sqltoy-orm

sqltoy-orm English Manual

2026-10-24 · https://github.com/sag-framework/sagacity-sqltoy

Introduction

What is sqltoy-orm?

- 1. sqltoy-orm is a Java ORM framework that combines JPA-style object CRUD with native SQL querying.
- 2. It distills and shares practices from a wide range of project types: simple business systems, large multi-tenant SaaS suites (ERP, MES, CRM) and big-data platforms.

Its history

- **2007–2008**: while building a management system for Agricultural Bank of China — full of queries and ever-changing filter conditions — an accidental insight produced a dynamic-SQL authoring style more concise than MyBatis. It began as a query-side complement to Hibernate/JPA and visibly improved development efficiency. See a blog post from 2009 (Chinese): https://blog.csdn.net/iteye_2252/article/details/81683940
- **2008–2012**: years of financial-industry projects facing tens-of-millions-row datasets focused SqlToy on enhancing SQL querying over JPA. By then Cache Translate, Fast Pagination and pivot/unpivot were already in place.
- **2013–2014**: to spare developers from mixing two technologies in one project, object-based CRUD was implemented inside SqlToy, completing the full sqltoy-orm framework.
- **2014–2017**: while leading the Lakala big-data platform, SqlToy went through a major refactoring that rationalized its internals, and was hardened on Lakala CRM and a big-data platform accumulating up to tens of billions of rows.
- **2018–present**: thoroughly tempered by complex multi-tenant SaaS ERP scenarios and real-time data warehouses combining Flink CDC with MPP databases, SqlToy has become mature and reliable — and keeps being shared and built together with the community as an open-source project!

Milestones of a few signature capabilities

- **2008** — an accidental insight produced the concise dynamic-SQL authoring model
- **2010** — thinking through a performance problem gave birth to Cache Translate
- **2011** — a demanding pagination-performance requirement inspired `@fast` Fast Pagination
- **2012** — analytics needs brought in pivot/unpivot and YoY & MoM calculations
- **2016** — first open-sourced on Alibaba's Code platform (the platform later shut down; the project moved to GitHub/Gitee in 2018 and has been maintained there since); active promotion began in 2020

Scenarios and domain extensions

SqlToy was forged in real projects and fits especially well:

- **Admin / back-office systems** — ever-changing filters and heavy complex querying are exactly the stage for SqlToy's dynamic SQL
- **Large multi-tenant SaaS suites** — ERP, MES, CRM scenarios, paired with multi-tenancy, data permissions and sharding
- **Reporting & analytics** — group summary, pivot/unpivot, YoY & MoM, tree sort & rollup ship as built-in in-memory algorithms, so SqlToy **extends naturally into a reporting platform**
- **Query-service foundation** — sqlId + named parameters make query-as-configuration a natural fit for **standing up configured query services** quickly
- **Big data / real-time warehousing** — first-class support for ClickHouse, Doris, StarRocks and other OLAP engines, proven together with Flink CDC

SqlToy Feature List

SqlToy starts from an intuitive dynamic-SQL authoring model, combining JPA-style object operations with native SQL querying. The full feature catalog by category — each row links to its detailed chapter.

1. Object Operations (JPA-style)

Feature	Description	Details
Object CRUD	Complete API set — save / saveAll / update / delete / load, etc. (~110 LightDao methods); annotated POJOs generated by quickvo	Object CRUD
Elastic update	One database interaction completes the update (unlike Hibernate's load-then-update); null fields are skipped, ensuring data accuracy under high concurrency	Object CRUD
updateFetch	Lock-query → logic validation → modify → return the latest result, all in one interaction; for flash sales, inventory and ledger accounts	Object CRUD
updateSaveFetch	Lock-query, insert when absent / update when present, returning the modified result	Object CRUD
updateByQuery	Batch single-table update by condition, supporting `set field=field+?` self-referencing expressions	Object CRUD
Cascade operations	`@OneToOne` / `@OneToMany` cascading save, load (with filtering/sorting), update and delete; batch cascade optimized with IN queries	Object CRUD
Query hierarchy packaging	One join SQL automatically packs parent-child tables into nested objects (1..n) per annotations	Query API
Tree-table route builder	`wrapTreeTableRoute` builds node route / level / leaf flags for non-recursive subtree queries	Tree-Table Route
Primary-key strategies	22/26-digit ordered ids, snowflake, UUIDv7, ULID, sequence/identity, Redis rule-based business keys; custom extensions supported	Primary Keys
Data versioning	`@DataVersion` optimistic locking against concurrent overwrites	Data Versioning
Common-field auto-fill	createdBy/updatedBy/createdTime/updatedTime/tenant filled automatically (can be paused/resumed)	Auto-Fill

2. SQL Query

Feature	Description	Details
Dynamic SQL	`#[]` condition fragments + `@if/@loop/@include/@joinFields` macros + filters; SQL with fragments still runs directly in any DB client	Dynamic SQL
Lambda conditions	Query/update/delete expressed through entity property method references (`StaffInfo::getStatus`); single-table and multi-table joins share one builder; strict `getOne` , `matchAll` guard, `setExpr` expression updates, CTE/ `@fast` pushdown/UNION composition. Built in since 6.0.8 (the former sqltoy-plus is now core)	Lambda Conditions
Link chain API	Chained entry points `query()/save()/update()/delete()/load()/unique()/execute()/batch()/store()/treeTable()/tableApi()/elastic()/mongo()` , with datasource, lock mode, batch size and forced-update fields	Link Chain API
Filters	eq / between / to-date / to-number / split / primary / cache-arg (reverse cache matching replacing LIKE) and 20+ more	Dynamic SQL
Query API forms	findOne / find / findPage / findTop / findRandom / getValue / getCount / isUnique / loadByQuery / findEntity (single-table EntityQuery shortcuts)	Query API
Cache translate	Code-to-name without joins; API access for dropdowns etc.; reverse key matching; conditional, split-value, tenant-isolated and i18n translation	Cache Translate
Pagination	Automatic count optimization + page-optimize cached pagination + `@fast` fast pagination (paginate before join) + parallel pagination + custom count-sql	Pagination
Hierarchy query	findByQuery + hiberarchy automatic parent-child packaging	Query API
Parallel query	parallelQuery runs several independent SQLs concurrently (mixed paging allowed) for faster page loads	Parallel Query
Stored procedures	executeStore / executeMoreResultStore with out params and multiple result sets	Stored Procedures
Stream query	fetchStream row-by-row callbacks for huge result sets without memory pressure	Query API
SQL execution	executeSql / batchUpdate (parallelizable) / insertReturnPrimaryKey / truncate	Link Chain API

3. Analytics (in-memory algorithms, database-agnostic)

Feature	Description	Details
Pivot / Unpivot	Row-to-column / column-to-row with dynamic columns, a few lines of config	Analytics
Group summary	summary for any-level subtotals, grand totals and averages	Analytics
YoY & MoM	cols-chain-relative (horizontal) / rows-chain-relative (vertical), automatic percentage output	Analytics
Tree sort & rollup	tree-sort orders hierarchies and rolls child values up level by level	Analytics
Group concatenation	link tag replacing group_concat / listagg / WM_CONCAT consistently across databases	Analytics
Result formatting	Date/number formatting including Chinese uppercase amounts (capital-rmb) for finance	Dynamic SQL

4. Cross-Database

Feature	Description	Details
25 dialects	MySQL, Oracle, PostgreSQL, SQL Server, DB2, SAP HANA, DM, Kingbase, HighGo, OceanBase, TiDB, ClickHouse, Doris, StarRocks, TDengine, etc.	Supported Databases
Function replacement	`functionConverts=default` adapts nvl/concat/to_char/datediff/uuid/date_add and 28 functions per database; custom extensions supported	Dialects & Functions
Dialect sqlId	`sqlId_dialect` / `dialect_sqlId` naming lets the framework pick database-specific SQL automatically	Dialects & Functions
Multi-DB verification	redoDataSources replays any query against other databases — a productization testing booster	Multi-DB Verify
Reserved words	Declare reservedWords once; object operations and cross-DB SQL are adapted automatically	Dialects & Functions

5. Enterprise Features

Feature	Description	Details
Sharding	Database and table sharding via annotations or xml strategies, routed by conditions	Sharding
Multi-tenancy	`@Tenant` + TenantFilterInterceptor for unified filtering and assignment	Multi-Tenancy
Data permission	Unified permission parameter injection and overreach validation	Data Permission
Data security	RSA field encryption/decryption with transparent query decryption; masking for phone, name, bank card, etc. (incl. discrete masking)	Security
SQL interceptors	SqlInterceptor for centralized SQL rewriting (schema replacement, injected conditions, etc.)	Interceptors
Slow SQL	Timeout-threshold logging and a slow-SQL collector handler (extensible to monitoring)	Slow SQL

6. NoSQL

Feature	Description	Details
Elasticsearch	eql supports SQL mode and native JSON mode, paging/Top, with cache translate and masking	Elasticsearch
MongoDB	mql query and aggregation pipelines, fields alias mapping + cache translate + masking	MongoDB

7. Engineering & Integration

Feature	Description	Details
quickvo generator	Generates annotated POJO/VO from database tables (standalone tool + Maven plugin), with Lombok and swagger support	quickvo / plugin
Automatic DDL	autoDDL + packagesToScan creates tables directly from POJOs	POJO to DDL
Debug mode	Prints executed SQL/params/timing; sql files hot-reload on change	Configuration
GraalVM AOT	Built into the spring-starter/solon-plugin; configuration-driven native metadata since 6.0.8	GraalVM Native Image Guide
Object mapping	convertType / MapperUtils for fast DTO↔POJO conversion (with @SqlToyFieldAlias)	Utils
Table metadata	TableApi for table/column info, truncate/drop	TableApi
Multi-framework	Spring Boot starter / traditional Spring / Solon / plain Java	Configuration
Type extension	TypeHandler for JSON and other non-standard types; geometry support via JTS	TypeHandler / Other Features

Supported Databases

SqlToy ships with **25 built-in database dialects** (SAP HANA added in 6.0; HighGo added in 6.0.5) and auto-detects the database type via `DialectFactory`, adapting pagination and functions automatically. Any database providing a JDBC driver can be queried by SqlToy.

