



DATABLAU · DAM

Data lineage

Product module user manual

Version 9.1.2

Contents

- 01** Data lineage
 - 02** SQLink quick start
 - 03** Administer SQLink
 - 04** Application scenarios
 - 05** Harvest metadata and synchronize lineage resources
 - 06** Prepare and verify SQLink deployment
 - 07** Explore data lineage
 - 08** SQLink integration reference
 - 09** Manage lineage spaces
 - 10** Create tasks and review parsing reports
 - 11** View and analyze lineage
 - 12** Add and manage manual lineage
 - 13** Configure namespaces, DBLink, and preprocessing
 - 14** Query lineage, transformations, and impact
 - 15** Online SQL lineage
-

Data lineage

SQLink is the Data Lineage Monitoring Platform, an independently deployable and licensed product. It provides lineage spaces, parsing tasks, maps, application scenarios, manual lineage, impact analysis, and root cause analysis. It can extend DAM's lineage capabilities; basic lineage viewing from metadata objects remains part of Metadata Management.

Product boundaries

Basic metadata lineage starts from tables, fields, and reports. SQLink's advanced features require the separate SQLINK license. Connectivity, synchronization, and navigation between DAM and SQLink depend on deployment. Similar views do not imply automatic data synchronization.

Key concepts

Term	Meaning
Lineage space	Workspace for source resources, files, parsing tasks, and results; not a database
Namespace	Identity scope distinguishing objects with the same name across systems or environments
Parsing task	Extracts processing and dependency relationships from scripts or task files and produces a report
Lineage map	Graph exploration with direction, level, depth, and filters
Application scenario	A persistent object and relationship scope for monitoring, change impact, or structural quality analysis

See [spaces](#), [tasks and reports](#), [lineage views](#), and [scenarios](#). Harvested metadata alone does not establish accurate lineage.

Users and entry points

User	Tasks	Entry points
Lineage consumer	Search objects, inspect dependencies, assess changes	Data Catalog, Lineage Map, Application Scenarios
Data developer	Parse SQL and inspect transformations	SQL Lineage, Lineage Map
Lineage administrator	Harvest metadata, parse files, manage namespaces	System Management, Lineage Management, Task Management

Plan namespaces and harvest metadata first, build and validate lineage next, then enable exploration and analysis.

Guides

- [Quick start](#): scope, initialize, harvest, validate, and analyze.
- [Explore lineage](#): discover objects through the catalog and map.
- [View and analyze lineage](#): direction, levels, depth, filters, exports, and comparisons.
- [Application scenarios](#): monitoring, impact, and structural quality.
- [Online SQL lineage](#): immediate SQL analysis.
- [Administration](#): resources, namespaces, preprocessing, and scheduling.
- [Lineage spaces](#): organize database, report, OpenLineage, and file resources.
- [Harvest and synchronize resources](#) and [parsing tasks](#).
- [Namespaces and preprocessing](#) and [manual lineage](#).
- [Integration reference](#): boundaries and API examples.
- [Deployment preparation](#): components, configuration, and acceptance.

SQLink quick start

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

Start with a specific business question, then initialize resources, harvest metadata, build lineage, verify accuracy, and analyze results. Later steps depend on the systems, namespaces, and metadata established earlier.

Try online parsing

Use **SQL Lineage** to understand how code maps to objects. This parses SQL text without executing it against a database or replacing formal metadata harvesting.

1. Open **SQL Lineage** from the left navigation.
2. Select Oracle and enter the following fictional example.
3. Leave forced metadata association disabled and click **Parse Lineage**.
4. Check the flow from `demo_input.customers` to `demo_output.customer_names`.
5. Expand both tables' two fields and check `customer_id` and `customer_name`.

 sql

```
INSERT INTO demo_output.customer_names
(customer_id, customer_name)
SELECT customer_id, customer_name
FROM demo_input.customers;
```

The same workflow and result view are illustrated in [online SQL lineage](#). Line breaks in the SQL aid readability. The objects exist in that result; the illustration does not indicate that real database metadata has been harvested. Verify source, target, arrow direction, and field names. If parsing fails, check engine and complete SQL.

For temporary SQL analysis, continue with that guide. Administrators maintaining real system lineage should follow the lifecycle below.

Implementation stages

Stage	Work	Completion evidence
Define the scenario	Goals, systems, plan	Question, scope, and owner are explicit
Initialize	Users, organizations, systems, sources, tags	Required base objects exist
Harvest metadata	Inventory, namespaces, harvesting	Target metadata is in the lineage space
Build lineage	Import files, preprocess, parse	Cross-system relationships are queryable
Validate	Parsing, coverage, structural quality	Key paths meet acceptance criteria
Analyze	Catalog, map, and scenarios	Results answer the original question

Step 1: Define the business scenario

Specify the object, direction, output, and reviewer. Questions might concern downstream tasks and reports affected by a change, the origin of a quality issue, systems traversed by a business flow, or nodes without matching metadata.

Inventory source systems, processing systems, warehouses, reports, and file exchanges. Record technology, environment, owner, sources, harvesting methods, and available SQL or ETL files. Prioritize onboarding and begin with a representative path that can be checked manually.

Step 2: Initialize the system

Confirm accounts, organizations, and functional permissions for administrators, consumers, and developers. Menu visibility does not grant access to every object.

Register application systems to provide business ownership for sources and high-level graph objects. Names and abbreviations identify systems; referenced systems cannot be deleted directly. Create reusable source connections and test after credential changes. Connectivity does not verify harvesting scope. Prepare consistent tags for temporary tables, core paths, or responsibility domains.

Step 3: Harvest metadata

Inventory databases, reports, files, and custom metamodel resources. Check scope, service account permissions, and refresh frequency.

Namespaces provide top-level identity. Separate development, test, production, and systems with colliding paths. Set a default schema when SQL omits it.

1. Create a database, report, or file harvesting task.
2. Select application system, source, and scope.
3. Assign the correct namespace.
4. Configure supported sampling, name enrichment, and blacklist options.
5. Save, execute, and monitor the task.
6. Verify Database Resources or Report Resources in the lineage space.

See [resource synchronization](#) and [namespace mappings](#).

