirement → test spec → test result*. That is four hops. No ALM filter language in common use can write it as one expression.

- JQL's `linkedIssues()` needs a fixed issue key. It cannot walk.
- Polarion Lucene cannot join at all. Siemens' own help page tells users to switch to `SQL:(…)` for this kind of question.
- OSLC has a one-hop nested filter and no `or`.
- RV&S filters only an item's own fields.

Even one hop is already out of reach for most of the list. Four hops is out of reach for all of it.

Gap 2: Missing logical operators. Even simple boolean parts are not available everywhere.

- OSLC drops `or` on purpose.
- GitHub, GitLab, Jama and Notion are filter APIs. They join their slots with `and` only. There is no spelling for `or` or `not`.
- A pure `NOT` query in Lucene matches **zero**. It gives no error. This is the most dangerous case in this paper.
- WIQL ties `not` to the operator, not to the expression. So `not (a and b)` has no form.

The problem is not that the languages are weak. The problem is that each one leaves out a different piece. A team working across three tools has to learn where each one stops before it can trust a number.

Gap 3: Dropping to SQL does not fix it. Exporting ALM data into a data warehouse is not a shortcut. The multi-hop question above becomes a long cascade of `LEFT JOIN`s, anti-join idioms, recursive CTEs for depth, and `NULL`-safe comparisons for every missing field. On top of that, the author must add a `GROUP BY` and then read the result back out. SQL can say everything. It just cannot say it the way an engineer thinks it.

SQL also makes it hard to ask *why*. A composite condition is satisfied, or it is not — the engineer then wants to know which part is to blame. In Metrix this is easy and intuitive. The DSL is a logical structure, so you take the same expression and invert one atomic element at a time: swap `DecomposesTo links 'Sys Req' { … }` for `not DecomposesTo links 'Sys Req'` to find the ones with no decomposition at all; swap `'Verdict' == "Passed"` for `unspecified('Verdict')` to find the ones where the test ran but the verdict is blank. Each variant is one small edit. In SQL the same probes mean rewriting a cascade of joins, flipping `IS NULL` across several tables, and keeping `NULL` semantics straight each time. The question is the same. The effort is not.

Gap 4: Metrix fills these gaps. The Metrix DSL writes the four-hop question above as one line:

```
// applied to the 'Customer Req' scope
DecomposesTo links 'Sys Req' { VerifiedBy links 'Sys Test Spec' { 'Verdict' == "Passed" } }
```

One expression. Three hops. Each hop is a small predicate on the far end of a link. No joins. No `NULL` handling. No subquery. To ask the inverse — *which customer requirements are **not** tested?* — put one `not` in front of the same expression.

The subject — the customer requirement itself — is not in the expression. It comes from the **scope** the expression is applied to. The scope is a second key idea in Metrix. A scope names the universe: "*customer requirements in the Fabrikam baseline*", "*system tests in sprint 42*", "*Markdown files under docs/safety/*". It is declared once and reused. Change the scope and the same filter runs over a different source — Jira, Polarion or RV&S — with no edits.

In short, Metrix fills the four gaps:

1. Multi-hop traceability as one expression.
2. A full set of logical operators that compose the same way everywhere.
3. A filter that reads like an engineering question, not like a database join.
4. A scope construct that makes the same question work across every tool the team uses.

This paper is the full comparison behind that claim. One language per chapter. One table per language. For each: what translates, what does not, and why.

How to read this document

This paper compares the **languages**, not the connectors. Each chapter answers two questions: *what can this language say that Metrix cannot?* and *what can Metrix say that this language cannot?*

How the paper is structured. It opens with *The anatomy of a Metrix query*, which explains the filter language and the four book objects around it. Next, *The Metrix language in one page* is a reference to every construct the parser accepts. It is the left column of every table that follows.

Then come **eleven per-language chapters**: SQL, openCypher, Gremlin, JQL, WIQL, Polarion Lucene, cbQL, RV&S, OSLC, DOORS Classic, and the parameter-bag family (GitHub, GitLab, Jama, Notion). Every chapter has the same shape. You can read one on its own.

Three **cross-cutting sections** follow. They cover themes that run through every chapter: negation and missing values, why `and` survives everywhere and query identity. The paper closes with *What the DSL deliberately cannot do* — the limits on purpose — a conclusion and sources.

The mapping tables use three marks:

✓	translates, and this is the spelling
✗	the language has a form, but Metrix **refuses** to use it — the reason is given
—	the language has no form at all

The ✗ column matters more than the ✓ column. Every ✗ in this paper is a place where the easy translation would quietly return the wrong set. It might be a smaller set, an empty set, a superset, or nearly the opposite set. There would be no error to catch it. A refusal costs the author a rewrite. The alternative costs them a number they will trust.

A note on the Metrix snippets. Every DSL expression in this paper is a predicate on a single item. The comment above it names the scope the expression is applied to. That scope is the subject the predicate asks about. In the mapping tables the scope is left implicit, because the row-by-row translation is the point.

The anatomy of a Metrix query

The problem is not that teams lack a query language. The problem is that they have too many, and none of them answers the whole question. Metrix takes a different line.

A Metrix filter is a short true-or-false statement about **one item**. The metric counts the items for which it is true. That is the whole job of the language. So the engineer writes one line:

```
 // applied to the 'Customer Req' scope
DecomposesTo links 'Sys Req' { VerifiedBy links 'Sys Test Spec' { 'Verdict' == "Passed" } }
```

It reads: *a customer requirement decomposed to a system requirement verified by a passed system test*. The tool returns a number, a breakdown and a drill-down. Whether the requirements live in Jira, Polarion or RV&S is not in the sentence.

Notice what is also not in the sentence: the customer requirement itself. The subject of the predicate comes from the **scope** the expression is applied to, not from the text. The engineer attaches the filter to the `'Customer Req'` scope. The tool then evaluates it once per item in that scope. Point the same filter at a different scope — `'Safety Req'`, or a scope defined by a Jira JQL `project = ADAS AND type = Requirement` — and the question moves with it. That is the first piece of leverage the scope construct gives.

The second piece of leverage is that scopes are also where the hops land. `'Sys Req'` and `'Sys Test Spec'` above are scope names. They stand for "*system requirements in the current baseline*" and "*system test specifications in the Fabrikam project*". They are declared once in the book, named, versioned and reused. They are not written inside every expression that reaches them. Rebaseline, and every metric that walks through `'Sys Req'` picks up the new universe. No text change.