Mainstream Relational Databases

MySQL, Oracle, PostgreSQL, SQL Server, DB2, SAP HANA, H2, SQLite, TiDB, OceanBase, PolarDB

Chinese / Domestic Databases

DM (Dameng), Kingbase, Oscar, HighGo, MogDB, Vastbase, StarDB, GaussDB, OpenGauss

Distributed OLAP Databases

ClickHouse, Doris, StarRocks, Greenplum, Impala (Kudu), TDengine

NoSQL

Elasticsearch, MongoDB

Others

Any SQL/JDBC-capable database can be queried. For databases without a dedicated dialect (especially PostgreSQL derivatives), configure `dialect` / `dialectMap` to map them onto an existing dialect — see SQL Showcase (Chinese).

> For cross-database adaptation (function replacement, dialect-specific sqlId, multi-DB verification), see the Cross-DB & Dialect chapter (Chinese).

Database Migration Guide

This chapter targets **database migration** projects (migrating from Oracle, MySQL, etc. to DM, Kingbase, HighGo, Oscar, MogDB, Vastbase, GaussDB, OpenGauss and other databases). It organizes sqltoy's cross-database adaptation capabilities into an executable six-step checklist — assess → configure → refactor → verify → perform → cut over; each step links to the chapter holding the full mechanism and parameters. For the selection rationale and overall introduction, see `docs/multi-db-adaptation.en.md` in the source repository (linked from the README's "Documentation & Resources" table).

1. Assess First: Map the Adaptation Surface

1. **Classify SQL sources**: object operations (save/update/delete/load/updateFetch, etc.) have pagination, locking, merge and primary-key backfill generated per dialect by the framework — **no refactoring needed**; what actually needs assessment is custom sql (the three forms: xml configuration, `@Sql` annotations, and code sql — see Dynamic SQL Specification);
2. **Confirm target-database coverage**: see Supported Databases for the dialect list; for domestic databases without a dedicated dialect (especially PostgreSQL derivatives), first alias them onto the nearest dialect via `dialect` / `dialectMap`:

```
spring.sqltoy.dialectMap.kingbase=mysql
```

3. **Assess primary-key strategy**: `identity` / `sequence` carry dialect and case-sensitivity differences across databases; for a database migration prefer programmatic keys (22-digit/26-digit/snowflake/UUID) to eliminate this class of difference at the source — see Primary Key Strategies; existing sequence-based tables keep working, with backfill handled per dialect.

2. Configuration: Switch On Cross-DB Adaptation

```
# automatic cross-DB function adaptation (28 built-in functions)
spring.sqltoy.functionConverts=default
# reserved words used by the project; object operations and cross-DB SQL adapt automatically
spring.sqltoy.reservedWords=maxvalue,minvalue
# proxied databases / configured dialect differs from the real one (e.g. OceanBase configured as mysql): variant lookup follows the real dialect
spring.sqltoy.realDialectFirst=true
```

- Function adaptation covers the common differences: `nvl/ifnull/coalesce`, `to_char/date_format`, `decode`, `group_concat/listagg/string_agg`, `sysdate/now/current_timestamp`, `trunc/date_trunc`, `uuid`, `date_add`, etc.; forms that cannot be converted safely are kept **loud** (passed through for the target database to error on) and never silently rewritten — see Dialect Self-Adaption & Function Extension;
- The `<sql dialect="...">` attribute (solidify parsing in a declared dialect) and `realDialectFirst` (prefer the real dialect for variant lookup) are in sections 2.1 and 2.2 of the same chapter.

3. SQL Refactoring: Handle Each Difference Class Its Own Way

Difference class	Typical symptom	Handling
Function differences	`nvl`, `decode`, `to_char`, `sysdate`, `group_concat`	Adapted automatically by `functionConverts=default` — no manual work
Pseudo-tables	oracle family requires `from dual`; on pg/mysql/sqlserver writing it errors	Since 6.0.6 use the `@virtualTable()` macro — expanded or stripped per dialect at parse time
Structural differences	oracle `connect by` vs recursive CTE, database-specific hints, etc.	Dialect variants `sqlId_dialect` / `dialect_sqlId`; the framework picks by current database
Primary-key return case	PG-folding kernels are case-sensitive on `getGeneratedKeys` column hints	The framework lowercases the whole PG-folding family by default; overridable per dialect via `dialectReturnPrimaryColumnCase`
Create-table DDL	type and partition syntax differences	POJO annotations + quickvo generate DDL per database — see POJO-to-DDL

Pseudo-table macro (expansion rules in Dialect Self-Adaption & Function Extension, section 3):

```
select '0' id,'Global' name @virtualTable()
union
select id,name from sys_tenant_info
```

Structural differences: attach dialect variants to the sqlId; business code calls only the base sqlId; lookup order is `dialect_sqlId → sqlId_dialect → sqlId`:

```
<!-- base version (recursive CTE): used on pg / mysql8+ / DM / Kingbase etc. -->
<sql id="dept_sub_tree">…recursive CTE…</sql>
<!-- automatically selected on oracle: connect by hierarchical query -->
<sql id="dept_sub_tree_oracle">…connect by…</sql>
```

4. Verification: Run Every Target Database Before Go-Live

```
spring.sqltoy.redoDataSources[0]=dmdb
spring.sqltoy.redoDataSources[1]=kingbasedb
```

Once configured, any query is **replayed once on those data sources** during testing, so incompatible constructs surface immediately instead of erroring at a customer site after the switch — see Multi-DB Verification. Two framework-level guarantees complement it:

- Pagination/locking/merge/sequence of every dialect are verified item by item against real databases, with reflection-based exhaustive assertions locking coverage completeness (adding a database type without registering its adaptation fails the test);
- The "loud retention" principle of function conversion means every error during verification is exactly an adaptation-surface item — there are no silent wrong values.

5. Performance: No Regression After Migration — Often a Gain

- **Pagination**: `findPage` enables smart count optimization by default (strips order by, trims select columns); for frequently paged scenarios add `<page-optimize parallel="true" alive-max="100" alive-seconds="120"/>` to cache the count and run it in parallel with the page query; for large-table join pagination use `@fast()` (paginate first, join afterwards) — see Pagination;
- **Cache translate**: code-to-name translation happens in memory, eliminating dictionary-table JOINs; supports incremental updates and FIFO caching for ultra-large master data — see Cache Translate;
- **High-concurrency ledgers**: `updateFetch` / `updateSaveFetch` complete lock-query → check → modify → return in a single database interaction — see Objectified CRUD;
- **Slow-SQL governance**: automatic slow-SQL identification and handling — see Slow SQL; the SQL signature (`/*-id-*/`) makes database-log location a seconds-level task.

6. Cut-Over and Long-Term Operations

- During parallel running, combine with Dynamic Datasources & Transactions to cut over gradually per datasource;
- When a new domestic database appears, onboard it via a `dialectMap` alias first; once an official dialect lands, remove the alias to get native adaptation;
- Keep `redoDataSources` in the regression environment as a permanent gate for cross-DB adaptation.

7. Related Chapters

- Supported Databases · Dialect Self-Adaption & Function Extension · Multi-DB Verification
- Pagination · Cache Translate · Primary Key Strategies · POJO-to-DDL

Technical Architecture

SqlToy is a Java ORM framework centered on **dynamic SQL** and built for enterprise-scale complexity. It integrates with Spring / Spring Boot / Solon and also runs in plain Java. Understanding its internals helps you extend it confidently in complex scenarios.

1. Design Principles

- **SQL separated from code**: SQL lives in `*.sql.xml` files (hot-reloaded in debug mode) and is invoked by `sqlId`; raw SQL in code works too.
- **Object-to-table mapping**: POJO/VO classes carry annotations (`@SqlToyEntity` / `@Entity` / `@Id` / `@Column`) that map them to tables; `EntityManager` manages them centrally.
- **Dialect isolation**: all database differences are contained in the `dialect` package; `DialectFactory` detects the database type and routes to the right implementation, keeping business SQL portable.
- **In-memory algorithms**: pagination optimization, pivot, summary, chain-ratio, tree sorting, cache translation and more run as algorithms in application memory — offloading the database and staying consistent across databases.
- **Three ways to organize a query**: SQL mode (sqlId + `#[...]`, for complex dynamic and configurable queries), `EntityQuery` / `EntityUpdate` (single-table string conditions) and lambda conditions (property method references; one builder for single-table and joins) — all three share the same execution path and can be mixed per scenario.

2. Core Packages

The framework core lives under `org.sagacity.sqltoy` (artifact `sagacity-sqltoy`):

Package / Class	Responsibility
`SqlToyContext`	**Root context of the whole framework**, holding all configuration (data sources, dialect, SQL resources, cache translate, id strategies, interceptors, etc.)
`SqlToyConstants`	Framework constants, loaded from `sqltoy-default.properties`; overridable via `SqlToyContext` dialectConfig
`dao`	Public DAO interfaces: `LightDao` (main entry, ~110 methods), `SqlToyDao`, `SqlToyLazyDao` with `impl` implementations
`support.SqlToyDaoSupport`	Base class implementing all database interactions (the real workhorse)
`service.SqlToyCRUDService`	Generic CRUD service — simple operations need no hand-written Service
`link`	**Chain API**: `Query` / `Save` / `Update` / `Delete` / `Load` / `Store` / `Unique` / `Execute` / `Batch` / `Elastic` / `Mongo` / `TreeTable` / `TableApi`, all extending `BaseLink`
`lambda`	**Lambda condition API** (since 6.0.8): `LambdaQuery` / `LambdaUpdate` / `LambdaDelete` plus `JoinCol` / `ExcludeCols` / `LambdaFn` / `OuterRef`; `engine` (segment merging, parameter numbering, deferred assembly), `join` (join context, table aliases, portable EXISTS rewrite), `condition` (compare/aggregate/logic conditions), `toolkit` (property metadata extraction, serialization mirror)
`model`	Data models: `Page`, `QueryExecutor`, `EntityQuery` / `EntityUpdate`, `ParallelQuery`, `QueryResult`, `StoreResult`, `TreeTableModel`, `Summary`, `ColsChainRatio` / `RowsChainRatio`, `CacheArg`, `TableMeta` / `ColumnMeta`, `LockMode` / `SaveMode` / `MaskType`, etc.
`config`	SQL/entity loading and parsing: `EntityManager`, `EntityScanner`, `SqlScriptLoader` (parses `*.sql.xml`), `SqlXMLConfigParse`, `SqlFileModifyWatcher` (hot reload in debug); `config.annotation` (annotations), `config.model` (parsed config models)
`dialect`	Database dialects: `Dialect` interface + `DialectFactory`; 25 implementations under `impl` (SAP HANA added in 6.0; HighGo added in 6.0.5); `executor` (incl. the parallel-query executor); root-level `PageOptimizeUtils` (pagination/count optimization engine), `QueryExecutorBuilder`, `CrossDbAdapter`
`plugins`	Extension points: `calculator` (summary/chain-ratio/tree-sort/unpivot in-memory algorithms), `datasource` (dynamic datasource selection), `ddl` (POJO → DDL), `function` (28 cross-DB function adapters), `id` (primary-key strategies; `id.macro` implements the `@if/@loop/@include/@case/@df/@substr` macro directives), `interceptors` (tenant filtering), `json` (`JsonProvider`, the pluggable JSON library), `nosql` (Elasticsearch, `MongoElasticOperations`), `overtime` (slow SQL), `secure` (encryption/masking), `sharding`; plus root-level `SqlInterceptor`, `TypeHandler`, `IUnifyFieldsHandler`
`translate`	Cache-translate subsystem: `TranslateManager`, `TranslateFactory`, `TranslateConfigParse` (parses `sqltoy-translate.xml`), `CacheUpdateWatcher`, `DynamicCacheFetch`; `cache` (ehcache / caffeine / FIFO dynamic cache)
`callback`	22 SPI callback interfaces: `RowCallbackHandler`, `StreamResultHandler`, `TransactionHandler`, `UpdateRowHandler`, `EntityUpdateCallback`, `DecryptHandler`, etc.
`integration`	Framework-agnostic SPI: `AppContext`, `ConnectionFactory`, `DistributeIdGenerator`, `MongoQuery`
`utils`	Utility classes: `SqlUtil`, `BeanUtil`, `CollectionUtil`, `DateUtil`, `ResultUtils`, `DBTransUtils`, geometry helpers, etc.