Step 4: Build lineage

Source	Method	Key settings
SQL scripts	Script task	Engine, namespace, default schema
ETL files	Tool-specific ETL task	Source and target connections, mappings, preprocessing
Custom Excel relationships	Download, complete, and import a template	Object paths and namespaces
Database objects	Synchronize then parse automatically or manually	Engine and metadata constraints
Reports	Synchronize and parse preparation SQL	Connection namespaces
OpenLineage	Consume standard events	Event source and job state

Check variables and every connection mapping before parsing. An unmapped connection can cause failure.

Choose the type matching the code source, then verify engine, namespace, and import mode. The exact task form depends on the deployed SQLink release; see [parsing tasks](#).

Step 5: Validate accuracy

Parsing success: inspect successful, partially successful, parsing, failed, and unparsed files. Open highlighted failures and details for partial results.

Metadata coverage: compare lineage nodes against database and report metadata in the same namespace. Investigate harvesting scope, names, and namespace when nodes cannot match real tables or fields.

Structural quality: investigate isolated tables, cycles, breaks, and field consistency using source code, schedules, and business processes.

Step 6: Analyze lineage

Use Data Catalog for precise object lookup, Lineage Map for global flows, general scenarios for persistent monitoring and similar-path comparisons, Impact Analysis for changes, Quality Analysis for graph issues, and SQL Lineage for immediate code inspection.

Completion checks

- Systems, sources, and namespaces match the plan.
- Metadata is synchronized and key objects are searchable.
- Failed and partial files are resolved or recorded.
- Owners verify sample endpoints, intermediate nodes, and direction.
- Users access paths and reports within permissions.
- Analysis answers the original business question.

Related guides

- [Explore lineage](#)
- [View lineage](#)
- [Application scenarios](#)
- [Administration](#)

Administer SQLink

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

Manage harvesting, spaces, parsing, preprocessing, edits, and scheduling. Use [resource synchronization](#), [parsing tasks](#), [namespaces](#), [manual lineage](#), and [scheduling](#) for detailed procedures.

Implementation sequence

1. Inventory systems, environments, sources, and code.
2. Plan namespaces and default schemas.
3. Harvest metadata and inspect results.
4. Add database, report, or OpenLineage resources to spaces.
5. Create parsing tasks and configure preprocessing and mappings.
6. Resolve report failures and omissions.
7. Add evidence-based relationships beyond automatic coverage.
8. Schedule refreshes and monitor parsing, coverage, and path quality.

Harvesting management

Data sources: configure source, scope, sampling, name enrichment, and blacklist. Check that exclusions do not remove critical dependencies. Inspect task state and sample schemas, tables, and fields after execution.

Reports: test connections before saving, then inspect background execution, counts, and failures.

Files: configure the matching engine, namespace, and schema for DDL or other inputs. For API-submitted files, verify source identifiers and metadata context.

Custom metamodels: confirm types, attributes, and relationships before harvesting. Verify object types and connections afterward to avoid isolated nodes.

Lineage spaces

Inspect saved relationships, database resources, report datasets and connections, and OpenLineage identifiers, inputs, outputs, and events. Resolve missing resources through harvesting rather than long-term manual substitutions.

Lineage tasks

The current English environment did not provide a stable task-form capture for this overview. Use the task labels and settings described above, then confirm the actual form in your deployed release.

Use `Script` for SQL and scripts, `ETL` for supported tool exports or task definitions, and `Container` of `User-Defined-Object` for saved manual lineage that cannot be parsed again. Record purpose, owner, source, engine, namespace, and execution mode. Test a small sample.

Full import overwrites repeated files and resets them to unparsed, including nested paths. It does not necessarily delete all existing files. Incremental import skips identical content, size, path, and name combinations and updates differences.

Before forced metadata binding, align namespaces, connections, schemas, and naming. Repair context when binding fails rather than selecting objects by name alone.

Preprocessing

Use controlled JavaScript, regular expressions, or models to normalize variables, connections, and irregular text. Reference task variables and connections as required. Compare before and after text, preserve semantics, and retain change records.

Parsing reports

Result	Meaning	Action
Success	Parsing completed	Sample key paths
Partial success	Some statements or objects parsed	Inspect failures, correct, and rerun
Syntax error	Input does not match engine syntax	Check engine, complete code, and variables
Missing metadata	Parsed nodes cannot bind	Harvest missing objects or fix namespace/schema
Other error	Runtime or unknown failure	Inspect logs and retain a reproduction sample

Review parsing success and metadata coverage separately. Map DBLink identifiers to correct connections and namespaces to avoid broken or cross-environment paths.

Namespace and edit management

Record system, environment, resource type, and naming rules. Changing a task's namespace affects matching; assess related resources first. Manual edits require evidence. Current-edit deletion does not remove original canvas content; saved custom files can be viewed or deleted but cannot be edited again.

Scheduling and operations

Select time templates and recurrence. Multiple templates use the intersection of executable dates; deleting or disabling a referenced template removes its constraint. Monitor last and next runs, state, file counts, reports, and repeated failures. Verify an actual trigger after schedule changes.

For new sources, confirm network, permissions, database type, and compatible drivers. Test driver or parser plugin changes in a controlled scope and record affected sources and tasks. Keep passwords out of documentation and screenshots.

Verify results

Check namespace placement, blacklist scope, failure outcomes, separate parsing and coverage acceptance, cross-source mappings, manual edit evidence, and actual scheduled runs.

Troubleshooting

Symptom	Check first
Metadata missing from space	Connectivity, scope, blacklist, task, permissions
File parsing fails	Complete file, engine, variables, preprocessing, logs
Many unmatched nodes	Namespace, schema, harvesting, case
Cross-system breaks	Connections, DBLink, and coverage of both upstream and downstream tasks
Duplicate nodes	Namespace planning, identifiers, import modes
Schedule not triggered	Enabled state, template intersection, dependencies, logs

Related: [quick start](#), [exploration](#), [lineage views](#), [scenarios](#), [SQL lineage](#).

Application scenarios

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

A scenario saves object scope and viewing settings for a specific analysis goal. It organizes sources, entities, and metrics into a persistent business scope using existing metadata and lineage. It does not copy data or replace harvesting and parsing.