The design works because Metrix pulls four decisions out of the expression. Each one is a separate object in the **book**:

- the **scope** names the universe — both the one the predicate is asked of (the subject) and the ones the hops land in (above: `'Customer Req'`, `'Sys Req'`, `'Sys Test Spec'`);
- the **metric** says what is counted (the filter itself, plus whether true or false is the number);
- the **evaluator** says how the count is broken down (by status, by component, by owner);
- the **report** says how it is drawn (table, KPI tile, trend line).

Each object is authored once by the person responsible for it. Each is reused everywhere. The filter language is left with one job: say whether one item passes or not. That is why it has no `FROM`, no `SELECT`, no `GROUP BY` and no `ORDER BY`. The other three objects own those jobs.

By **shape**, the Metrix DSL sits next to JQL, Polarion Lucene and the RV&S query definition — the selection languages that filter one list of items. By **reach**, it sits with SQL, Cypher and Gremlin. It can walk a link, filter on the far end, count across a reachable set, and chain any number of hops. It gets both because a Metrix run collects every source first into a single graph. The expression is translated against that graph, not against the source tool.

That single design choice is the whole paper in miniature. The rest is spelling out what it buys.

The Metrix language in one page

Each line below is a construct the parser accepts. This is the left column of every table in the rest of the paper.

```
 and / or / not                          boolean operators, two-valued
==  !=  >  >=  <  <=                     comparison; either side may be an attribute
contains / startsWith / endsWith         case-sensitive text match
contains* / startsWith* / endsWith*      case-insensitive forms
not contains / not startsWith / …        negative forms
matchesRegex / not matchesRegex          pattern match
specified(x) / unspecified(x)            total predicates about absence
== anyOf("a", "b")                       some listed value matches
== allOf("a", "b")                       every element is one of these
== atLeast(2).of(1, 2, 3)                at least two of these are matched
== atMost(0).of(1, 2, 3)                 non-membership
("a", "b") / ["a", "b"] / 1..10          bare arrays; a range is an array
effort + 1 > other * 2                   arithmetic: + - * /
count(x) sum(x) mean(x) min(x) max(x)    aggregates over a multi-valued attribute
length(x)                                as an operand, never as output
Rel links Scope                          at least one link landing in Scope
Rel linksAtLeast(2) Scope                at least two
Rel linksAtMost(0) Scope                 none — the complement
Rel links* Scope                         …and no links of any other kind
Rel links Scope [1..3] / [1..*] / (0..3]  transitive, bounded hops
Rel links Scope { priority > 5 }         correlated predicate on the reached item
count(Rel.attr [1..2] { … }) > 0         aggregate over a filtered reachable set
status.name == "Done"                    a record's part
Assignee.email contains* "@t4test"       a relation's sub-attribute
'test result'.verdict == "Passed"        an edge attribute — the relation's own column
$threshold                               a bound book parameter
"2026-01-01" / 2026 / 2026-03            date, date-time, year, year-month literals
```

Two conventions trip people up. Both are on purpose:

- `'…'` **is an identifier.** `"…"` **is a textual literal.** Always, everywhere. An earlier shortcut guessed the kind from where the operand sat. It was removed. It turned `Level == "SafetyFunction"` into a column-vs-column comparison on any book whose vocabulary happened to include that word.
- **A bare comparison on a multi-valued attribute means *some element*.** "Every element" is never implicit. Use `allOf` for that. So `m > 9` is true when at least one value of `m` exceeds 9.

Per-language comparisons

The eleven chapters that follow do two things for the reader at once.

First, they show coverage. For each language, the chapter lists which of its features Metrix covers, which it does not, and — in the other direction — which Metrix constructs the language has no spelling for. The reader sees, at a glance, the overlap and the two sets of gaps on either side.

Second, they are a bridge from a language the reader already knows. Most engineers come to Metrix already fluent in one or two of these neighbours. A Jira power user already thinks in JQL. A codebeamer admin already thinks in cbQL. A data engineer already thinks in SQL. Each chapter is written so that reader can understand the Metrix concept from their familiar starting point: "*the thing I write as* `field in (a, b)`

in JQL, I write as `== anyOf("a", "b")` in Metrix; the thing I cannot write at all in JQL — a hop across a link — Metrix writes as `Rel Links Scope { … }`."

The chapters cover every language a cross-tool team is likely to meet:

- The three general-purpose languages under everything: SQL, openCypher, Gremlin.
- The five ALM-native languages: JQL, WIQL, Polarion Lucene, cbQL, RV&S.
- The industry standard that crosses several ALM tools: OSLC.
- The scripting language DOORS Classic users mean by "the language": DXL.
- The parameter-bag filter APIs of lightweight trackers: GitHub, GitLab, Jama, Notion.

Each chapter has the same five parts. A reader who knows the language can read just that one chapter and skip the rest.

- **What it is** — a short characterisation of the language, with a native-syntax example so the mapping table has a place to anchor. For a reader fluent in the language this is the hook: it shows the comparison starts from code they recognise.
- **Mapping table** — row by row against the Metrix constructs from the one-page reference. The three marks (✓, ✗, —) are defined in *How to read this document*. This is the coverage picture in one place.
- **What the language has that Metrix does not** — an honest list. This is where the paper stops claiming for Metrix.
- **What Metrix has that the language does not** — the mirror list. Usually shorter, but always important. For a reader coming from the language, this is the list that is worth learning new syntax for.
- **Takeaway** — one sentence that stays with the reader if nothing else does.

For the five ALM filter languages (JQL, WIQL, Polarion Lucene, cbQL, RV&S) the mapping tables are read off the production Metrix translators. The spellings are what Metrix actually sends. The refusals are quoted from the code. For the others (SQL, openCypher, Gremlin, OSLC, DOORS Classic, the parameter bags) the rows come from the language specifications cited in *Sources*.

SQL

What it is. The baseline. A SQL statement is a relation-valued expression. It names its own universe, its own columns, its own grouping and its own order. It can also write.

```
 SELECT r.id, count(t.id) AS tests
FROM requirement r
LEFT JOIN verifies v ON v.req_id = r.id
LEFT JOIN test t ON t.id = v.test_id
WHERE r.status = 'Approved'
GROUP BY r.id
HAVING count(t.id) >= 2
ORDER BY tests DESC;
```

Mapping. Every Metrix construct has a SQL spelling. The interesting column is how much machinery each needs.