3. Integration Modules

Besides the core, SqlToy ships several framework integrations:

Module	artifactId	Description
Core	`sagacity-sqltoy`	ORM core, usable in plain Java
Traditional Spring	`sagacity-sqltoy-spring`	Transaction-aware DAO implementations, `SpringAppContext` / `SpringConnectionFactory`
Spring Boot	`sagacity-sqltoy-spring-starter`	Auto-configuration (`spring.sqltoy.*`), task pool, GraalVM AOT native support
Solon	`sagacity-sqltoy-solon-plugin`	Solon adaptation (`sqltoy.*` prefix, `@Db` injection)

4. Lifecycle of a Query

Take `lightDao.findPage(page, "sqlId", params, VO.class)` as an example:

1. `LightDao` interface → implemented by the `SqlToyDaoSupport` base class;
2. The `SqlToyConfig` for the `sqlId` is fetched from `SqlToyContext` (pre-parsed by `SqlScriptLoader`);
3. `filters` normalize parameters and `#[]` fragments are trimmed, producing the final SQL and parameter array;

4. `DialectFactory` detects the dialect and builds the page SQL plus the optimized count SQL (`PageOptimizeUtils`);
5. Configured `sharding` /tenant interceptors, `translate` cache translation and `secure` masking are applied in turn;
6. The query executes and the result set goes through in-memory algorithms (summary/pivot/chain-ratio/tree-sort, etc.);
7. Everything is wrapped into a `Page` and returned.

Overall flow:

```
flowchart LR
    A[LightDao interface] --> B[SqlToyDaoSupport<br/>operation impl]
    B --> C[SqlToyConfig<br/>sqlId pre-parsed]
    C --> D[filters normalize params<br/>dynamic SQL fragment trimming]
    D --> E[DialectFactory<br/>dialect routing]
    E --> F[page SQL + count optimize<br/>PageOptimizeUtils]
    F --> G[interception & enhancement<br/>sharding · tenant · translate · secure]
    G --> H[in-memory algorithms<br/>summary · pivot · tree-sort]
    H --> I[wrapped into a Page and returned]
```

> To dig into the source, start from `SqlToyDaoSupport` (operation implementation), `LightDao` (API entry), `DialectFactory` (dialect routing), `SqlToyContext` (configuration hub), `TranslateManager` (cache translate) and `PageOptimizeUtils` (pagination optimization).

Upgrade to 6.0.x

Changes and notes for upgrading from 5.6.x to 6.0.9 (the 6.0 line). All changes have been verified against the 6.0 source code.

1. Environment Requirements

Item	Requirement
JDK	**17+** (targeting Spring Boot 3/4); JDK 8 projects should use `5.6.96.jre8` (the final jre8 version, also works with the quickvo-maven-plugin)
Dependency coordinates	Unchanged: `com.sagframe:sagacity-sqltoy` (plus spring / spring-starter / solon-plugin); only the version number is upgraded to 6.0.9
Configuration parameters	Two parameters are added on top of the 58 `spring.sqltoy.*` parameters of 6.0.2: `realDialectFirst` (new in 6.0.3) and `ddlLowerOrUpper` (exposed as a starter property since 6.0.3); all other parameters are unchanged (parameters added in later versions are covered in their respective sections below and are all optional), and existing configuration files need no adjustment

2. API Changes (Code Changes Required)

Change	5.6.x	6.0.x
Parallel query	`lightDao.parallQuery(List<ParallQuery>, ...)`	`lightDao.parallelQuery(List<ParallelQuery>, ...)` (the model class is renamed accordingly: `ParallQuery` → `ParallelQuery`)
Connection callback	`doConnection(Connection conn, Integer dbType, String dialect)`	`doConnection(Connection conn, DBProfile profile)` — `DBProfile` provides the complete database execution profile

`DBProfile` (`org.sagacity.sqltoy.model.DBProfile`) provides `getDbType()` / `getDialect()` / `getUrl()` / `getProductName()` / `getMajorVersion()`, plus convenience checks such as `isMysqlFamily()` / `isOracleFamily()` / `isOceanBase()` / `isBackslashEscape()`; see Other Features for details.

> The other frequently used APIs (the remaining 71 LightDao method names — 109 declarations in total — the Link chain API, QueryExecutor/EntityQuery/EntityUpdate, and the existing configuration parameters) are **all unchanged**; the upgrade cost is concentrated in the two items above.

3. Package Structure Changes (Adjust imports If You Reference Framework Internal Classes)

Class	5.6.x Location	6.0.x Location
`PageOptimizeUtils`	`dialect.utils`	`dialect`
`ParallelUtils`	`utils`	`dialect.executor`
`QueryExecutorBuilder`	`utils`	`dialect`
`CrossDbAdapter`	`plugins`	`dialect`
`MacroIfLogic`	`utils`	`config`
`ParamFilterUtils`	`utils`	`config` (renamed to `ParamFilterProcessor`)
`TranslateUtils`	`utils`	`translate`
`HttpClientUtils`	`utils`	`plugins.nosql`
`MongoElasticUtils`	`utils`	`plugins.nosql` (renamed to `MongoElasticOperations`)
`SavePKStrategy`	`dialect.model`	`model`

4. New Capabilities

- **SAP HANA dialect**: the total number of dialects reaches 24; see Supported Databases;
- Internal model enhancements: `ExecuteParams`, `ResultColumnMeta`, etc. (used internally by the framework; no impact on existing APIs).

> For more complete version evolution details, refer to `sqltoy-orm-core/changelog.md` in the source repository.

5. What's New in 6.0.3

6.0.3 builds on 6.0.2 with additional capabilities — all incremental, requiring **no changes to existing code**:

SQL dialect system

- The `<sql>` element gains a `dialect` attribute: it declares that the sql is parsed and solidified in the specified dialect form — at load time function/reserved-word conversion is done once against that dialect; at execution time, if the current database dialect matches the declaration, conversion is skipped, otherwise lazy conversion adapts it (`mql` / `eql` do not support it). See Dialect Self-Adaption & Function Extension;
- New `spring.sqltoy.realDialectFirst` configuration (default `false`): when enabled, sqlId dialect variant lookup first follows the **real dialect detected from the connection** (`DBProfile.realDialect` — typical cases: OceanBase configured with the mysql dialect, proxied databases); if the real-dialect variant does not exist, the original configured-dialect lookup chain is used as fallback.

POJO-to-DDL generation (see POJO-to-DDL)

- New `@Partition` / `@PartitionDef` / `@MppTable` entity annotations: describe table partitioning strategies and MPP analytical-database (ClickHouse/Doris/StarRocks) table-engine metadata; quickvo generates them automatically from the database partition/table definitions;
- `PARTITION BY` clauses (RANGE/LIST/HASH/RANGE COLUMNS/LIST COLUMNS, with partition details) are now rendered automatically in create-table statements;
- Dedicated DDL generators added for Doris, StarRocks, ClickHouse, DB2, HANA and SQLite (previously Doris/StarRocks reused the MySQL generator and the rest fell through to the default generator);
- `@Foreign`'s `deleteRestict` / `updateRestict` now support the full set of referential actions (`ReferentialAction`: CASCADE/RESTRICT/SET_NULL/NO_ACTION/SET_DEFAULT), trimmed automatically per database compatibility;
- JSON column types degrade gracefully per database (HANA→NCLOB, SQL Server→NVARCHAR(MAX), DB2/Oracle11→CLOB).

Fixes and optimizations

- Sharded parallel execution: `ShardingResult` gains a `cause` field preserving the original exception, and the exception thrown on global rollback carries the root-cause stack; `@Sharding.maxConcurrents` semantics fixed (0 = unlimited, 1 = serial);
- `executeSql` / `batchUpdateByJdbc` now force a rollback when a manual commit fails, and `autoCommit` restoration moved into `finally`, preventing connections from being returned to the pool with a rewritten commit mode after an error;
- Result mapping: numeric strings with thousands separators produced by `number-format` (e.g. `8,888.0000`) have the grouping commas stripped automatically when mapped back to numeric properties;
- SqlServer pagination: the WITH section is now stripped and placed first, and the union derived-table wrapping order is fixed.

6. What's New in 6.0.4

6.0.4 builds on 6.0.3 with further enhancements — likewise **incremental, requiring no changes to existing code**:

Cache translate REST authentication enhancement (see Cache Translate)

- `rest-translate` / `rest-checker` / `rest-increment-checker` gain the `auth-type` authentication mode: `basic` (default, backward-compatible with the existing `username` / `password`) and `jwt`;
- For jwt, the token is obtained in one of three ways (priority order applies when multiple are configured): `token-url` login endpoint that posts username/password as a form and extracts the token from the JSON response via `token-path` (dot paths like `data.token` supported), then caches it; `token-secure-key` local self-signed jwt (HS256/384/512 shared secrets and RS256/384/512 PKCS#8 private keys, no login endpoint needed); or `authorization` with a directly configured long-lived static token;
- Companion attributes: `token-header` (default `Authorization`), `token-prefix` (default `Bearer`, set to `none` to send the bare token), `token-expire` (default 1800 seconds; when the response contains `expires_in` the response value wins). Tokens are cached and reused, refreshed proactively when close to expiry, and on a 401 the token is invalidated and re-login (or re-signing) plus one retry happens automatically.