Prerequisites

Target objects must be in the catalog, key lineage must be parsed and sampled, and users need object access. Administrators assign create, edit, and delete permissions. Identify a scenario owner, objective, and result recipient.

Select catalog objects after checking system and namespace. Follow scenarios for continued observation. Sharing does not expand recipients' existing object permissions.

General scenarios

Use a general scenario to monitor a core business path and changes. A core path is the set of objects and dependencies selected for the scenario; SQLink does not determine business importance automatically.

1. Open **Application Scenarios** and create a general scenario.
2. Enter a business-specific name and description.
3. Add sources, entities, or metrics from the catalog.
4. Verify system, environment, and dependency scope.
5. Save and inspect the core path.

Authorized users can edit scope, delete, or open management views. Confirm that monitoring and collaboration no longer depend on a scenario before deletion. Investigate change events using task logs and actual release records before deciding whether to notify owners.

Impact analysis

Object	Purpose	Inspect
Data source	Assess structure or harvesting scope changes	Entities, tasks, reports, downstream systems
Entity	Assess tables or views	Entity and field dependencies
Metric	Assess definition or dependency changes	Sources, calculations, consumers
Task	Assess schedule or transformation changes	Inputs, outputs, dependent tasks

Before a change

1. Select the object to change.
2. Set direction, level, and depth.
3. Inspect direct and complete downstream paths.
4. Identify critical systems, reports, tasks, and owners.
5. Save or export results for review.
6. Record risks and rollback in the change plan.

After a change

Repeat the same analysis and compare new, missing, or reversed relationships. Check whether harvesting and parsing were refreshed before diagnosing an implementation gap.

Impact broadcasts

Broadcasts notify relevant people of identified potential impact. They do not calculate lineage or replace change approval. Confirm automatic and manual trigger conditions, recipients, and delivery settings with an administrator.

Structural quality analysis

Check isolated tables, cycles, breaks, and upstream/downstream consistency.

Name	Type	Owner	Related Users	Update Time	Actions
通用场景	Normal	冷鹏(lengpeng)		2026-06-22 15:38:06	Edit
LENG质量分析	Quality Analysis	超级用户(admin)		2026-03-20 18:36:53	Edit
LENG的影响分析	Impact Analysis	超级用户(admin)		2026-03-20 18:09:45	Edit
我的测试	Impact Analysis	超级用户(admin)		2026-02-02 15:59:00	Edit
Demo	Normal	超级用户(admin)		2025-12-17 10:30:01	Edit
血缘影响	Impact Analysis	超级用户(admin)		2025-10-23 10:22:05	Edit
质量分析-事件测试	Quality Analysis	超级用户(admin)		2025-08-26 15:31:27	Edit
影响分析-事件测试	Impact Analysis	超级用户(admin)		2025-08-26 15:20:28	Edit
事件测试数据源	Impact Analysis	超级用户(admin)		2025-08-20 14:44:40	Edit
事件测试	Normal	超级用户(admin)		2025-08-19 17:20:39	Edit
质量分析-高于	Quality Analysis	超级用户(admin)		2025-08-08 16:19:50	Edit
质量分析-精度小于等于	Quality Analysis	超级用户(admin)		2025-08-08 10:35:36	Edit
质量分析-长度不是4倍	Quality Analysis	超级用户(admin)		2025-08-08 09:51:19	Edit
质量分析-长度4倍	Quality Analysis	超级用户(admin)		2025-08-08 09:37:49	Edit
质量分析-长度小时转换失败	Quality Analysis	超级用户(admin)		2025-08-08 09:34:58	Edit
质量分析-字段类型转换失败	Quality Analysis	超级用户(admin)		2025-08-07 13:41:45	Edit

Isolated tables: determine whether processing relationships are expected, related code is harvested, namespaces match, and parsing succeeds. Legitimately independent tables are not defects.

Cycles: compare iteration logic, mappings, code, and schedule records. Do not delete relationships based only on appearance.

Breaks: inspect output/input name mappings, temporary tables, cross-source connections, variables, and missing tasks.

Consistency: assess field names, types, and lengths against transformation code. Differences may be intentional transformations.

Verify results

Confirm scenario purpose, object identity, direction, level, depth, broadcast recipients, and investigation outcomes or owners for structural issues. Rerun after correcting harvesting, mappings, or code.

Troubleshooting

Objects are missing. Check harvesting, catalog inclusion, and permissions.

Impact is unexpectedly small. Check depth and filters, tasks, namespace, and coverage. Missing discovered impact does not exclude unharvested dependencies.

Issues increase suddenly. Review recent harvesting, parsing, or mapping changes and sample objects before adding manual relationships.

Related: [lineage views](#), [exploration](#), [administration](#).

Harvest metadata and synchronize lineage resources

SQLink harvests object structure and synchronizes resources into lineage spaces for parsing and matching. A source connection can be reused across schemas; saving it does not complete harvesting.

Database harvesting

1. Prepare application systems and sources under **System Management > Data Source Management** and test connectivity.
2. Open **Lineage Management > Harvesting Management > New Harvest > Data Source**.
3. Enter name, source, and scope.
4. Configure sampling, name enrichment, and blacklist without excluding key objects.
5. Save, then select **Execute** in the task list.
6. Monitor results and verify objects in the catalog and lineage space.

Report harvesting

Choose **Report**, select the tool type, enter service URI, authentication, and report folders, then test and confirm. Check background task status and associations among reports, datasets, and database objects. Map report database connections to correct namespaces before parsing.

DDL files and APIs

Choose **File**. For DDL, select DDL Scripts, namespace, engine, and default schema. The engine determines syntax; the namespace determines storage and identity. Their database types can differ when appropriate to the actual semantics.

API metadata uses the built-in REST API metamodel through JSON files or direct HTTP access. Verify objects and associations; ordinary prose API descriptions are not structured definitions.

Custom metamodels

Create a file harvest, select **Metamodel** as database type, choose the existing model and namespace, download the template, populate it, upload, and run. See [metamodel design](#) and [import](#).