Metrix	SQL
`and` / `or` / `not`	✓ `AND` / `OR` / `NOT` — but three-valued; see below
comparisons, arithmetic	✓ identical
attribute vs attribute	✓ `a.x > a.y`
`contains` / `startsWith`	✓ `LIKE '%x%'` / `LIKE 'x%'`; `matchesRegex` depends on the dialect (`~`, `REGEXP_LIKE`)
`specified(x)`	✓ `x IS NOT NULL`
`== anyOf(…)`	✓ `x IN (…)`
`== allOf(…)`	— for a scalar column; needs an array type or a junction table
`== atLeast(2).of(…)`	— as one construct; a correlated `count(*) >= 2` subquery
`Rel links Scope`	✓ `EXISTS (SELECT 1 FROM … )`
`Rel linksAtLeast(2) Scope`	✓ `(SELECT count(*) …) >= 2`
`[1..3]` depth	✓ recursive CTE with a depth column
`{ … }` filter block	✓ the subquery's own `WHERE`
`count(Rel.attr { … })`	✓ correlated aggregate subquery

What SQL has that Metrix does not. Almost everything SQL does besides the `WHERE` clause. A Metrix filter is a condition, not a statement. It answers *does this item count?* and leaves the rest to the next tool. The purpose of a metric is to produce a number and the set of items behind it, ready for a downstream receiver — a report, a trend sidecar, an Excel export, a BI dashboard, a human reviewer — to process further. Picking columns and sorting rows are the receiver's job, not the filter's. So the DSL deliberately leaves those out:

- **Picking columns (`SELECT`)** — a report decides what is shown. A JSON export carries every attribute. An Excel export flattens a chosen set. The filter does not need to list them.
- **Grouping and aggregating as output (`GROUP BY`, `HAVING`)** — an *evaluator* declares the buckets up front, by name. A chart then draws them. Grouping is a modelling decision, not a per-query decision.
- **Sorting, limiting, deduplicating (`ORDER BY`, `LIMIT`, `DISTINCT`)** — the report and the drill-down grid own these. A metric is a count, and a count has no natural order.
- **Window functions, set operations, `CASE`, casts, string functions** — these shape the output record. The downstream tool (Excel, Power BI, a notebook) has them, and does them better.
- **Joins between tables that have no relationship** — a Metrix walk crosses a *link* that exists in the graph. An ad-hoc cross-join is not a cross-tool engineering question. It is a data-exploration question. The right tool for it is SQL, on an export.
- **Subqueries that return values, and writes** — a definition is read-only by construction, so it can be hashed, versioned and reviewed.

If a question needs any of the machinery above, it is not a metric. It is an analysis. The right tool is a drill-down export or direct access to the source.

What Metrix has that SQL does not.

- **Two-valued missing semantics.** `NULL` makes `NOT (x = 5)` *unknown* in SQL. So a predicate and its negation do **not** partition the table. In Metrix they do. This is the single most important difference on this page. It changes numbers, not convenience.
- **Multi-valued attributes as first-class values.** With a clear quantifier ladder. In SQL, a multi-valued field is either an array type or a second table. The quantifier is spelled differently in each case.
- **A traversal that cannot multiply rows.** A SQL join can turn a count of items into a count of links — a bigger number. Bigger numbers are the direction nobody questions. In Metrix `links` is a quantifier, not a join. So that failure cannot happen.
- **Portability.** `State == "Reviewed"` is a sentence about an item, not about a table. The same metric runs over a Jira scope and an RV&S scope.
- **Query identity.** A canonical form and a fingerprint. See the cross-cutting section.

Takeaway. What Metrix gives up against SQL is the output-shaping half: there is no `SELECT`, no `GROUP BY` as output, no `ORDER BY`, no window functions, no writes. The condition is all there is. The report, the evaluator and the downstream export carry the rest.

---

openCypher

What it is. The compile target. Metrix translates the DSL into openCypher. ArcadeDB 26.x answers it natively. The mapping below is not an analogy. It is what the translator emits.

```
MATCH (n:Item) WHERE n.`State` = 'Approved'
  AND EXISTS { MATCH (n)-[:`Rel_VerifiedByTest`]->(m) WHERE 'TST' IN m.`_scopes` }
RETURN n
```

Metrix	What the Cypher translator emits
`Rel links Scope`	`EXISTS { … }`
`Rel linksAtLeast(n) Scope`	`COUNT { … } >= n`
`Rel linksAtMost(n) Scope`	`COUNT { … } <= n`
`Rel links* Scope` (exclusive)	`(<quantified> AND COUNT { <every link> } = COUNT { <inside> })`
`[lo..hi]` depth	`*lo..hi` on the relationship pattern
`specified(x)`	`x IS NOT NULL`
multi-valued bare comparison	`any(e IN n.m WHERE …)` — never a bare `n.m > 9`, which is a type mismatch
`allOf(…)`	`(size(n.m) > 0 AND all(e IN n.m WHERE …))`
`atLeast(k).of(…)`	`size([e IN n.m WHERE …]) >= k`
`sum(Rel.attr)`	`COLLECT { MATCH … RETURN col }` folded with `reduce`
two-valued leaf	an `IS NOT NULL` guard per operand, inside a negation

What openCypher has that Metrix does not. `RETURN` and projection. Paths as values, including `shortestPath`. Aggregation as output. `WITH` pipelines. `OPTIONAL MATCH`. `ORDER BY` / `SKIP` / `LIMIT`. `CREATE` / `MERGE` / `SET` — it can write.

What Metrix has that Cypher-on-this-engine does not. This is the interesting direction here. Each item is a measured engine behaviour the translator exists to route around.

- **Two-valued semantics.** Cypher's `null` is SQL's. The collapse happens at each **leaf**, not at the top. On this engine, neither `coalesce` nor `CASE WHEN` collapses a comparison that contains a list `IN`.
- **A list literal inside a projected subquery answers `true` for every row.** Measured on ArcadeDB 26.x. `EXISTS { … WHERE m.x IN ['Nope'] }` as a *projected column* is true for everything. The same subquery in a `WHERE` is correct. A one-statement-per-evaluator form was built, measured and withdrawn. It reported every item in every bucket with no error anywhere.
- **Parameters do not survive into a subquery.** They are not rejected. The subquery just matches nothing. So everything inside a walk is rendered as a literal. Dates use `date(…)` and `datetime(…)` constructors. Text is escaped.
- **Pattern comprehensions return an empty list.** `[(n)-[:X]->(m) | m.attr]` parses and answers nothing. Every aggregate built on one would silently answer zero. Aggregates use `COLLECT { … }` and `reduce` instead.