Cross-database function adaptation improvements (see Dialect Self-Adaption & Function Extension)

- New source-function recognition: `nvl2(a,b,c)` (non-oracle databases convert to a case-when null-check), single-argument `isnull(x)` (mysql returns 0/1 semantics), `sys_timestamp` (passthrough on oracle-family, fractional-second precision kept on mysql-family, sysdatetime on sqlserver), `strpos(str,sub)` (passthrough on pg-family/clickhouse, converted to instr elsewhere), `substr(s,-n)` with a negative start, and the `curdate()` / `CURRENT_DATE` function class (oracle-family converts to `TRUNC(CURRENT_DATE)`, db2 uses the CURRENT DATE keyword form, sqlite uses the local clock);
- `group_concat` / `listagg` / `string_agg` now support the trailing `within group (order by ...)` clause, an order by embedded in the arguments, and DISTINCT prefix recognition;
- `datediff(month,...)` is unified to the year/month component difference (no longer registered with the oracle MONTHS_BETWEEN decimal semantics); `to_char` conversion to sqlserver is now based on FORMAT, fixing wrong output for uppercase format tokens (e.g. `YYYY-MM-DD HH24:MI:SS`); on sqlite targets `now()` / `CURRENT_DATE` / `CURRENT_TIMESTAMP` uniformly use the local clock.

Fixes

- Fixed silent wrong results of `SubStr` with a negative start that deviated from the mysql contract (pg-family/sqlserver/h2/db2/hana: when the computed start is less than 1 an empty string is now returned uniformly);
- Fixed standalone HH hour tokens in minute-less formats (e.g. `yyyy-MM-dd HH`) being interpreted as 12-hour clock on pg/oracle-family targets, which skewed hour-bucket boundaries.

Companion tooling

- The companion quickvo-maven-plugin is upgraded to **2.0.2**.

7. What's New in 6.0.5

6.0.5 builds on 6.0.4 with further enhancements — likewise **incremental, requiring no changes to existing code**:

New database support (dialect total 24 → 25; see Supported Databases)

- **HighGo**: PG12 kernel + Oracle compatibility layer; the SQL-generation layer reuses the PostgreSQL family (`HighgoDialect extends PostgreSqlDialect`), with differences concentrated on the function side (no `uuid()` / `gen_random_uuid()` but has `sys_guid()`; `last_day` / `add_months` / `trunc` native; no `listagg` but `string_agg` native), carried by the HIGHGO branch of the function adapters; supports both the official driver URL `jdbc:highgo://` and the vanilla pg driver (distinguished via `version()` kernel probing).

Cross-database function adaptation improvements (see Dialect Self-Adaption & Function Extension)

- 11 new function classes (all registered into default): `find_in_set`, `uuid` (unique-id generation), `year/month/day` component extraction, `last_day/eomonth`, `add_months`, `date_add/date_sub/dateadd` interval arithmetic, `left/right`, `lpad/rpad`, `extract/date_part`, `trunc/date_trunc` date truncation, and `mod`;
- Source-spelling expansion for existing functions: `instr` adds `locate/posstr`, `length` adds `character_length/octet_length`, `now` adds `current_timestamp`, `if` adds `iif`, `substr` adds `mid`, `to_date` adds `str_to_date`;
- All verified by real-database probing across multiple databases; forms that cannot be converted safely are kept "loud" (passed through for the target database to error on), avoiding silent wrong values.

Lowercase primary-key insert return (PostgreSQL family)

- The case conversion of the primary-key column name when returning the key value on single-row inserts (`getReturnPrimaryKeyColumnCase`) is expanded from PostgreSQL/PostgreSQL14 only to the entire **PG-folding family**: OpenGauss, GaussDB, MogDB, Vastbase, StarDB, Oscar, Kingbase and HighGo are now lowercased as well. Reason: `getGeneratedKeys` column hints are case-sensitive on PG-folding kernels (e.g. opengauss/vastbase/highgo report "column does not exist" for "ID"/"Id", and kingbase errors on mixed case), so lowercasing is safe for the whole family; the mysql family/tidb/oceanbase/sqlserver/db2/dm/oracle keep DEFAULT (the driver ignores the hint or folds consistently). This can still be overridden per dialect via the `dialectReturnPrimaryColumnCase` configuration.

Other optimizations and fixes

- Fixed bare functions at the very start of a string never being converted (when the whole sql is a function expression, the adaptation regex `\W(funcName)\(` never matches; a space sentinel is added at the entry);
- Completed TO_DATE/TO_TIMESTAMP cast-family handling for HIGHGO/H2/SQLITE.

8. What's New in 6.0.6

6.0.6 builds on 6.0.5 with further enhancements — likewise **incremental, requiring no changes to existing code**:

Virtual-table macro @virtualTable() (see Dialect Self-Adaption & Function Extension)

- Solves the cross-database dilemma of "select constant-row union real-query" SQL (dictionary dropdown fallback rows, fixed report total rows, etc.): oracle/db2/hana require a pseudo-table (`from dual` / `from sysibm.sysdummy1` / `from dummy`), while mysql/pg/sqlserver etc. natively support FROM-less select (writing `from dual` actually errors there);
- Unified form: `select '0' id,'Global' name @virtualTable() union select id,name from t` — expanded per dialect at SQL parse time: oracle family (oracle/oracle11/dm/oceanbase) → `from dual`, db2 → `from sysibm.sysdummy1`, hana → `from dummy`, stripped elsewhere;
- Compatible aliases `@dual()` / `@dummyTable()` — all three names are equivalent, case-insensitive, tolerant of spaces inside the parentheses; may appear multiple times in one SQL (one per constant select);
- Works in xml-configured SQL, `@Sql` annotation SQL and code SQL (`findBySql` etc.) alike (unified parse pipeline); SQL without the macro is completely unaffected (plain SQL containing `@` such as emails is not touched);
- Expansion happens at parse time and is solidified with SqlToyConfig (same timing as function/reserved-word conversion), zero runtime overhead; with `realDialectFirst` the macro is expanded by the parse dialect — the runtime lazy layer does not re-expand it.

Companion tooling

- The companion quickvo-maven-plugin is upgraded to **2.0.3**.

9. What's New in 6.0.7

6.0.7 builds on 6.0.6 with further enhancements — likewise **incremental, requiring no changes to existing code**:

Pluggable JSON library (JsonProvider SPI)

- sqltoy is no longer hard-bound to fastjson2: JSON/JSONB column mapping, the rest/service paths of cache translation, and the whole Elasticsearch stack (including jwt authentication) now delegate to `org.sagacity.sqltoy.plugins.json.JsonProvider`, with three built-in implementations — fastjson2, jackson3 (`tools.jackson`) and jackson2 (`com.fasterxml`);
- Resolution order: an explicit `spring.sqltoy.jsonProvider` wins, otherwise the classpath is probed in the order fastjson2 → jackson3 → jackson2. fastjson2 comes first so that projects whose JSON columns already worked keep behaving exactly as before;
- The configuration value accepts a built-in alias — `fastjson2` / `jackson3` / `jackson2` (case, hyphens and surrounding whitespace insensitive) — or the fully qualified class name of a custom implementation;
- A probed candidate must pass a smoke parse before it is accepted: being on the classpath is not the same as being usable (for instance jackson3's databind needs `jackson-annotations` 2.20+, and when jackson2 is also present an older annotations version can win the conflict resolution and cause `NoSuchFieldError: POJO`). An unusable candidate is logged as a warning and the next one is tried; a missing marker class is skipped silently;
- Initialization is attempted at startup: an illegal or unusable explicit configuration **fails immediately and preserves the root cause** (it does not silently fall back, which would make the configuration meaningless); when no usable JSON library is present it only logs a warning and does not block startup — **projects that use neither JSON columns nor Elasticsearch now need no JSON library at all**, and a clear guided exception is thrown only when a JSON feature is actually touched, replacing the former `NoClassDefFoundError`;
- The jackson2 / jackson3 dependencies in core are both `provided + optional` and are not transitive;
- See Configuration and Elasticsearch.

Field-list joining macro @joinFields() (see Tags & Expressions)

- For page-configurable reports and API services where the configuration decides the `select` / `group by` / `order by` columns. Written as `@joinFields(:fields,default)`, or directly in positional form `@joinFields(?,default)` — it is processed at the same stage as `@value`, so once named parameters have become `?` the two forms are exactly equivalent; the consumed parameter is removed from the parameter set, leaving the remaining `?` placeholders aligned;
- When `fields` is null, empty or all-blank, `default` is emitted. `default` comes from the SQL text (author-written, trusted) and is emitted verbatim without identifier validation, so it may be `t.*` or `sum(t.amount)` ; wrap it with `{}` , `''` or `""` when it contains commas or parentheses. Without a `default` the result is an empty string;
- Input normalization: a single string or a 1-element array is split on commas (page multi-selects often serialize to `["a,b,c"]`); with more than one element the elements are the fields and no further splitting occurs, so `["a,b","c"]` fails on `"a,b"` instead of being silently split into three fields; blank fragments are skipped to tolerate trailing commas;
- Security: field names go through a character whitelist (letters including Chinese, digits, underscore, at most two dots for `schema.table.column`). `$` and `#` are explicitly excluded (they collide with the `#[]` dynamic segments and the `${}` / `#{}` placeholders), as are whitespace, quotes, brackets, parentheses, semicolons, commas, comment markers and backslashes. An illegal value throws immediately and names the offending value — it is never silently sanitized (the DAO layer wraps it into a `DataAccessException`, so inspect the root cause when catching);
- Legal fields are quoted per dialect **only when reserved words** (backticks for the mysql family, double quotes for oracle/pg, square brackets for sqlserver); non-reserved identifiers stay bare, avoiding the case sensitivity that quoting introduces on oracle/pg. Dotted names are handled segment by segment, so the column part of `t.order` is quoted too;
- **Prerequisite**: reserved-word quoting depends on the `spring.sqltoy.reservedWords` configuration, which has **no default**. Without it the reserved-word set is empty and `convertWord` returns names unchanged, so a field list containing a reserved word (`order` / `desc` / `level` / `group` ...) produces a bare identifier that the database rejects — verified to raise a syntax error on PostgreSQL, MySQL 9 and SQL Server 2025;
- Plain identifiers only: expressions such as `sum(col)` or `col as alias` still belong to `@value` (safety then on you); operator forms (`a % b` , `a || b`) are not function calls and are likewise out of scope;
- Verified end to end against three real databases — PostgreSQL, MySQL 9 and SQL Server 2025 (plain columns, default fallback, group-by aggregation, reserved-word columns, injection rejection).