Database and report resources

Harvesting automatically creates resource tasks. Database resources include fields, views, and procedures; report resources include structure and preparation SQL. Update metadata through harvesting rather than direct editing or deletion. You can change the engine and parse again.

Resynchronizing metadata first clears the task's existing metadata and lineage. Confirm need and source availability, then verify restored resources and paths. Assess relationship impact before changing namespaces.

OpenLineage

OpenLineage tasks consume third-party Kafka events, run state, and parent-child job relationships. Column lineage events can create relationships; event SQL can become lineage files. Verify topic, event format, producer, namespaces, inputs, outputs, and generated relationships. Event ingestion does not replace database harvesting, and missing events prevent conclusions about full coverage.

Prepare and verify SQLink deployment

Implementation and operations teams should use the project's delivery manifest for packages, topology, sizing, and scripts. After service acceptance, proceed to [metadata and lineage setup](#).

Prepare a deployment manifest

Area	Prepare	Verify
Topology	Standalone or DAM-integrated services and nodes	Match the delivered architecture
Hosts and storage	Hosts, addresses, deployment user, installation, data and log directories	Required access and planned capacity
Database	Instance, character and case settings, account, backups	Application connectivity and initialization privileges
Configuration	Nacos address, namespace, application settings, registration addresses	Reachability from actual clients
Network	Browser entry, interservice links, source connections	Every configured segment
License	Required file and machine information	Match license to deployment
Materials	Components, services, frontend, templates, scripts	Compatible packages in an archived manifest

Use the agreed deployment user and directories. Reference script paths do not authorize overwriting existing directories.

Components and responsibilities

Component	Responsibility	Verify
Relational database	Business persistence	Instance, account, database, case behavior
Nacos	Registration and configuration	Namespace, settings, registration
Redis	Cache	Address, authentication, runtime user
Elasticsearch	Full-text search	Service, permissions, resource limits
Nebula	Graph storage	Graph and registered storage nodes
Kafka	Messaging	Connections and configuration
User and base services	Identity and shared functions	Database and configuration connectivity
Metadata, scheduler, SQLink	Harvesting, execution, lineage	Logs, registration, interservice calls
Application gateway, Nginx	Backend routing and frontend	Targets, static directory, entry paths

The application gateway routes platform requests; DDS's security gateway performs dynamic masking. Deploy shared services such as standards according to the integration plan.

Install and configure

1. Prepare database and accounts with initialization permissions.
2. Install base components from the manifest and verify each, including registered Nebula storage nodes.
3. Extract matching service packages and scripts into agreed directories.
4. Configure registration address, ports, Nacos, and JVM memory for actual resources.
5. Publish configuration in the correct Nacos namespace; check database, component, and service links.
6. Start identity, base, and business services as the agreed user. Check process, port, logs, and registration after each.
7. Deploy frontend resources and Nginx routes. Verify SQLink entry and API paths.
8. Initialize licenses, roles, and sources, then perform acceptance checks.

Common configuration names include `datablau_usermgmt.yaml`, `datablau_base.yaml`, `datablau_metadata.yaml`, `datablau_sqlink.yaml`, and `datablau_gateway.yaml`. Use names and namespaces delivered with the actual packages.

Layered acceptance

Layer	Test	Pass criterion
Processes	Check components and services	Stable execution without exit or restart loops
Registration	Inspect Nacos	Correct reachable addresses and ports
Entry	Log in and open SQLink	Pages, resources, and business requests work
Metadata	Harvest a controlled source sample	Expected schemas, tables, and fields
Lineage	Parse a known script	Report and correct dependencies
Scheduling	Run a controlled schedule	Triggered on time with retained results
Permissions	Test business roles	Authorized visibility and operations

An accessible homepage does not validate harvesting, parsing, or scheduling. Retain environment summary, task IDs, times, samples, and conclusions.

Backups and recovery

Back up the database regularly and inventory graph data, uploads, service configurations, and frontend resources. Define scope, frequency, retention, and recovery order. One database backup does not prove whole-system recoverability.

Rehearse recovery in isolation. Verify login, counts, key paths, and task configuration. Check schedule state before allowing restored tasks to access sources.

Startup troubleshooting

Symptom	Check
Elasticsearch rejects root	Use the agreed non-root user and directory ownership
AccessDeniedException	Runtime access to configuration and data directories
File descriptor or memory map limit	Operations should adjust limits according to component errors
Missing registration or failed calls	Nacos, namespace, addresses, ports, network
Frontend works but APIs fail	Nginx, gateway, target service logs
Harvesting works but no lineage	SQLink and graph service, parsed results, namespace

Related: [resource synchronization](#), [parsing tasks](#), [scheduling](#).

Explore data lineage

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

Data Catalog starts from harvested metadata for precise table, field, report, or metric lookup. Lineage Map starts from parsed objects for an overall view of flows.

Choose an exploration path

Aspect	Data Catalog	Lineage Map
Content	Harvested metadata	Saved parsing results
Purpose	Find a known object	Explore cross-level flows
Filters	Keywords, system, source, type, tags	Global and in-graph search, level, depth, tags
Detail	Business names, field types, and metadata	Simplified objects emphasizing relationships
Next actions	View lineage, add to a scenario	Drill down, filter, analyze impact, export

Use Data Catalog

1. Open **Data Catalog** and search a name or keyword.
2. Filter by application system, source, and asset type.
3. Add tags or owner criteria as needed.
4. Verify object type, path, and system.
5. Open lineage or add the object to a scenario.

Investigate mismatches

Condition	Display	Action
Table metadata without table lineage	Metadata and fields remain visible; paths can be absent	Check file harvesting and parsing
Table lineage without metadata	Difference styling; field types may use default icons	Check harvesting scope and namespace
Field metadata without lineage	Field appears in the metadata drawer without a relationship	Check granularity and SQL
Field lineage without metadata	Difference styling	Check field harvesting, names, and mappings

These differences usually indicate that harvesting, parsing, and object identity are not aligned.

Use Lineage Map

1. Open **Lineage Map** and globally search one or more schemas, entities, or properties.
2. Choose upstream, downstream, or full lineage.
3. Drill from schema to entity or property level.
4. Adjust depth to control expansion.
5. Use in-graph search to locate displayed objects.
6. Filter by tags or entity types.