Takeaway. What Metrix gives up against openCypher is the output half of the graph language: no `RETURN`, no projection, no paths as values, no `OPTIONAL MATCH`, no writes. The Metrix expression is only the pattern filter — the shaping of the answer is done elsewhere.

Gremlin

What it is. An imperative traversal language inside a host language (Groovy, Java, Python). A query is a chain of steps. With lambdas, it is Turing complete.

```
g.V().hasLabel('requirement').has('status', 'Approved')
     .not(__.out('verifiedBy').hasLabel('test'))
     .count()
```

Metrix	Gremlin
`Rel links Scope`	✓ `where(__.out('rel').hasLabel('scope'))`
`Rel linksAtLeast(2) Scope`	✓ `where(__.out('rel').count().is(gte(2)))`
`[1..3]` depth	✓ `repeat(__.out('rel')).times(3).emit()`
`{ … }` filter block	✓ the anonymous traversal's own filters
`allOf` / `atLeast(k).of`	✓ expressible by counting inside a traversal
everything else	✓ — Gremlin is strictly more expressive

What Gremlin has that Metrix does not. `path()`. Full traversal control (`sack`, `barrier`, `by()` modulators). OLAP execution over a whole graph. Branching and loops with arbitrary exit conditions. Lambdas, which means anything at all. It can also write.

What Metrix has that Gremlin does not. Nothing about power. The gap is **identity and reviewability**.

- A Gremlin traversal is a program in a host language. It has no canonical form. It cannot be fingerprinted. "*Did this question change?*" has no machine answer.
- It cannot be round-tripped through an editor form. That is how most Metrix expressions are authored and shown.
- It cannot be shown to an auditor as the definition of a number. Reading it needs executing it in your head.

Takeaway. What Metrix gives up against Gremlin is unrestricted programming: no lambdas, no host-language control flow, no `path()`, no OLAP traversal, no writes. Gremlin is strictly more expressive; Metrix trades that expressiveness for a definition that can be hashed, reviewed and shown to an auditor.

JQL (Jira)

What it is. A selection language. In shape it is the closest neighbour to the DSL. It selects issues and orders them. That is all it does. Its text operator `~` is backed by a Lucene index.

```
project = METRIX AND status in ("In Review", "Approved")
  AND assignee is not EMPTY AND summary ~ "safety*"
ORDER BY created DESC
```

Metrix	JQL
`and` / `or` / `not`	✓ `AND` / `OR` / `NOT (…)`, every operand bracketed
`==` `!=` `>` `>=` `<` `<=`	✓ `=` `!=` `>` `>=` `<` `<=`
`== anyOf(…)`	✓ `field in (a, b)`
`== atMost(0).of(…)`	✓ `field not in (a, b)`
ordering against a list	*✗ JQL compares a list only with `in` / `not in`*
`contains*` / `not contains*`	✓ `field ~ "x"` / `field !~ "x"`
`startsWith*`	✓ `field ~ "x*"` — a **word** prefix: `~ "in*"` matches *in progress*, not *spin*
`contains` / `startsWith` (case-sensitive)	*✗ JQL text matching is case-insensitive, so this would quietly match more than it says*
`endsWith`	*✗ JQL has no suffix match*
`matchesRegex`	*✗ JQL has no pattern match*
`specified` / `unspecified`	✓ `field is not EMPTY` / `field is EMPTY`
`status.name == "Done"`	✓ `status = "Done"` — but only for `status`, `statusCategory`, `priority`, whose published meaning *is* comparison by name
any other record part	✗ names the part, and says which three to use instead
attribute vs attribute	*— the left side must be an attribute of the item being selected*; **JQL cannot compare two fields at all**
arithmetic	—
aggregates	—
relation walk	*— a hop between issues has no equivalent on a stock Jira*
dates	✓ `"yyyy-MM-dd"`, date-times as `"yyyy-MM-dd HH:mm"` — a space, not the ISO `T`, which JQL rejects; seconds dropped
year / year-month literals	✗ a year is a *range*; JQL has no literal for one, so no boundary is invented

What JQL has that Metrix does not.

- **History operators.** This is a real architectural difference: `WAS`, `WAS IN`, `WAS NOT`, `WAS NOT IN` and `CHANGED`, narrowed with `BY`, `BEFORE`, `AFTER` and `DURING`. `status WAS "Blocked" DURING ("2026-01-01", "2026-02-01")` has no Metrix spelling. Metrix answers history from the trend sidecar and from the scope's temporal marker, never from the expression. The limitation on Jira's side is that these operators only work for six fields (assignee, fix version, priority, reporter, resolution, status).
- **Functions.** `currentUser()`, `membersOf()`, `openSprints()`, `startOfDay()`, `endOfMonth()`. The DSL has no function library at all. The equivalent of `currentUser()` is a bound parameter.
- `ORDER BY` and the text index behind `~` — stemming and relevance. The DSL's character-level operators do not have those.

What Metrix has that JQL does not. The quantified link hop. (`linkedIssues()` takes an issue **key**, so "*issues linked to anything whose status is Done*" has no native spelling. Functions like `linkedIssuesOf(...)` are ScriptRunner's, not Jira's.) Any comparison whose right side is not a constant. Arithmetic. Aggregates. Suffix and regex matching. Case-sensitive text matching. Counting quantifiers.

Takeaway. What Metrix gives up against JQL is history and the function library: no `WAS`, no `CHANGED`, no `currentUser()`, no `@Today`, no `ORDER BY`, and no text ranking. History answers come from the trend sidecar and the scope's temporal marker; "current user" is a bound book parameter; ranking is not a job for a metric.