Dangling and/or connectors inside brackets are cleaned up after `#[]` removal

- For a construct like `#[and ( #[t.a=:a] #[or t.b=:b] )]` , when `a` is null the previous behaviour expanded to the broken `and ( or t.b=:b )` ; after removal, a trimmed leading/trailing `and` / `or` inside brackets is now cleaned up, so each member of a bracket group is independently optional (`and|or` only matches when followed by `\s+` or `)` , so keywords such as `( order by ...)` that start with `or` are left alone). See Dynamic SQL Guide.

10. What's New in 6.0.8

6.0.8 is the **heaviest release in the 6.0 line**: the lambda condition API moved into core and the GraalVM AOT support was rewritten. **Projects running on the JVM need no code changes**; projects using sqltoy-plus should follow item 2; projects building native images should check their build environment against item 4.

1. The lambda condition API is now part of core (see Lambda Conditions)

- New package `org.sagacity.sqltoy.lambda`: `LambdaQuery` / `LambdaUpdate` / `LambdaDelete` plus `JoinCol` / `ExcludeCols` / `LambdaFn` / `OuterRef`, expressing conditions through entity property method references (`User::getStatus`);
- `LightDao` gains three default factories `lambdaQuery/lambdaUpdate/lambdaDelete(Class)`; execution is driven by the builder terminals (`find/getOne/findPage/count/exists/update/delete`), or build first and execute later with `LambdaQuery.create(entityClass).bind(lightDao)`;
- **A single-table query is just the degenerate form of a join**: without a join step the single-table channel converts conditions into `EntityQuery` / `EntityUpdate` and reuses `findEntity` / `findPageEntity` / `getCount` / `deleteByQuery` / `updateByQuery`, so sharding / cache translate / encryption-masking / tenant filtering are all inherited; calling `leftJoin/rightJoin/innerJoin` + `on` switches to the join channel, which assembles full SQL and goes through `find` / `findPage` / `getCount`, supporting `select/selectAll(ExcludeCols...)/JoinCol.of|avg|count|sum|min|max|distinct().as()` as well as CTE / `@fast` pushdown / UNION;
- Behavioural differences to watch: `getOne()` now has **strict semantics** — 0 rows returns null, more than 1 row throws `DataAccessException` (it no longer silently takes the first row); a conditional `update/delete` with no where clause throws by default and requires an explicit `matchAll()`;
- Condition parameters are renamed from sqltoy-plus's `type.valuePairs.MPG1` form to `:p1…:pN`, unique across the whole builder and shared across nesting — **unit tests that assert SQL text must be updated**; external parameters of `having` / `inSql` should avoid the `p` + pure-digit form;
- Value filtering changes from sqltoy-plus's global static strategy to a **builder-instance** `valueFilter(...)` (nulls are filtered by default; set it before adding conditions);
- `LambdaUpdate.setExpr(field, expression[, value])` supports expression updates based on the field's current value (e.g. `setExpr(User::getLoginCount, "loginCount + 1")`), reusing the existing "field = expression" form of `updateByQuery`.

2. Migration checklist for sqltoy-plus users

Migration point	Action
Dependency	Remove the `sqltoy-plus` dependency (the capability is in core); `@EnableSqlToyPlus` and the mapping-strategy beans are no longer needed
Imports	`com.sagframe.sagacity.sqltoy.plus.*` → `org.sagacity.sqltoy.lambda.*`
Class names	`LambdaQueryWrapper` / `LambdaUpdateWrapper` → `LambdaQuery` / `LambdaUpdate`; `LambdaColumn` → `JoinCol`; `SFunction` → `LambdaFn`; `FiledNature` → `record FieldNature(entityClass, fieldName)`
DAO	`SqlToyPlusDao.findList/findOne/findPage/count/exists/delete/update(wrapper)` → builder terminals; `SqlToyHelperDao` has been deleted
Entry points	`Wrappers.lambdaQuery(cls)` → `LambdaQuery.create(cls)` or `lightDao.lambdaQuery(cls)`; `lambdaQueryChain/lambdaUpdateChain` → `lightDao.lambdaQuery/lambdaUpdate`
Multi-table	`Wrappers.lambdaMultiWrapper(toClass).select(...).from(...)` → `lightDao.lambdaQuery(toClass)[.select(JoinCol...)].from(...).leftJoin(...).on(...)`; `lambdaUpdateMultiWrapper/lambdaDeleteMultiWrapper` → `lightDao.lambdaUpdate/lambdaDelete` plus join steps
Parameter names	`type.valuePairs.MPG1` → `:p1…`
String wrappers	`QueryWrapper` and friends are no longer provided: use `EntityQuery` for single-table string conditions and SQL mode for complex dynamic SQL

> For the full field-by-field table see Lambda Conditions · 9. Migrating from sqltoy-plus.

3. Multi-table join update and delete

Join updates and deletes are folded into `lambdaUpdate()/lambdaDelete()` (the same "no join means single table" rule as for queries): a join update generates `UPDATE main alias JOIN ... SET alias.column=:p WHERE ...`, a join delete generates `DELETE alias[,...] FROM main alias JOIN ... WHERE ...` (`deleteTables` defaults to the main table), and both run through `executeSql`. Relational databases use a portable EXISTS-over-derived rewrite, verified on real mysql/postgresql/sqlserver/oracle/db2 databases; clickhouse/doris/starrocks and other databases without transactional row-level DML do not support it. **Merging a whole entity into an update is not supported on the join channel** — use `lightDao.update(entity)`.

4. GraalVM native / AOT rewritten (see GraalVM Native Image Guide)

- Fixed the defect, present since 5.6, that **AOT entity reflection hints were always empty**: during AOT Spring registers no `BeanPostProcessor` and pre-instantiates no singletons, so the `SqlToyContextProperties` obtained from `getBean` was an unbound empty instance and every entity hint was lost. The processor now binds `spring.sqltoy` straight from the `Environment` through `Binder` and scans entities with `EntityScanner` according to `packagesToScan` / `annotatedClasses`. `packagesToScan` **is mandatory in native scenarios**;
- Registers **dynamically from the configuration**: resource patterns (`sqlResourcesDir` with multiple directories / `sqlResources` / `translateConfig` / key files), reflection for 15 extension points (`unifyFieldsHandler`, `typeHandler`, `sqlInterceptors`, `sqlFormater`, `translateCacheManager`, `dataSourceSelector`, `connectionFactory`, `fieldsSecureProvider`, `desensitizeProvider`, `customFilterHandler`, `overTimeSqlHandler`, `dialectDDLGenerator`, `firstBizCodeTrace`, `dynamicCacheFetch`, `dynamicFecthCacheManager`), the `functionConverts` function classes resolved by

`FunctionUtils.resolveFunctionClassNames`, lambda's `SerializedLambda`/mirror-class serialization, and the 8 model classes populated reflectively by XML parsing (without them `<unpivot>`/chain-relative/`<tree-sort>` tag attributes are **silently dropped**);

- Registers **conditionally on the classpath** (a native image is frozen at build time, so nothing can be switched at runtime): the caffeine/ehcache manager for `cacheType`, the mongo/redis integration classes, the three JSON providers, the five driver-isomorphic `PGobject` classes for the PG family (postgresql/opengauss/vastbase/gaussdb/kingbase), mssql `Geometry`/`Vector`, and `oracle.jdbc.OracleTypes`;
- New settings `native-reflection-classes` / `native-reflection-packages` (for non-entity DTO/VOs) and `native-entity-proxies` (pre-generates `updateSaveFetch` proxies at build time; the application must add `net.bytebuddy:byte-buddy`);
- The four `@Bean`s `sqlToyLazyDao`/`sqlToyDao`/`lightDao`/`sqlToyCRUDService` in the starter changed their return types from interfaces to **concrete implementation classes**, with an explicit `@ConditionalOnMissingBean(interface.class)`: Spring AOT infers bean types from the declared type, and an interface declaration prevents the transactional CGLIB proxy from being pre-generated at build time, which fails at native runtime with "CGLIB runtime enhancement not supported on native image". Application-side `@Bean`s declared with an interface return type plus `initMethod` should likewise be switched to concrete types;
- **The build environment requires GraalVM 25+** (verified matrix: 17 and 23 do not understand the new reachability-metadata schema — native-maven-plugin rejects them outright; 21 does not support `typeReached` conditional metadata, so spring-aop's AspectJ serialization entries get registered wrongly; 25.0.4 verified with the full matrix on both stacks);
- Known limitation: the `EntityUpdateCallback` path of `updateSaveFetch` relies on byte-buddy generating entity subclasses at runtime; under native it needs `native-entity-proxies` to pre-generate the proxy, and entities not listed there remain unavailable (the insert branch is unaffected).

5. Companion verification project

Added sqltoy-aot-verify (one matrix shared by the spring and solon stacks, PostgreSQL pgvector + postgis): it covers CRUD × json/vector/geometry in three shapes, `functionConverts` dialect conversion, cache translate on ehcache and caffeine, the `updateSaveFetch` proxy path, and lambda single-table/join/setExpr/matchAll-guard/delete; the JVM matrix is 17/17 PASS on both stacks and, with `native-entity-proxies` configured, the native smoke run is 17 PASS + 0 KNOWN-LIMITATION.

11. What's New in 6.0.9

6.0.9 is mostly a batch of cross-database empirically verified fixes with no breaking changes — upgrading only requires the version bump.

1. CTE materialization hint kept/stripped by target database

`WITH xx AS [NOT] MATERIALIZED (..)` is only natively supported by the PostgreSQL family (including openGauss, Kingbase, Vastbase, GAUSSDB and HighGo); on other databases (mysql, sqlserver, oracle, db2, ...) the framework strips the hint at parse time — only the materialization optimization is lost, the query semantics are unchanged, so you no longer need one sqlId per dialect. The rendering gate of `.with(name, q, Boolean.TRUE/FALSE)` on the lambda side shares the same source (see Lambda Queries and Updates).