Map object counts can differ from catalog counts. Verify current object, namespace, and system. Use a scenario for persistent monitoring of a defined scope; ask an administrator to investigate space, namespace, and parsing details for mismatches or breaks.

The current English environment did not provide a stable, distinct catalog-search capture. Follow the filters and object checks above, then confirm the deployed catalog layout.

Verify results

Check environment identity, metadata attributes against harvested results, and graph direction against actual processing. Record mismatches and assign harvesting or parsing owners.

Related guides

- [View lineage](#)
- [Application scenarios](#)

- [Administration](#)

SQLink integration reference

This reference describes documented interfaces for developers integrating lineage. Verify paths and parameters against the target deployment. Examples contain no real hosts, credentials, or object IDs.

DAM integration

DAM can use SQLink management for spaces, preprocessing, and namespaces. Metadata-object lineage and the standalone Data Lineage module have different licensing scopes; see [licenses](#). Embedding an object lineage view does not grant permissions.

Request conventions

Append the paths below to `/sqlink` on the deployed service. All listed operations use POST. Requests and responses normally use JSON; uploads use `multipart/form-data`; template downloads return binary data.

Use deployment-specific unified authentication, including actual headers, token acquisition, and expiry. Distinguish invalid parameters, authentication, missing resources, and service errors. Verify business output even after HTTP success.

Namespaces, tags, and custom relationships

Operation	Path	Request
Create namespace	<code>/dataArea/createDataArea</code>	Body: required name, objectType; optional defaultSchemas
Metadata coverage	<code>/dataArea/getMetadataCoverage</code>	Query: required Long namespaceId
Delete namespaces	<code>/dataArea/deleteDataArea</code>	Body: ID array; related lineage files prevent direct deletion
Create custom lineage	<code>/customLineage/create</code>	Body: required name, edges; optional description; edges contain startId, endId
Tag objects	<code>/tag/addTag</code>	Body: objectId, tagIds, optional propagation
Find tagged objects	<code>/tag/getByTag</code>	Query: required Long tagId

Propagation uses diffusion, direction, and depth. Directions are FORWARD (downstream), BACKWARD (upstream), and ALL. It is disabled by default; define depth and scope before enabling.

Graph, transformation, and impact queries

Operation	Path	Request and result
Query graph	/lineageGraph/query	Body nodes and conditions; returns a subgraph
Get transformations	/lineageGraph/getTransform	Graph conditions and id or queryIds; optional query isGraph
Get impact	/lineageGraph/getImpact	Node, direction, depth, resource types

See [query API details](#) for types, defaults, responses, and direction differences. Verify granularity and temporary-object handling; graph and impact contracts differ.

Files, SQL, and broken-lineage jobs

Operation	Path	Request
Upload file	/lineage/uploadLineageFile	Query: required lineageEngine, objectType, folderId; optional defaultSchema, parse; multipart file
Parse SQL	/lineage/parse	Body: required sqlContent; optional defaultSchema, sqlType
Analyze broken lineage	/lineage/executeBrokenLineageJob	Body may limit sourceNamespaces, targetNamespaces, direction

 json

```
{
  "sqlContent": "INSERT INTO demo_target.orders SELECT order_id FROM demo_source.orders",
  "defaultSchema": "demo_target",
  "sqlType": "DML"
}
```

Replace example schemas with authorized context. Parsing is separate from executing SQL. After upload, verify file ID, parsing state, report, and object matching.

Scripts and scheduling

Operation	Path	Request
Create preprocessing script	/script/createScript	Body: name, scriptType, objectType, scriptContent; optional enabled
Download script template	/script/exportScriptTemplates	Returns an Excel import template
Create job	/job/createJob	Body: name, binding, schedule, execution class
Run now	/job/runJob	Query jobId; optional descriptor body overriding defaults

scriptType is JAVASCRIPT, REGEX, or LLM. Job creation requires name, bindResourceId, bindResourceType, schedule, and className, with optional description and jobContent. Obtain binding types and execution classes from the supported definition rather than inventing Java names.

Running returns an instance identifier, not completion. Check execution records. Deletion, propagation, upload with parsing, and execution change service data; bound integration tests and avoid resubmitting successful mutations.

Manage lineage spaces

A lineage space organizes resources, tasks, namespaces, and parsed revisions. It is neither a database nor a metadata catalog. A space groups work; a namespace identifies objects. Verify mappings even after files are added.

Resource types

Area	Content	Sources
Lineage Management	SQL, ETL, Excel files and results	Scripts, ETL, custom relationships
Database Resources	Harvested objects and models	Databases, DDL
Report Resources	Reports, datasets, preparation SQL	Report harvesting
OpenLineage Tasks	Run events and job relationships	Kafka/OpenLineage

Tasks can be created manually or synchronized from harvesting. Update harvested database and report objects at their source task. The Lineage Management resource area is distinct from the similarly named DAM metadata license.

Plan spaces and folders

Define business scope, source systems, environment, owner, and engine. Group spaces by domain or environment and folders by source and purpose. Keep development, test, and production scripts distinguishable. Configure roles, space permissions, and task permissions before shared write access.

Create lineage tasks

1. Open **Lineage Space > Lineage Management > New Task**.
2. Select folder, engine, and import method.
3. Configure full or incremental import, source folder, and metadata constraints.
4. Bind namespace, default schema, DBLink, and preprocessing.
5. Save and test a small sample before expanding.

Full import processes all files in scope. Incremental import compares content, size, path, and name. Forced binding requires parsed objects to match harvested metadata; an unmatched table or field can fail the task. Check coverage first.

Database and report resources

After database or DDL harvesting, check database, schema, table, and field counts and namespaces, then run parsing and inspect its report. Report resources contain reports, datasets, and preparation SQL. Resolve failures through connections, namespace mappings, and engine settings rather than direct metadata editing or deletion.

OpenLineage tasks

Verify event source, Job identifiers, event order, RunEvent times and states, parent-child relationships, and generated column lineage. Missing events should not be represented as observed runtime facts through manual edits.

Namespaces and schemas

Configure database type, default schema, connection, and DBLink mappings before parsing. Inspect object identity before correcting unbound namespaces, incorrect schemas, or inconsistent connection names. See [namespace configuration](#).