WIQL (Azure DevOps)

What it is. The most SQL-shaped ALM language here. The only one with a temporal clause and a link-query mode built in. Its projection is nominal — the REST API returns ids.

```
 SELECT [System.Id] FROM WorkItems
WHERE [System.TeamProject] = @Project
  AND [System.WorkItemType] = 'Requirement'
  AND [System.AreaPath] Under 'Fabrikam\Safety'
  AND [Microsoft.VSTS.Common.ClosedDate] >= [System.CreatedDate]
ORDER BY [System.CreatedDate] ASOF '2026-03-01'
```

Metrix	WIQL
`and` / `or`	✓ `AND` / `OR`, bracketed
`not(…)`	✗ *WIQL negates operators —* `<>`, `Not In`, `Does Not Contain` *— and a* `NOT (…)` *grouping has not been established against a server*
`==` `!=` orderings	✓ `=` `<>` `<` `<=` `>` `>=`
`== anyOf(…)`	✓ `field In (a, b)`
`== atMost(0).of(…)`	✓ `field Not In (a, b)`
ordering against a list	✗ *WIQL compares a list only with* `In` *and* `Not In`
`contains*` / `not contains*`	✓ `field Contains "x"` / `field Does Not Contain "x"`
`contains` (case-sensitive)	✗ `Contains` is case-insensitive; admitting the sensitive twin would silently return a superset
`startsWith` / `endsWith`	✗ *WIQL has no prefix match —* `Contains` is unanchored; `Contains Words` is a full-text index match, a different question
`matchesRegex`	✗ *WIQL has no pattern match*
`specified` / `unspecified`	✓ `field Is Not Empty` / `field Is Empty` — admitted on a **published spelling**, which Azure documents for long-text fields
a field name containing `]`	✗ *WIQL publishes no escape for a* `]` *inside a field reference*
relation walk	— not as a predicate; see below

What WIQL has that Metrix does not. More than any other comparator here.

- **`ASOF '<date>'`** — a temporal clause *in the language*. Metrix's equivalent is the scope's temporal marker. That is better for reuse. It is worse for an ad-hoc *"what did this look like in March?"* — that is a second scope.
- **`Was Ever`** — a historical value operator. It cannot be negated. The Query Editor does not offer it on date fields.
- **`Under`** — a tree-path prefix operator over area and iteration paths. Metrix has no path-hierarchy operator. The nearest thing is a transitive walk over a relation. That is a different data shape.
- **`In Group`** — membership of a named security group, team or category, resolved server-side.
- **Macros** — `@Me`, `@Today±x`, `@CurrentIteration`, `@Project`.
- **Field-to-field comparison** with a bracketed right side (`>=[System.CreatedDate]`). JQL cannot do this. Metrix can.

What Metrix has that WIQL does not. All three differences are about **composition**, not reach.

- **Negation of an expression.** WIQL's negation is bound to a comparison. So `not(a and b)` has no WIQL form at all.

- **A link predicate that composes with the rest of the filter.** WIQL *can* ask a one-hop link question. But only by changing `FROM WorkItems` to `FROM WorkItemLinks`. That makes the whole query about link pairs. In Metrix, `links` is a term inside an ordinary predicate. It `and`s with everything else.
- **Bounded transitive depth.** `mode(recursive)` is unbounded. It needs a forward tree-topology link type. Per Microsoft's own reference, it does not support `ORDER BY` or `ASOF`. That is exactly the pairing you most want. `[1..3]` has no such restrictions.

Takeaway. What Metrix gives up against WIQL is the in-language temporal and hierarchy machinery: no `ASOF`, no `Was Ever`, no `Under` for area and iteration paths, no `In Group`, no `@Me` / `@Today±x` macros. Metrix moves these to the scope and to bound parameters, so they cost a scope change rather than a per-query clause.

Polarion (Lucene, and the `SQL:` escape)

What it is. Polarion's work item query language is Apache Lucene's, with Polarion fields and a documented ceiling of 2048 clauses. Next to it sits a virtual field `SQL:(…)` that runs SQL against Polarion's embedded database. That is the vendor's own escape hatch for questions Lucene cannot ask.

```
type:requirement AND status:approved AND HAS_VALUE:assignee
    AND created:[20260101 TO *]
```

A construct is admitted below only where Siemens' published documentation gives the spelling **and** a wrong guess would be *rejected*, not answered.

Metrix	Polarion Lucene
`and` / `or`	✓ `AND` / `OR`
`not(…)`	✗ **the whole reason this is narrow** — see below
`==`	✓ `field:value`
`>=` / `<=`	✓ `field:[value TO *]` / `field:[* TO value]`, the documented open-ended range
`>` / `<`	✗ an exclusive bound uses braces; *a server that treats `{` as `[` would silently include the boundary item*
`!=`	✗ *this becomes a negation with nothing to subtract from, which matches no items rather than failing*
`== anyOf(…)`	✓ `field:a OR field:b`
`!=` or ordering against a list	✗ *Lucene expresses a list only as a disjunction, which carries `==` and nothing else*
every text operator	✗ all of them — Lucene's analyzer decides tokenisation and case, so a character-level operator is a different question
`specified(x)`	✓ `HAS_VALUE:field`
`unspecified(x)`	✗ `NOT HAS_VALUE:field` is what Polarion's own query builder emits and is still a top-level negation
relation walk	—

The negation ledger. This is the most dangerous row in the paper, so state it once in full: **a purely negative Lucene query matches nothing, not an error.** `NOT status:open` has no positive clause to subtract from. Classic Lucene answers it with an empty result set — no error, no warning, a plausible zero, in a metric that reports 0 and looks like good news. Polarion's query builder emits top-level negations, so its parser very likely wraps them. *Very likely* is not the bar.

What Polarion has that Metrix does not. Everything Lucene is actually for. Wildcards (`title:req*`). Fuzzy matching (`term~`). Proximity. Term boosting (`^2`). Relevance ordering. Free text across many fields at once. The DSL has `contains` / `startsWith` / `endsWith` / `matchesRegex` with case-insensitive twins, and **no ranking at all**. A metric is a count, and a count has no use for relevance. And `SQL:(…)` gives real joins, grouping and aggregation against Polarion's own schema.

What Metrix has that Polarion does not. Every row refused above, plus the quantified hop without dropping to SQL. That is the comparison that matters. Dropping to `SQL:` means writing against Polarion's physical tables. That is a different skill and a different maintenance story from writing a filter.

Takeaway. What Metrix gives up against Polarion is everything Lucene is actually for: wildcards, fuzzy matching, proximity, term boosting, relevance ordering, free text across many fields at once. A metric is a count, and a count has no use for ranking — but if the question really is "find the most relevant document", that is a search, not a metric.

---

cbQL (codebeamer)

What it is. A real query language with a parser — `SELECT`, `WHERE`, `GROUP BY`, `ORDER BY`. The most SQL-like of the ALM selection languages. Fields carry their own supported operator lists.

```
 SELECT workItemId, status
WHERE tracker.id IN (1024, 1025) AND status != 'Closed'
  AND summary LIKE '%safety%' AND storyPoints BETWEEN 3 AND 8
GROUP BY status ORDER BY priority
```

cbQL has a parser, so a construct it does not know is a **syntax error returned by the server**. That means a wrong guess lands where the author can see it.