2. Function adaptation fixes verified on real databases

- The bare `HH` token of `to_char`/`to_date`/`str_to_date` is unified to the **24-hour** convention across databases (lowercase `hh` is 12-hour, `HH24` is always 24) — on DB2 and H2 it previously rendered as 12 hours, silently corrupting values (verified on real databases);
- When `to_date` targets the pg family / HANA / ClickHouse and the format contains time tokens, it is rewritten to `to_timestamp`/`toDateTime` to avoid the time part being silently truncated (pg's `to_date` only returns a date, verified on a real database);
- `trim(from x)` (the form omitting the trim character) raises error 1064 on mysql 9.3 and ORA-30000 on oracle 23ai; it is now rewritten to the equivalent `trim(x)`/`ltrim`/`rtrim`;
- `substr` start position 0 is unified as "from the beginning"; `mid` targeting sqlite is converted to `substr` (sqlite has no native `mid`); mysql's `left/right(str,0)` and negative lengths align on "return an empty string";
- `instr` supports the four-argument form `instr(str,sub,pos,occurrence)`;
- `find_in_set` comparisons extend to any position convention: `=1/=2/>1/<3` are rewritten as integer-position expressions (the existing `>0/=0` boolean family is unchanged);
- `datediff` for oceanbase joins the oracle branch (OB oracle-mode tenants have no datediff; mysql mode uses `dialect=mysql` by convention).

3. Pagination count cache isolated per dialect

The pageOptimize total-count cache key is upgraded from sqlId to `sqlId@dialect`: when the same sql runs on different dialect databases via `redoDataSources`/multiple datasources, the count results are cached separately instead of cross-contaminating; `PageOptimizeUtils.remove(sqlId)` also clears all dialect caches of that sql. The accompanying fix prevents the cross-db adapter from rewriting the shared `SqlToyConfig` instance when a custom countSql is present, which polluted subsequent queries of the same sqlId.

4. Lambda condition refinements

- Added `likeExact/likeExactLeft/likeExactRight`: `%` inside the value is first escaped as a literal and then padded by direction — user input containing `%` matches as an ordinary character instead of being treated as a wildcard that widens matches (`like/likeLeft/likeRight` semantics unchanged);
- `setExpr` fails fast when a quoted literal `?` appears inside the expression (the update channel replaces `?` textually, so a quoted `?` would be mistaken for a parameter slot);

- Empty groupBy columns and empty having conditions are skipped during assembly (previously illegal `GROUP BY` / `HAVING` fragments could be produced).

5. Others

- SqlScriptLoader initialization/destruction thread-safety hardening (flags made volatile, methods synchronized).

12. Recommended Upgrade Steps

1. Upgrade the dependency version numbers uniformly to 6.0.9 (starter / spring / solon-plugin use the same version as core);
2. Search globally for `parallQuery` / `ParallQuery` and change them to `parallelQuery` / `ParallelQuery` (an IDE rename is enough);
3. If you have custom `DataSourceCallbackHandler` implementations, adjust the `doConnection` signature to take a `DBProfile` parameter;
4. If you reference the framework internal utility classes listed in the table above, adjust the imports to their new package locations;
5. **Projects using sqltoy-plus**: migrate to `org.sagacity.sqltoy.lambda` following the checklist in section 10.2, remove the sqltoy-plus dependency and `@EnableSqlToyPlus`, and regression-test the two behaviour changes (`getOne()` with multiple matches, unconditional update/delete);
6. **Projects building native images**: confirm GraalVM 25+, configure `packagesToScan`, register `native-reflection-classes/packages` as needed, and add entities that use `updateSaveFetch` to `native-entity-proxies`; on GitHub Actions set `distribution: graalvm` in `actions/setup-java` (`graalvm-community` was removed by setup-java v4, see the GraalVM Native Image guide);
7. Startup verification: watch the SQL output in debug mode and run business regression tests (prioritize cache translate, pagination, parallel query and lambda conditions); unit tests that assert SQL text must be updated to the `:p1…` parameter names.

Migrating from MyBatis / MyBatis-Plus

SqlToy shares MyBatis's **dynamic-SQL mindset** (SQL in xml, conditions assembled on demand); what differs is how conditions are expressed and how strong the object operations are. Migration is normally **incremental**: let new queries go through sqltoy first, then convert the old mappers.

1. Concept mapping

MyBatis / MyBatis-Plus	sqltoy	Notes
`<select id="findXxx">` in a mapper XML	`<sql id="findXxx">` in `*.sql.xml`	Files are organized by module and hot-reload in debug mode
`<if test="x!=null"> and t.a=#{x} </if>`	`#[and t.a=:x]`	A null parameter removes the fragment automatically, and **the SQL can be pasted straight into a client**
`<where>` / `<trim prefixOverrides>`	`where 1=1` + `#[]`	No `<where>` wrapper needed
`#{}`	`:name` named parameter	Both end up as `?` through `PreparedStatement`
`${}` (string splicing)	`@value(:name)` / `@joinFields(:cols,default)`	Identifier scenarios get whitelist validation — see Tags & Expressions
`<foreach collection="ids">`	`t.id in (:ids)`	Arrays/collections expand automatically and split above 1000 values
ResultMap + `<result column="A" property="a">`	Result type (VO/POJO) + `AS a` aliases	Use `@SqlToyFieldAlias` when names differ
`<collection>` nested mapping	Hierarchy packaging (hiberarchy) or a flat VO	One join SQL packs parent-child objects — see Query API
Pagination plugins (PageHelper etc.)	`findPage`	Automatic count optimization + `@fast` + parallel pagination
`@Select` annotation	`@Sql` annotation / SQL in code	Complex SQL still belongs in xml
MP `QueryWrapper` / `LambdaQueryWrapper`	`EntityQuery` / `LambdaQuery`	The former takes string conditions, the latter property references
MP `BaseMapper` / `IService`	`LightDao` / `SqlToyCRUDService`	`LightDao` has ~110 methods — see API Cheat Sheet
MP `@TableLogic` logical delete	Shared SQL fragment / interceptor	sqltoy has no built-in logical delete; use a common `<sql>` fragment or a `SqlInterceptor`
MP `@Version` optimistic lock	`@DataVersion`	See Data Versioning
MP `@TableField(fill=...)`	`unifyFieldsHandler`	Unified gap-filling — see Common-Field Auto-Fill

2. Suggested rollout order

1. **Dependencies and configuration**: add `sagacity-sqltoy-spring-starter`, configure `sqlResourcesDir` + `debug=true` — see Environment & Dependencies;
2. **Create the SQL directory**: one `*.sql.xml` per module (e.g. `com/acme/crm/modules/staff.sql.xml`); write **new requirements** in sqltoy form first and leave the old mappers alone;
3. **Generate POJOs**: use quickvo against the existing tables (fully annotated — do not hand-write them);
4. **Convert queries** (best payoff, lowest risk): start with the query that has the most conditions and is hardest to maintain, following section 3;
5. **Convert writes**: replace "load, modify, update" with elastic updates; high-concurrency ledgers become `updateSaveFetch` (select-or-insert-and-modify);
6. **Run the old and new side by side**: compare result sets for a while. `redoDataSources` can replay the new SQL on another database for verification — see Multi-DB Verification.

3. Typical rewrites

3.1 Multi-condition dynamic query

```
<!-- MyBatis -->
<select id="findOrder">
    select * from sqltoy_order_info t
    <where>
        <if test="beginDate!=null"> and t.trans_date &gt;= #{beginDate} </if>
        <if test="endDate!=null"> and t.trans_date &lt;= #{endDate} </if>
        <if test="status!=null and status!=''"> and t.status=#{status} </if>
        <if test="orderId!=null and orderId!=''"> and t.order_id=#{orderId} </if>
    </where>
</select>
```

```
<!-- sqltoy: conditions do not fragment the SQL, which stays runnable in a client -->
<sql id="findOrder">
    <value><![CDATA[
        select * from sqltoy_order_info t
        where 1=1
        #[and t.trans_date>=:beginDate]
        #[and t.trans_date<=:endDate]
        #[and t.status=:status]
        #[and t.order_id=:orderId]
    ]]></value>
</sql>
```

"An empty string counts as not passed" is handled once by a filter rather than per condition:

```
<filters>
    <!-- blank strings become null, which removes the matching #[] fragment -->
    <blank params="status,orderId" />
</filters>
```

3.2 LIKE queries

```
<!-- MyBatis: concat('%',#{name},'%') -->
<!-- sqltoy: the framework adds the wildcards -->
#[and t.staff_name like :staffName]
<filters>
    <r-like params="staffName" />   <!-- appends %: starts with name -->
</filters>
```

3.3 IN collections and empty collections

```
#[and t.organ_id in (:organIds)]
```

- Passing a `List`/array expands to several `?`;
- **An empty collection skips the condition** (never produces `in ()`);
- More than 1000 values are split automatically (the Oracle limit).

3.4 Pagination

```
// MyBatis-Plus
Page<OrderInfo> page = orderMapper.selectPage(new Page<>(pageNo, pageSize), wrapper);
```

```
// sqltoy
Page<OrderInfoVO> page = lightDao.findPage(new Page(pageSize, pageNo),
        "crm_order_findlist", paramsMap, OrderInfoVO.class);
```

The count SQL is rewritten by the framework (order by removed, select list trimmed); for large-table join pagination use `@fast()` — see Pagination Optimize.

3.5 Single-table conditional update

```
// MyBatis-Plus
new LambdaUpdateWrapper<StaffInfo>()
        .set(StaffInfo::getStatus, 2)
        .eq(StaffInfo::getStaffId, id)
```

```
// sqltoy: use lambda for single tables (reuses the entity path, so sharding/translate/tenant apply)
lightDao.lambdaUpdate(StaffInfo.class)
        .set(StaffInfo::getStatus, 2)
        .eq(StaffInfo::getStaffId, id)
        .update();

// or with string conditions
lightDao.updateByQuery(StaffInfo.class,
        EntityUpdate.create().set("status", 2).where("staff_id=?").values(id));
```

3.6 Load-then-update → elastic update / upsert (updateSaveFetch)

```
// The usual MyBatis shape: load, decide, update (two interactions, and it can overwrite a concurrent change)
OrderInfo o = mapper.selectById(id);
o.setStatus(2);
mapper.updateById(o);
```

```
// sqltoy elastic update: update itself is elastic — one interaction, null fields are skipped
// (you never wipe a field someone else changed, no hand-written update set fragment)
OrderInfo orderInfo = new OrderInfo();
orderInfo.setOrderId(id);
orderInfo.setStatus(2);
lightDao.update(orderInfo);

// High-concurrency "select-or-insert-and-modify": one interaction — lock-query, update if exists
// or insert if not, return the result. uniqueProps names the business unique-key properties
// (falls back to the primary key when omitted). Inside the callback the entity is a proxy:
// get reads the current DB value, set writes back to the current row; a thrown exception rolls back.
OrderInfo entity = new OrderInfo();
entity.setOrderId(id);
entity.setStatus(2);
OrderInfo order = lightDao.updateSaveFetch(
        entity,
        (loaded, rowIndex) -> {
            if (!"1".equals(loaded.getStatus())) {
                throw new IllegalStateException("status already changed, please refresh and retry");
            }
            loaded.setStatus(2);
        },
        "orderId");
```

> If the record is **guaranteed to exist** and you need freely-defined query conditions and locking, use `updateFetch` (sql + `UpdateRowHandler`, see Object-Oriented CRUD §10); for ledger-style "update if exists, insert if not", prefer `updateSaveFetch`.