Preprocessing and reports

Preprocessing normalizes variables, repairs format, or extracts embedded SQL using configured JavaScript, regular expressions, or models. Retain original files and script revisions. Reports distinguish success, partial success, syntax errors, missing metadata, and other errors. Inspect failed fragments and re-execute after correction; submission success is insufficient.

Check space quality

1. Review parsing success and partial counts.
2. Inspect coverage and unbound objects.
3. Investigate breaks, cycles, and isolated nodes.
4. Retain revision, time, and file traceability.
5. Have owners sample key table, field, and report paths.

Counts and coverage are clues, not proof of correct business lineage.

Revisions and corrections

Retain files and task records for automatic results. Use manual relationships only with evidence. Record objects, direction, reason, and owner before changes. Prefer reharvesting or reparsing to repair configuration, then remove incorrect manual relationships.

Troubleshooting

Symptom	Check
Successful task without a graph	Resource inclusion, namespace, partial report results
Tables without field paths	Field harvesting, SQL references, forced binding
Wrong same-name table	Namespace, schema, DBLink
Missing report lineage	Harvesting, connection, preparation SQL engine
Repeated incremental parsing	File path, name, size, content, and import mode

Related: [resources](#), [tasks](#), [lineage views](#).

Create tasks and review parsing reports

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

Lineage tasks organize source files, preprocessing, engines, and object matching. Prepare metadata and namespaces, then validate a small file sample.

Create a task

1. Open **System Management > Lineage Management > Lineage Space > Lineage Management**.
2. Select **Add Task** in the target folder.
3. Choose Script or ETL. Custom Excel lineage uses ETL. Container of User-Defined-Object contains saved manual lineage and cannot be parsed again.
4. Configure task, engine, and namespace; import the required format.
5. Check preprocessing variables and map every connection to its namespace.

Use actual task types and objects rather than screenshot values.

The current English environment did not provide a stable task-form capture. Confirm the exact field labels and available engines in your deployed release.

Import modes

Mode	Behavior
Full	Imports the directory's scripts, overwrites duplicate files, and resets them to unparsed; supports nested folders and matching names in different folders
Incremental	Compares content, size, path, and name; skips identical files and updates changed files

Check counts, paths, and state after import rather than judging duplicates by name alone.

Metadata constraints

Forced metadata binding is off by default. When enabled, an unmatched table or field can fail the task. Verify harvesting, schemas, names, and mappings. Any connection lacking a namespace can cause parsing failure; check source and target connections individually for cross-system ETL.

Execute and inspect

Each parsing run records start and end times, state, and counts for success, partial success, syntax errors, missing metadata, and other errors.

1. Review totals and file errors.
2. Open highlighted failed fragments for partial files.
3. Compare original and preprocessed text.
4. Correct engine, connections, metadata, or preprocessing and parse again.
5. Sample actual graph nodes, directions, and field relationships.

Parsing success and metadata coverage measure different aspects. Verify object identity even for successfully parsed files.

Related: [namespaces and preprocessing](#), [lineage views](#).

View and analyze lineage

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

This guide covers graph operations used in Data Catalog, Lineage Map, and Application Scenarios: search, direction, level, depth, filters, exports, processing detail, code slicing, and comparisons.

Direction is upstream, downstream, or both relative to current objects. **Level** is object granularity. **Depth** is the number of relationship steps expanded. Changing one does not change the others. An undisplayed node may still exist.

Search lineage

Global search

Search schemas, entities, properties, or script files. Select multiple schemas, entities, or properties to view their full paths together. A script opens a detailed lineage view. Verify systems and namespaces before combining objects.

In-graph search

Find and highlight schemas, entities, properties, or tags already loaded on the canvas. In-graph search does not expand depth or load additional objects.

Choose direction

The current entity is the analysis reference. **Upstream** shows objects producing or affecting it. **Downstream** shows objects it may affect. **Full lineage** includes both directions. Recheck current entity after changing direction; multiple current entities also change export and impact scope.

Choose a level

Drill from platform or schema to entity and property level, or navigate upward. A schema organizes database objects; entities include tables, views, reports, and files; properties include fields; operators are computation or transformation steps.

Level	Question
Schema	How does data move between systems or database areas?
Entity	Which tables, views, reports, or files depend on each other?
Property	Where does a field or metric originate and flow?
Operator	How do SQL steps compute or pass a field?

Set depth

Depth controls successive groups of directly related objects. Start small and expand key paths gradually to keep large graphs readable.

Filter objects

Filter by entity type or tag, with single or multiple selection. The filter icon reports hidden items. For $A > B > C$, hiding B can display $A > C$. Clearing the filter restores the original view; it does not delete B or its relationships. Use filters for temporary tables, core tables, responsibility domains, and systems.

Canvas controls

Control	Action
SQL	Show or hide SQL in script lineage
Zoom	Adjust object scale
Full screen	Expand workspace
Display	Control temporary objects, Chinese business names, and indirect relationships
Edit	Enter controlled manual editing

Edit mode provides create, undo, redo, delete, and save. Delete applies only to objects and lines added in the current edit.

Export impact reports

Choose table or field granularity, upstream/downstream/full scope, object types, whether to exclude intermediate processing, and an available format. Reports can include objects hidden by display controls, so they may exceed the visible canvas.

Right-click an object and select **Impact Analysis** for its all-depth report. Multiple current entities produce separate reports. Without an explicit current entity, scope may include all graph objects. Verify the reference before exporting.

Processing detail and operator lineage

The second-level lineage view expands processing details beneath an object relationship graph; it is not an independent lineage dataset. Tables, fields, and operators support this view. Operator lineage connects computation steps to inputs and outputs. **fx** marks a calculation expression. Highlight an operator's upstream and downstream paths to inspect its relationships.

Check processing detail against full SQL, variables, and scheduling context.

Code slicing

Extract relevant processing fragments for a selected entity or property. Slicing changes the displayed analysis scope without editing source code.

To inspect upstream writes, select the entity, right-click **Code Slicing**, and choose input analysis. To inspect a field's downstream use, expand the entity, select the field or metric, open **Code Slicing**, and choose output analysis.