Metrix	cbQL
`and` / `or`	✓ `AND` / `OR`, bracketed unconditionally
`not(…)`	✗ cbQL negates a *field* with `!=` / `NOT IN`; whether it negates a **group** is not established
`==` `!=` orderings	✓ `=` `!=` `<` `<=` `>` `>=`
`== anyOf(…)`	✓ `field IN (a, b)`
`== atMost(0).of(…)`	✓ `field NOT IN (a, b)`
ordering against a list	✗ the DSL's modifiers say *how many* must match; cbQL has no counting form
text operators	✗ `LIKE` exists; its wildcard and case semantics are not established, and getting them wrong returns a superset
`specified` / `unspecified`	✗ not established
relation walk	—

What cbQL has that Metrix does not. Projection, `GROUP BY`, `ORDER BY`, `LIKE` / `NOT LIKE`, `BETWEEN`, arithmetic operators. Also, something distinctive: **subqueries that reference saved filters, reports or item descriptions**. For an organisation whose query library *is* its process, that last one is real leverage the DSL does not offer. No Metrix expression can name another metric, scope or saved query.

What Metrix has that cbQL does not. Expression-level negation (unestablished there). Element quantifiers over multi-valued fields. Counting modifiers. The hop. Aggregates over a reachable set.

Takeaway. What Metrix gives up against cbQL is the output machinery (projection, `GROUP BY`, `ORDER BY`, `LIKE`, `BETWEEN`, arithmetic operators) and — the one that stings — **subqueries that reference saved filters, reports or item descriptions**. No Metrix expression can name another metric, scope or saved query. For organisations whose query library *is* their process, that is a real limit.

The Windchill RV&S query definition

What it is. A parenthesised filter rule, passed to `im issues --queryDefinition=` or stored by `im createquery`. Fields are written `field["Name"]`. Groups combine with `and` / `or`. A value is a literal, a range, or the `is empty` keyword. PTC publishes no grammar. The public record is worked examples. That sets this translator's bar.

```
((field["Type"] = "Requirement") and (field["State"] in ("Open","In Review")))
```

Metrix	RV&S
`and` / `or`	✓ `and` / `or`
`not(…)`	✓ `not (…)` — **the one construct admitted on measured evidence**: a working product asks a real server `(… and (not (field["State"] = "Cancelled")))`
`==` `!=`	✓ `field["X"] = "v"` / `!=`; text quoted, numbers bare
orderings	✓ numbers and dates only — what a server does when asked to order *text* is not established, and a lexicographic answer and a rejection are both plausible
`3 < x`	✓ **mirrored** to `x > 3`, which was invisible while `=` was the only operator admitted
`== anyOf(…)`	✓ `field["X"] in ("a","b")` — documented rather than measured, admitted because the failure would be *loud*
`== atMost(0).of(…)`	✓ `not in (…)` — this is how the DSL spells non-membership
`!= anyOf(…)`	✗ it means *differs from **some** listed value*, true of an item in `Open` against `anyOf("Open","Done")` — **not** non-membership
`!= allOf(…)`	✓ on a single-valued field; ✗ on a multi-valued one, where it reads *every element differs from some literal*
`== value` on a multi-valued field	✓ RV&S's `contains` — membership, a different operator from the DSL's substring `contains`
`== allOf(…)`	✗ the DSL asks that every *element* is one of the listed — a subset test — while RV&S's `contains all` asks the **opposite containment**
text operators	✗ `contains` exists and is recommended for text fields, but nothing establishes its case sensitivity; mapping a case-sensitive operator onto a case-insensitive one returns a superset in silence
`specified` / `unspecified`	✗ *RV&S has `isEmpty` in computed expressions; whether a query filter accepts it is a different question, pending measurement*
boolean literals	✗ a logical field compared against `true` may match on the text and may match on nothing
year / year-month	✗ each names a *range* where RV&S compares against an instant
text containing a quote	✗ the escape is unknown
relation walk	— a search runs server-side over an item's own fields

What RV&S has that Metrix does not.

- **Subquery filters by saved-query name** — the same leverage cbQL has. The same thing the DSL does not offer.
- **`item.live` / `item.versioned`** — version-aware selection inside the filter. With document versioning on, a query returns both by default. This narrows it. Metrix's equivalent lives on the scope's temporal marker instead.
- **`is empty`**, which is part of the value grammar in PTC's query-filter documentation.

- **Text search across all text fields.** It matches all words of a string by default.

What Metrix has that RV&S does not. Every refused row above. The hop. Aggregates. Arithmetic. One more property is worth naming: **a Metrix temporal marker is resolved against the server before anything is read**. `BASELINE` and `REVISION` name a label. RV&S says of every label whether it marks a version or a moment. So a marker of the wrong kind is refused, not silently resolved to *now*.

Takeaway. What Metrix gives up against RV&S is subquery filters by saved-query name, `item.live` / `item.versioned` selection inside the filter, `is empty` as a value, and the default all-words text search. The first three cost structural leverage; the fourth is a search convenience a metric does not need.

OSLC `oslc.where` — DOORS Next, ELM, ClearQuest

What it is. The ALM industry's standard query syntax. It is carried as URL parameters. It is the only language here designed to be implemented by many vendors. It is also the only one that leaves out a boolean operator on purpose.

```
 ?oslc.prefix=dcterms=<http://purl.org/dc/terms/>
&oslc.where=dcterms:modified>"2026-04-01" and oslc_cm:status in ["Submitted","Open"]
&oslc.select=*,dcterms:creator{foaf:givenName}
&oslc.orderBy=-dcterms:modified
```

Metrix	OSLC
`and`	✓ `and`
`or`	— **absent by design**: *not allowed in the interests of keeping the syntax simple*
`not`	—
comparisons	✓ the usual binary operators
`== anyOf(…)`	✓ `prop in ["a","b"]`
`specified(x)`	✓ `prop=*` — the wildcard matches any value
`{ … }` filter block	✓ **the closest analogue in any ALM standard**: `oslc_rm:uses{dcterms:title="x"}` filters on a nested property
counting / depth / aggregates	—

What OSLC has that Metrix does not. `oslc.select` — a projection that can reach nested properties (`*,cq:customer{cq:Company}`). `oslc.searchTerms` for provider-defined full-text search. `oslc.orderBy` . And standardisation itself. That is the whole point: one syntax across DOORS Next, ELM and ClearQuest.

What Metrix has that OSLC does not. `or` . Negation of any kind. Counting quantifiers. Bounded transitive depth. Aggregates.