3.7 Batch insert

```
// MyBatis: foreach over values (the SQL length explodes with the data volume)
// sqltoy: batch APIs that split automatically (batchSize defaults to 200)
lightDao.saveAll(list);
lightDao.saveAllIgnoreExist(list);   // skip existing rows
```

4. Common pitfalls

Symptom	Cause and fix
A parameter has no effect	Named parameters are **case-sensitive**, and the name inside `#[]` must match the Map key exactly
Every result property is null	Column and property names differ: add `AS` aliases, configure `humpMapResultTypeLabel`, or use `@SqlToyFieldAlias`
`in` raises a syntax error	String splicing was used instead of `:ids`; hand empty collections to the framework
The paged total is smaller than the data	`@fast` plus an outer join filter makes the count use the inner SQL; supply `<count-sql>` explicitly — see FAQ 10
Dynamic SQL needs an identifier	Never splice user input with `@value`; use `@joinFields` (whitelist) or `<valid-sqlInjection>`
Logical-delete conditions everywhere	There is no built-in logical delete; inject it once with an `@include`d `<sql>` fragment or a `SqlInterceptor`
Transaction behaviour differs	Object operations still honour Spring `@Transactional`; for multiple datasources see Multi-Datasource & Transactions

5. What you gain

- **Querying**: native-shaped SQL (paste it into a client), smart counts, `@fast`, parallel pagination, join-free cache translate;
- **Writing**: elastic update / `updateFetch` / `updateSaveFetch` collapse "several interactions plus manual locking" into one call;
- **Cross-database**: 25 dialects with automatic function replacement, so one SQL runs on many databases;
- **Analytics**: pivot/unpivot, YoY/MoM and tree rollups happen in memory, so you no longer write a different SQL per database.

> Related: Best Picks (side-by-side sqltoy vs MyBatis), Feature List, API Cheat Sheet

Environment Setup

Environment & Dependencies

1. Environment Requirements

Item	Requirement
JDK	**17+** (the 6.0 line requires JDK 17 and targets Spring Boot 3/4); for JDK 8 projects use `5.6.96.jre8` (the final jre8 version, also works with the quickvo-maven-plugin)
Build tool	Maven 3.6+ or Gradle
Database	Any JDBC database (MySQL/Oracle/PostgreSQL/Dameng/GaussDB/SQL Server/DB2, etc.; see Supported Databases)
Connection pool	Druid, HikariCP or any other connection pool (sqltoy is not bound to a connection pool)

> [!NOTE] For version numbers, refer to the latest release on Maven Central; this article uses `6.0.9` as an example.

2. Add Dependencies

Choose the corresponding artifact according to the framework you use (the groupId is always `com.sagframe`):

Spring Boot (recommended, auto-configuration)

```
<dependency>
    <groupId>com.sagframe</groupId>
    <artifactId>sagacity-sqltoy-spring-starter</artifactId>
    <version>6.0.9</version>
</dependency>
```

Traditional Spring (XML/Java configuration)

```
<dependency>
    <groupId>com.sagframe</groupId>
    <artifactId>sagacity-sqltoy-spring</artifactId>
    <version>6.0.9</version>
</dependency>
```

Solon

```
<dependency>
    <groupId>com.sagframe</groupId>
    <artifactId>sagacity-sqltoy-solon-plugin</artifactId>
    <version>6.0.9</version>
</dependency>
```

Pure Java: just include the core package `sagacity-sqltoy`; see How to Use sqltoy in a Pure Java Project.

3. Connection Pool and Datasource

- The demo project uses the **HikariCP** connection pool (running on an in-memory H2 database, see Quick Start), but sqltoy does not mandate a connection pool — choose one according to your actual situation.
- Spring Boot's default connection pool is **HikariCP**; `spring-boot-starter-jdbc` must be included for the datasource to be initialized automatically.
- Make sure to **configure the dataSource and connection pool first**, and then configure sqltoy.

4. Cache Translate Dependencies (Optional)

If you use the cache translate feature, a cache implementation is required. sqltoy supports **ehcache** by default, and **caffeine** can also be selected:

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.12.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

> Why not use Redis as the translate cache by default? Because cache translate pursues high-frequency code-value conversion with **zero local network overhead**, and local caches (ehcache/caffeine) perform better. For very large master data scenarios, use the FIFO dynamic cache.

5. Next Steps

- For the complete flow of creating a project → creating tables → generating POJOs → writing queries, see helloworld Quick Start.
- POJOs/VOs are generated automatically from database tables by the **quickvo** tool; see the quickvo Code Generator.

quickvo Code Generator

sqltoy's POJO/VO classes do **not rely on runtime reflection guesswork**; they are generated by the **quickvo** tool, which connects to the database and produces annotated entity classes based on the table schema. This guarantees an exact correspondence between objects and tables, and gives you full property code-completion when writing SQL.

- Download (releases): https://gitee.com/sagacity/sagacity-quickvo/releases
- For a complete configuration example, see the quickvo configuration in the `tools` directory of the `sqltoy-quickstart` project.

1. How It Works

quickvo is an executable jar (`sagacity-quickvo.jar`) launched via `quickvo.bat` (Windows) or `quickvo.sh` (Linux/Mac):

```
:: core statement of quickvo.bat
java -cp ./libs/* org.sagacity.quickvo.QuickVOStart quickvo.xml
```

- Put the database driver jar into the `libs` directory; quickvo loads the driver automatically;
- quickvo reads `quickvo.xml` , connects to the database, and generates the VO/POJO classes using freemarker templates;
- When writing `quickvo.sh` on Linux/Mac, pay attention to path separators (backslash → forward slash).

2. Key quickvo.xml Configuration

> For the schema location of `quickvo.xml` and IDE completion setup, see the XML Schema Guide.

`quickvo.xml` mainly consists of `property` (parameters), `datasource` (database connection), and `tasks/task` (generation tasks). The structure below is for standalone execution (for the full configuration when using the Maven plugin, see quickvo-maven-plugin):

```
<?xml version="1.0" encoding="UTF-8"?>
<quickvo xmlns="http://www.sagframe.com/schema/quickvo">
    <!-- property: global parameters or a parameter file, referenced later via ${} -->
    <property file="src/main/resources/application.yml" />
    <property name="project.package" value="com.demo" />

    <!-- datasource: database connection (values can be written directly or referenced via ${} from property) -->
    <datasource name="demo" url="jdbc:mysql://localhost:3306/demo"
                driver="com.mysql.cj.jdbc.Driver" schema="demo"
                username="root" password="***" />

    <!-- tasks: dist output path; task matches tables with the include regex; entity generates pojo, vo generates dto/vo -->
    <tasks dist="src/main/java" encoding="UTF-8">
        <task datasource="demo" author="dev" include="^SQLTOY_\w+" active="true">
            <entity package="${project.package}.entity" substr="Sqltoy" name="#{subName}" lombok-chain="true" />
            <vo package="${project.package}.dto" substr="Sqltoy" name="#{subName}VO" lombok-chain="true" />
        </task>
    </tasks>

    <!-- also configurable: primary-key (primary key strategy), cascade (parent-child cascade), type-mapping (type mapping), swagger (v3/v2/false) --
>
</quickvo>
```

Key configuration:

Configuration	Description
`property`	Global parameters or a parameter file (`file` imports an external config; `name` / `value` define parameters)
`datasource`	Database connection; **for multi-instance databases always configure `schema` or `catalog` correctly** (see the notes below)
`tasks` / `task`	`dist` output path and `encoding` charset; `task` matches table names with the `include` regex and is toggled by `active` ; multiple tasks may generate into different packages
`entity` / `vo`	Generate POJO and DTO/VO respectively; `package` is the target package, `substr` strips the table-name prefix, `name` is the class-name template (`#{subName}`), `lombok-chain` generates chained setters
`swagger`	Generate swagger annotations: `v3` / `v2` / `false` (do not generate)
`field.support.linked.set`	Switch: whether the set methods in generated VOs return the object itself (chainable)
`primary-key`	Primary key strategy for tables; identity-type keys need no configuration. quickvo sets the strategy automatically based on the primary key type/length: `bigint` → snowflake, 22-character → DefaultIdGenerator, 26-digit → NanoTimeIdGenerator
`cascade`	Cascade operation configuration: load cascade (e.g. load only valid child tables), update cascade (delete needs no configuration — child rows are removed directly via the primary/foreign key), and `orderBy` (ordering when loading child tables, e.g. `showIndex desc`)
`type-mapping`	Field type mapping; `import-types` can additionally specify import types for the VO. If a type cannot be matched, quickvo prints a hint in the log — use it to supplement type-mapping

> Since quickvo 5.0.1, entity supports **lombok** as well as custom api-doc frameworks.

3. Generated Classes and Annotations

Each table generates **two classes, AbstractVO + VO**, where the VO extends AbstractVO. sqltoy parses the VO through annotations:

- `@Entity(tableName="sqltoy_dict_detail", pk_constraint="PRIMARY")` : corresponds to the concrete table (`org.sagacity.sqltoy.config.annotation.Entity`);
- `@SqlToyEntity` : marks the precise anchor when resolving a VO — after finding `@SqlToyEntity` , sqltoy looks at its parent class to locate the class that actually maps to the table, then collects `@Id` and `@Column` to build the complete object-to-table mapping;
- `@Id` , `@Column` : primary key and column mapping;
- `@PartitionKey` : partition column marker (the partition key hint for MPP table creation and DML); partitioned tables additionally get `@Partition` / `@PartitionDef` (partition strategy and partition details) on the entity, and ClickHouse/Doris/StarRocks tables get `@MppTable` (table engine metadata) — these three work with sqltoy 6.0.4's POJO-to-DDL feature (see POJO to DDL); with older quickvo versions you can annotate them by hand.

quickvo also writes the database table/column comments into the VO, which makes it easier to read and maintain.