Slicing helps trace definitions across steps, but a focused fragment is not necessarily a complete executable script.

Compare code and lineage

In a space file's comparison view, inspect line-by-line code revisions with syntax and difference highlighting.

Lineage comparison highlights added, modified, and deleted nodes or relationships across two parsed results and lets you switch sides. Confirm comparable harvesting scope, namespace, and parsing rules before attributing differences to business changes.

The current English environment did not provide stable, distinct captures for operator detail, code slicing, and code comparison. Use the controls and verification points above, and confirm the exact layout in your deployed release.

Verify results

Check current entity, system, namespace, direction, adequate level and depth, hidden objects, consistency between fragments and full scripts, report scope, and comparable parsing conditions.

Related: [exploration](#), [scenarios](#), [SQL lineage](#), [administration](#).

Add and manage manual lineage

Use manual lineage for small, evidence-based gaps. Resolve recurring omissions through harvesting and parsing rather than inferred relationships.

Edit relationships

1. Open property-level lineage in **Lineage Map**, then choose **Edit**.
2. Create entities as needed and verify identity.
3. Pin source and target fields, then drag from the source field's plus sign to the target.
4. Click blank space to cancel a pending connection. Use undo and redo for current changes.
5. Delete only entities or green connections added in this edit; existing canvas content cannot be deleted here.
6. Verify direction, objects, and evidence, then save.

Saving creates a timestamped custom file with extension ULE and exits editing. Leaving, switching, or refreshing discards unsaved edits.

Saved relationship limitations

Saved manual lineage can be viewed or deleted but cannot be edited again. Record original relationships and reasons before correcting them; do not assume saved files support undo.

Manage files

Find generated files under **System Management > Lineage Space > Lineage Management > Custom Lineage**, or through the task list between graph entities. Deletion affects the file's relationships; verify the file and business impact first. Retain source evidence, maintainer, and reason, and check direction, field granularity, and metadata associations.

Configure namespaces, DBLink, and preprocessing

Namespaces distinguish identically named objects across systems and environments. Preprocessing transforms variables, connections, and irregular text into parseable input. Both affect correct object matching.

Plan namespaces

Open **System Management > Lineage Management > Namespaces** to create or edit unique names and attributes. Define technical source, environment, and lifecycle. SQL executed within a database usually uses its metadata namespace; cross-system ETL maps each source and target connection separately.

The default schema fills omitted schema names. Use the actual execution context rather than an arbitrary value to suppress missing-object errors.

Configure DBLink

Review harvested DBLinks or add one manually in DBLink mapping management.

Setting	Meaning
DBLink name	Connection keyword in code, such as remote in customer@remote
Default schema	Schema used when an object referenced through the link omits one
Creation namespace	Namespace of the database owning the DBLink
Target namespace	Namespace of the connected database

Test a cross-database sample after mapping.

Apply preprocessing

1. In **Preprocessing Management**, choose JavaScript, regular expression, or a large language model.
2. Define transformations for the database and code pattern. Configure model interfaces before model-based processing.
3. Select rules in the task's **Preprocessing** settings.
4. Check extracted variables and connections; export, maintain, and import them in bulk if needed.

5. Compare a small sample before and after processing. Preserve business meaning.
6. Parse and verify reports and lineage.

Models can extract SQL from Java or Python or suggest syntax corrections. Review output and retain originals; inferred rewrites are not evidence of actual system execution.

Diagnose broken paths

Check harvested metadata, all connection mappings, default schema, DBLink direction, variable replacement, and omitted statements. Reparse after corrections and verify results.

Query lineage, transformations, and impact

Use the [request conventions](#). All methods are POST, with paths appended to /sqlink. Replace example IDs and tags with actual environment identifiers; names and IDs from other environments are not interchangeable.

A subgraph is a filtered part of the lineage graph. Transformation logic explains processing between inputs and outputs. Impact identifies potential dependencies in harvested and parsed relationships; it does not establish actual business loss.

Query a subgraph

For cross-module integration, see the [DAM lineage API scenario](#) and verify the deployed contract and enum differences.

Path: /lineageGraph/query. Parameters are in the JSON body.

Parameter	Type	Required	Meaning and documented default
id	String	Yes	Target node ID
queryType	String	No	SCHEMA, TABLE, COLUMN; default TABLE
depth	Integer	No	Expansion depth; default 3
direction	String	No	FORWARD downstream, BACKWARD upstream, ALL both; default ALL
hasTemp	Boolean	No	Include temporary tables; default false
tags	Array	No	Example uses tag names, unlike tagIds in tagging APIs

Set scope explicitly where possible. This ID is a placeholder:

json

```
{
  "id": "example-node-id",
  "queryType": "TABLE",
  "depth": 2,
  "direction": "FORWARD",
  "hasTemp": false
}
```

Responses contain nodes and relationships. Nodes include `id`, `name`, `parentId`, `type`, and `properties`. Relationships use `srcId`, `dstId`, `lineageId`, and `properties`. Verify start object, direction, depth, filters, and endpoint consistency. An absent object may be beyond the query scope.

Retrieve transformations

Path: `/lineageGraph/getTransform`.

- Optional query Boolean `isGraph` defaults to `true` in the source reference for in-graph mode.
- The body uses graph conditions and a target `id`. The source also mentions `queryIds` without a complete structure; verify the actual contract before batch use.
- Example result fields include `startIds`, `endId`, `transforms`, `lineageName`, `lineageId`, and `lineageTransformationType`.

Compare transforms with source code and input/output objects. Fragments may not be executable standalone. See [code slicing](#).

Query impact

Path: `/lineageGraph/getImpact`. Parameters are in the JSON body.

Parameter	Type	Required	Meaning and documented default
<code>id</code>	String	Yes	Reference node ID
<code>direction</code>	String	No	FORWARD or BACKWARD; default FORWARD; do not assume every graph enum applies
<code>depth</code>	Integer	No	Default 3
<code>resourceTypes</code>	Array	No	Filter TABLE, REPORT, COLUMN

Example response fields are `id`, `name`, `alias`, `type`, `dataArea`, and `direction`. The source's unrestricted-scope wording is inconsistent with its array type; do not invent a string ALL value for the array. Follow the deployed definition.