Takeaway. What Metrix gives up against OSLC is the projection and the standardisation: `oslc.select` can reach nested properties, `oslc.searchTerms` offers provider-defined full-text search, and one OSLC query runs across DOORS Next, ELM and ClearQuest unchanged. Metrix moves projection into the report, and reaches cross-tool portability through the scope, not through a wire protocol.

DOORS Classic: filters and DXL

What it is. Two things get called "the language", and they are not alike.

A **filter** is a rule tree over an object's attributes, built in the UI and saved in a view. One rule type is a *DXL attribute condition*. IBM's own support note records that adding one runs the DXL immediately over every object in the module.

DXL is a general-purpose imperative language with full database access:

```
for o in current Module do {
    if (o."Status" "" == "Approved") { … }
}
```

Metrix	DOORS Classic
attribute comparisons	✓ in a filter
`and` / `or` / `not`	✓ in a filter's rule tree
text matching	✓ including regular expressions in a filter
`Rel links Scope`	— a filter narrows by link **module**, not by a predicate on the target's attributes
`{ … }` on the reached item	—
counting, depth, aggregates	— in a filter; ✓ in DXL, as a loop you write

What DOORS has that Metrix does not. DXL can do anything. Write attributes. Drive the GUI. Build exports. Implement suspect-link logic. It is also how most non-trivial DOORS reporting is actually done. That is the honest comparison — not filter against DSL, but *script* against DSL.

What Metrix has that DOORS does not. A declarative, hashable, side-effect-free definition. And specifically: **a predicate on the target of a link**. "*Requirements linked to an approved test*" is not a DOORS filter. The forum record on filtering by target module confirms the shape — you filter by the link module and still open the linked formal modules. Two smaller things: a filter built from attribute text cannot be saved in a view, and a DXL-attribute rule costs a full-module evaluation whenever the filter is applied.

Takeaway. What Metrix gives up against DXL is everything an imperative language can do: writing attributes, driving the GUI, building custom exports, implementing suspect-link logic. Metrix is read-only by construction — the price of a hashable, auditable definition. For read-write workflows DXL stays the right tool.

> DOORS **Next** is a different system. It belongs to the OSLC chapter. Its query surface is `oslc.where` plus Reportable REST. That is what the Metrix connector reads.

The parameter-bag family: GitHub, GitLab, Jama, Notion

What they are. Not query languages, but **filter APIs** — a bag of named slots the server conjoins. GitHub's search has a qualifier syntax over the top (`is:open label:bug no:assignee`). GitLab, Jama and Notion take parameters or a filter object.

```
GitHub   is:issue is:open label:bug no:milestone created:>2026-01-01
GitLab   ?state=opened&labels=bug&author_username=someone
```

Metrix	A parameter bag
`and`	✓ — adding a term is adding a parameter
`or`	✗ **refused**, and this is the family's defining limit: *GitLab combines its filter parameters with* `and` *only, so a disjunction would silently mean its opposite* (measured for GitHub)
`not`	✗
`==`	✓ where a slot exists for that field
orderings	partial (GitHub's `created:>…`), none for GitLab and Jama
text operators, aggregates, walks	—

What they have that Metrix does not. Very little as languages. GitHub's `no:` qualifier is a neat spelling of `unspecified` . These APIs are also cheap and fast, because they are indexes, not parsers.

What Metrix has that they do not. Everything structural. The important point is the *failure mode*. Rendering `a or b` into an and-only parameter bag sends a **smaller** set. It is returned successfully. Nothing says the query was not the one the author wrote. Doing it properly needs a request per operand and a client-side union. That changes paging, counts and the drift story. So it is refused instead.

Takeaway. What Metrix gives up against parameter bags is speed-of-fetch and the index-level `no:field` convenience. These APIs are not query languages at all, so the cost on the Metrix side is small; the cost of using them as if they were a query language is the silent wrong-set failure mode above.

Cross-cutting: negation and missing values

If you read one section of this paper, read this one. Every language takes a different position on missing values. The position is usually invisible until a metric and its complement fail to add up.

Language	Missing value	Negation
Metrix	a comparison about an absent value is **false**	total; `!=` *is* `not (==)`; complement sums to the population
SQL	`NULL` → unknown (three-valued)	`NOT (x = 5)` is unknown for `NULL`; complement does **not** sum
openCypher	`null` → unknown, SQL-like	as SQL
JQL	asked with `IS EMPTY`	has `NOT`; an empty field satisfies neither side of a comparison
WIQL	`Is Empty` / `Is Not Empty`	bound to **operators** — `<>`, `Not In`, `Does Not Contain`; `not(a and b)` has no form
Polarion Lucene	`HAS_VALUE:field` for presence	**subtractive**: a pure negative query matches **zero**, not an error
cbQL	per field	`!=`, `NOT IN`, `NOT LIKE`; group negation undocumented
RV&S	`is empty` keyword	filters negate; `not (…)` is measured to work
OSLC	`prop=*` for presence	none at all
DOORS filter	an attribute rule	✓ in the rule tree
parameter bags	GitHub's `no:field`	none

Metrix was three-valued before issue #191. It used Kleene's tables and openCypher's `NULL` semantics. It was given up for a rule an author can state in one sentence (*a comparison about an absent value is false*). It was also given up so counts sum. The loss was accepted knowingly. A negation now sweeps every unread item into a metric. A number that only grows is one nobody questions. What buys it back is that absence stays askable — and *only* askable — through two total predicates:

```
specified(state) and state != "Done"      // read, and not Done
unspecified(state)                        // never read
```

A rule worth stating for anyone porting expressions into Metrix: **a rewrite is only safe if it is unknown-transparent.** `not (a == 5)` ⇒ `a != 5` is sound. Collapsing `a > 10 and a < 5` to `false` is not. The two differ under a surrounding negation when `a` is absent.

Cross-cutting: why `and` survives everywhere

The pattern across all the translator chapters is the same. Every connector translates `and`. Half refuse `or`. Most refuse `not`. That is not a coincidence of effort.

`and` is the one operator whose translation **cannot be silently wrong**. A filter API is a bag of slots the server conjoins. A query language's `AND` is its most basic connective. Either way, translating a conjunction

adds a term and changes nothing about the shape of the request.

Every refusal in this paper is a place where the alternative was a query that *runs and answers with the wrong set*:

- **smaller**, for a mis-rendered `or` against an and-only parameter bag;
- **empty**, for a Lucene negation with nothing to subtract from;
- **a superset**, for a case-sensitive operator mapped onto a case-insensitive one;
- **differently grouped**, for an unestablished group negation;
- **nearly the opposite set**, for `!= anyOf(…)` rendered as `not in`.

A refusal costs an author a rewrite. The alternative costs them a number they will trust. Both ways out are always named in the message: write it as a native query, or split it into separate scopes.

Cross-cutting: query identity

A Metrix expression has a **canonical form** and a `DefinitionFingerprint`. The fingerprint is a hash of the canonical form. It carries a canonicaliser version. So a change to canonicalisation answers "*cannot tell*" rather than inventing a discontinuity nobody authored. A reordered `and` is not a change. A real reformulation is.

No language in this paper has any notion of query identity. A Jira filter, a Polarion saved query, an RV&S stored query, a cbQL report and a DOORS view can each be edited in place while a chart goes on drawing the same series. Nothing anywhere records that the question moved. That is not a criticism. It is a job those languages were never asked to do. It is also the whole reason the DSL has a canonical form. A trend series is keyed on ids. The fingerprint is the only thing that can say the meaning changed while the id did not.

What the DSL deliberately cannot do

"Cannot" and "not yet" are different claims. This list is the first kind.

- **No projection, ordering, or aggregate-as-output.** Those are the report's, the grid's and the metric's jobs. Aggregates exist only on the left of a comparison. `count('has execution')

  > `= 2 is a predicate, so it is a metric.` SELECT count(*) … GROUP BY status` has no spelling, because that is a chart.

- **No mutation, and no side effects of any kind.** An expression is a definition. It gets hashed, stored, and shown to auditors.
- **No reference to another metric, scope or saved query** from inside an expression — the one place cbQL and RV&S both have leverage the DSL does not.
- **No join between two scopes** except by walking a relation that exists in the graph.

- **No path return.** No ranking, relevance, stemming or fuzzy matching.
- **No function library.** No `currentUser()`, no `@Today`. A parameter is how a run supplies such a value. It is bound before translation.
- **No history operators.** JQL's `WAS` / `CHANGED` and WIQL's `Was Ever` / `ASOF` have no DSL equivalent. History is the trend sidecar's subject and the scope's temporal marker's, never the expression's.
- **No `or` across a relation walk in a pushed-down search.** A search runs server-side over an item's own fields. So a link-touching construct is refused at construction, not at read time.
- **No `GROUP BY`.** An evaluator's buckets are declared, identified and fingerprinted. So a bucket is a thing with a history. A SQL grouping's keys are whatever the data happened to hold. That is also why a *discovered* axis (vocabulary taken from the data) is the special case that has to report which of three nothings it found.

Conclusion

The eleven languages above are not weak. Every one of them is good at the question it was built to answer. None of them was built to answer the same question as the others. That is the problem.

An engineering programme that spans requirements in Polarion, defects in Jira, tests in Xray and code in GitHub pays for this. The team learns five query languages. Every cross-tool report is written in a BI layer that never saw an engineering link. Nobody can tell when the question behind a dashboard changed. The Metrix DSL replaces those five languages with one predicate language. It replaces the BI layer with a graph the predicate runs against.

What it buys is five things together:

1. **Portability of the question.** `State == "Reviewed" and not VerifiedByTest links TST` is a sentence about an item, not about a table. The same sentence runs over Jira, over RV&S, over Polarion, over a Markdown tree in git.
2. **Reach that the source filter does not have.** A quantified hop. A correlated predicate on the far end of a link. A counting quantifier. An aggregate over a reachable set. None of them are available in JQL, cbQL, Polarion Lucene, the RV&S query definition or OSLC on their own.
3. **Two-valued semantics that make counts sum.** A predicate and its negation partition the population. A trend you compare against last quarter's is a trend you can compare. Not a trend plus an unknown bucket neither side owns.
4. **Query identity.** A canonical form. A fingerprint. "*Did the question change?*" has a machine answer. A trend series is keyed on it.
5. **A refusal discipline at the translator boundary.** Every *X* in this paper is a place where the easy translation would answer the wrong set with no error to catch it. The refusal is the feature.

What it costs is a report of a kind you probably don't want a metric for anyway. Window functions. Free-text relevance ranking. Ad-hoc projection. Writes. For every one of those, the right tool is the drill-down export or direct access to the source — not a filter language.

The sharpest way to read this paper is the question "*which customer requirements are tested?*" If you can answer it on your current stack, on your current tools, in one expression that composes with the rest of

your filter — then you have what Metrix gives you. If you cannot, then you have the problem Metrix exists to solve.

Sources

Metrix — the Metrix DSL grammar, the normative semantics of its logical expressions, its relational and transitive-numeric expression types, the parser test suites, the shipped sample books, and the production translators for each target language cited below.

JQL — Issue Links JQL functions · Most commonly used JQL functions · Field values and comparison limits (that two fields cannot be compared, and the `WAS` / `CHANGED` field list)

WIQL — Work Item Query Language syntax reference · WIQL syntax for link query (the link modes and the `ASOF` /recursive restriction — a 2010 archived post, so treat as dated) · Query operators and macros (`Under`, `In Group`, `Was Ever`, `@Me`, `@Today±x`, field-to-field comparison)

Polarion — Advanced querying · `IQueryHelper` scripting reference (the `SQL:(…)` virtual field) · Siemens tech tip — requirements not refined yet (why joins need `SQL:`) · Lucene Query Expander extension

codebeamer — cbQL Structure · cbQL field and operator reference (`LIKE`, `BETWEEN`, subqueries, arithmetic)

Windchill RV&S — Query Filter · Query filters reference (`is empty`, subquery filters, `item.live` / `item.versioned`) · `im createquery`

OSLC — OSLC Core Specification: Query · Core query syntax draft (nested property filters, the `in` list, the deliberate absence of `or`) · ClearQuest query capabilities (`oslc.select`, `oslc.searchTerms`, `oslc.orderBy`)

DOORS Classic — DXL Reference Manual · DXL attribute conditions in advanced filters · Filtering by link module

SQL, openCypher and Gremlin are compared against their standard specifications. For openCypher, also against the behaviour measured on ArcadeDB 26.x.

About T4Test

T4Test Engineering Consulting is an independent engineering consultancy. We work on critical-systems development in Europe. We build PROmeter — a language and tool family for engineering data. It is designed for teams that work under ISO 26262, IEC 62304, DO-178C and similar regimes.

To learn more, request a webinar, or discuss how the Metrix DSL would fit your tool chain, write to contact@t4test.de or visit https://t4test.de.

© 2026 T4Test Engineering Consulting. All rights reserved.