Results do not notify owners or modify downstream assets. Assess business implications through [scenarios](#).

Namespace statistics

Path: /dataArea/getMetadataCoverage. Required query namespaceId is Long.

Level	Lineage count	Metadata count
Schema	schemaInLineage	schemaInMetadata
Table	tableInLineage	tableInMetadata
Field	columnInLineage	columnInMetadata

Verify namespace and [harvesting scope](#). Counts alone do not prove correct one-to-one matching. Use the defined coverage calculation rather than deriving an undocumented percentage.

Find objects by tag

Path: /tag/getByTag. Required Long query tagId is an actual tag ID, not a name. This read does not add tags or propagate them.

Validate responses

HTTP status	Meaning	Check
200	Request succeeded	Business fields and scope
400	Invalid parameters	Location, requirements, types, enums, IDs
401	Authentication failed	Current credentials and access
404	Path or resource missing	Base path, existence, visibility
500	Service error	Retain time, endpoint, error summary for administrator

Record scope, IDs, status, and business outcome without real tokens or keys. Page access, HTTP success, and correct relationships require separate checks.

Online SQL lineage

Screenshot note: Screenshots are illustrative. Controls or labels may remain untranslated in the demo release, and pages without a faithful English capture retain the source illustration. Control names and layouts can vary by release.

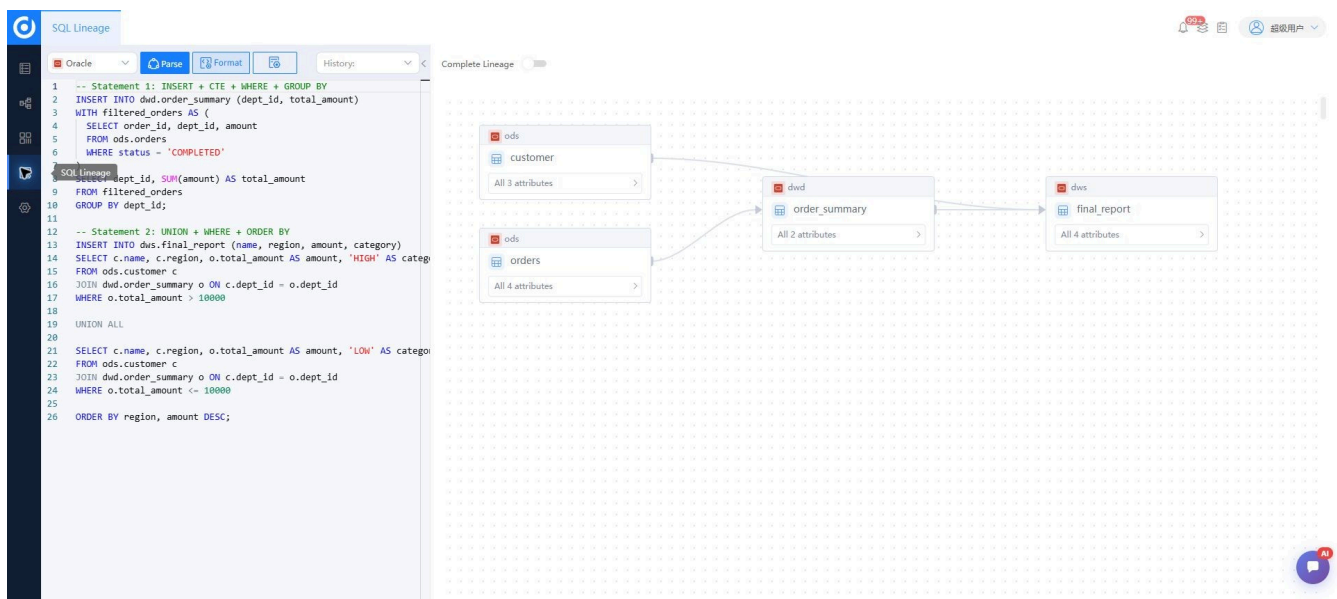
Analyze SQL inputs, outputs, intermediate processing, and field relationships without creating a formal parsing task. Use online results for development and investigation; manage ongoing code through formal harvesting and parsing.

Prerequisites

Use authorized SQL without passwords or tokens. Identify database syntax or engine. For DAM metadata associations, confirm namespace and default schema. First-time users can try the [tutorial example](#).

Parse SQL

1. Open **SQL Lineage**.
2. Select a matching SQL type or engine.
3. Enter SQL and optionally format it.
4. Open parsing settings and configure indirect relationships and metadata association.
5. Parse, then switch between table and field views to verify results.



Parsing settings

Setting	Purpose	Guidance
Show indirect relationships	Display dependencies through expressions or intermediate processes	Enable for complex transformations
Force metadata association	Bind parsed nodes to harvested metadata	Use confirmed namespace and schema
Namespace	Scope object identity	Match target metadata planning
Default schema	Complete unqualified object paths	Use actual execution context

Wrong namespace or schema can produce unmatched or incorrectly matched nodes. Validate a known small sample first.

Read results

Blue nodes are inputs, green nodes are outputs, and yellow nodes are intermediate objects or processes. Solid lines are direct relationships; dashed lines are indirect. Table view summarizes flow; field view explains origins and transformations. The full-lineage control expands intermediate paths. Use filtering and zoom for complex SQL.

Run history

Verify SQL, type, execution time, and result when reopening history. Business code in history remains subject to access and confidentiality rules.

Verify results

- Inputs and outputs match FROM, JOIN, INSERT, and creation statements.
- Mappings explain aliases, expressions, and aggregates.
- Direct and indirect display fits the question.
- Associated objects have correct namespace and schema.
- Warnings, unrecognized statements, and missing objects are recorded.

Troubleshooting

Parsing fails. Check engine, complete syntax, database extensions, and variables. Preprocess runtime substitutions using controlled samples.

Nodes lack metadata. Check harvesting, namespace, schema, and case. Avoid matching an identically named object in another source.

The graph is too complex. Start at table level with full paths or indirect relationships hidden, then expand gradually.

Related: [lineage views](#), [quick start](#), [administration](#).