> [!WARNING] Do not modify anything inside the "marked section" of a VO (it will be overwritten on regeneration). To add extra properties or get/set methods, place them **outside** that section.

4. Key Caveats

Duplicated or mixed-up fields on multi-instance databases: when tables with the same name (e.g. `SYS_STAFF_INFO`) exist in multiple instance databases, the generated VO may contain duplicated fields or fields that belong to another database.

Solution: **configure the datasource's `schema` or `catalog` correctly** (configuring `schema` is usually enough). Both table metadata and column metadata retrieval depend on this setting; a wrong value causes quickvo to fetch tables/columns that do not belong to the current database.

5. Strict VO (DTO) / POJO (entity) Layering

For medium and large projects we recommend strictly separating the VO/DTO used for external interaction from the persistence-layer POJO, generating them into different packages via quickvo configuration, and converting between VO and POJO in the business layer (see Utility classes: DTO ↔ POJO conversion).

- Layered example project: https://github.com/sagframe/sqltoy-strict

6. See Also

- To integrate quickvo into the Maven build, see quickvo-maven-plugin.

- For primary key strategy details, see Primary Key Strategies.

quickvo-maven-plugin

Besides running quickvo standalone (see quickvo Code Generator), the recommended approach is **quickvo-maven-plugin**, which integrates code generation into the Maven build — a single command generates/updates the POJO and VO classes from database tables.

> [!NOTE] The current version is **2.0.4**; the 2.0.0 release was a major refactoring of previous releases and has been tested against 19 databases (Oracle, DB2, SQL Server, MySQL, PostgreSQL, HANA, Kingbase, DM, openGauss, OceanBase, Doris, TiDB, StarRocks, ClickHouse, etc.). See the plugin repository README for the full feature list and configuration.

- Project home: https://gitee.com/sagacity/maven-quickvo-plugin (see its README for the full configuration)

1. Configure the Plugin in pom.xml

```
<plugin>
    <groupId>com.sagframe</groupId>
    <artifactId>quickvo-maven-plugin</artifactId>
    <version>2.0.4</version>
    <configuration>
        <configFile>./src/main/resources/quickvo.xml</configFile>
        <baseDir>${project.basedir}</baseDir>
    </configuration>
    <dependencies>
        <!-- add the driver for your database -->
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>${mysql.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

2. Write quickvo.xml

Create `quickvo.xml` under `src/main/resources`. It can directly reuse the datasource configuration from the project's `application.yml` (imported via `<property file>` and referenced via `${}`):

```
<?xml version="1.0" encoding="UTF-8"?>
<quickvo xmlns="http://www.sagframe.com/schema/quickvo"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.sagframe.com/schema/quickvo https://sagframe.github.io/schema/quickvo.xsd">
    <!-- import the db config file; its values can be referenced later via ${} -->
    <property file="src/main/resources/application.yml" />
    <property name="project.version" value="1.0.0" />
    <property name="project.name" value="sqltoy-helloworld" />
    <!-- project default package -->
    <property name="project.package" value="com.sqltoy.helloworld" />

    <!-- database definition; concrete values can also be written here directly -->
    <datasource name="helloworld" url="${spring.datasource.url}"
        driver="com.mysql.cj.jdbc.Driver" schema="helloworld"
        username="${spring.datasource.username}" password="${spring.datasource.password}" />

    <!-- dist: where generated code goes (relative to baseDir in the pom); encoding: file charset -->
    <tasks dist="src/main/java" encoding="UTF-8">
        <!-- multiple tasks can be configured to generate pojo into different packages; include matches table names with a regex -->
        <task datasource="helloworld" author="zhongxuchen" include="^SQLTOY_\w+" active="true">
            <!-- entity generates pojo; substr strips the table-name prefix; name uses the #{subName} placeholder; lombok-chain generates chained
setters -->
            <entity package="${project.package}.entity" substr="Sqltoy" name="#{subName}" lombok-chain="true" />
            <!-- vo generates dto/vo -->
            <vo package="${project.package}.dto" lombok-chain="true" substr="Sqltoy" name="#{subName}VO" />
        </task>
    </tasks>
</quickvo>
```

> **Zero-install variant**: the demo project sqltoy-helloworld points the datasource straight at an in-memory H2 whose url carries `INIT=RUNSCRIPT` to create the tables when the connection opens; with the h2 driver added to the plugin dependencies, `mvn quickvo:quickvo` needs no database at all. Switch back to the `${spring.datasource.url}` reference form above when generating against a real database.

```
<datasource name="helloworld" url="jdbc:h2:mem:quickvogen;INIT=RUNSCRIPT FROM 'src/main/resources/mock/h2/schema-h2.sql'"
    driver="org.h2.Driver" schema="PUBLIC" catalog="" username="sa" password="" />
```

Common configuration:

Configuration	Description
`<property file>`	Imports an external config file (e.g. application.yml); its values can then be referenced via `${key}`
`<datasource>`	Database connection; always configure `schema` / `catalog` correctly for multi-instance databases
`<tasks dist encoding>`	Root path and charset of the generated code
`<task include active>`	`include` matches the tables to generate with a regex (e.g. `^SQLTOY_\w+`); `active` enables/disables the task
`<entity>` / `<vo>`	Generate POJO and DTO/VO respectively; `package` is the target package, `substr` strips the table-name prefix, `name` is the class-name template (`#{subName}` is the table name with the prefix removed), `lombok-chain` generates chained setters

> For more options such as primary key strategy (`primary-key`), cascade (`cascade`), type mapping (`type-mapping`), and swagger annotations, see quickvo Code Generator and the official README.

3. Run the Generation

Execute in the project root:

```
mvn quickvo:quickvo
```

After execution:

- `OrderInfoVO.java` is generated under `src/main/java/com/sqltoy/helloworld/dto/` ;
- `OrderInfo.java` is generated under `src/main/java/com/sqltoy/helloworld/entity/` , containing annotations such as `@Entity` , `@Id` , `@Column` that describe the object-to-table mapping.

The generated POJO looks like:

```
@Data
@Accessors(chain = true)
@Entity(tableName="sqltoy_order_info", comment="sqltoy order info demo table", pk_constraint="PRIMARY")
public class OrderInfo implements Serializable {
    private static final long serialVersionUID = 7200696852961513069L;
/*---begin-auto-generate-don't-update-this-area--*/
    /** Order ID */
    @Id(strategy="generator", generator="org.sagacity.sqltoy.plugins.id.impl.NanoTimeIdGenerator")
    @Column(name="ORDER_ID", comment="Order ID", length=32L, type=java.sql.Types.VARCHAR,
            nativeType="VARCHAR", nullable=false)
    private String orderId;
    // ... remaining fields
/*---end-auto-generate-don't-update-this-area--*/
}
```

> [!WARNING] The content between `begin-auto-generate` and `end-auto-generate` is generated automatically by quickvo — **do not modify it manually** (it will be overwritten on regeneration). Put extension properties/methods outside the marked section.

For the complete create-project → generate → query workflow, see the helloworld quick start.

Quick Start

Principles for learning sqltoy-orm

- Do not carry over habits from MyBatis(-Plus) or similar frameworks
- Think in explicit logic and straightforward patterns
- When you see something slightly different from what you know, don't rush to reject it

Steps to build a sqltoy project

> This article is fully in sync with the demo project sqltoy-helloworld: the project runs on an **in-memory H2 database** by default (schema and seed data load automatically), so `mvn test` works right after cloning — zero install (9 demo tests, all green).

1. Create a Spring Boot project and configure the data source

- See: sqltoy demo project sqltoy-helloworld

```
@SpringBootApplication
@ComponentScan(basePackages = { "com.sqltoy.helloworld" })
@EnableTransactionManagement
public class SqlToyApplication {
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		SpringApplication.run(SqlToyApplication.class, args);
	}
}
```

- Hikari is used as the connection pool (choose any pool you prefer)

```
<!-- Spring's built-in database connection pool (version managed by spring-boot-starter-parent) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!-- zero-install in-memory database: the demo project runs on H2 by default -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

- Configure application.yml (the zero-install H2 form, identical to the demo project)

```
spring:
    # Zero install: H2 in-memory database by default; schema and seed data are
    # loaded by spring.sql.init (mock/h2/)
    datasource:
       name: dataSource
       type: com.zaxxer.hikari.HikariDataSource
       driver-class-name: org.h2.Driver
       username: sa
       password: ""
       isAutoCommit: false
       # random database name per Spring context, so test contexts never interfere
       url: jdbc:h2:mem:helloworld-${random.uuid};DB_CLOSE_DELAY=-1
    # demo schema + seed data (H2 DDL in step 3)
    sql:
       init:
         mode: always
         schema-locations: classpath:mock/h2/schema-h2.sql
         data-locations: classpath:mock/h2/data-h2.sql
    # note: sqltoy properties live under the spring.sqltoy prefix
    # (a common mistake is omitting the "spring." prefix so nothing gets loaded)
    sqltoy:
       # Optional: where the sql.xml files live (scanned recursively; multiple paths example:
classpath:com/sqltoy/helloworld,classpath:com/sqltoy/system
       sqlResourcesDir: classpath:com/sqltoy/helloworld
       # Defaults to false; in debug mode executed SQL is printed and sql files are watched and reloaded on change
       debug: true
```

- Switching to MySQL (or any real database): replace the datasource section and set `sql.init.mode` to `never` (create the tables with the MySQL DDL in step 3)

```
spring:
    datasource:
       name: dataSource
       type: com.zaxxer.hikari.HikariDataSource
       driver-class-name: com.mysql.cj.jdbc.Driver
       username: helloworld
       password: helloworld
       isAutoCommit: false
       url: jdbc:mysql://127.0.0.1:3306/helloworld?useUnicode=true&characterEncoding=utf-
8&serverTimezone=GMT%2B8&useSSL=false&allowPublicKeyRetrieval=true
```

> **Best practice: keep `*.sql.xml` files in the code directory (not resources)**, next to the VOs of the same module. Benefits: ① developers write and maintain SQL inside their own module without jumping between directories; ② the whole module can be extracted and reused as a unit, evolving toward productized modules. This requires a `resources` section in pom.xml so XML files under the java source directory get packaged:

```
<resources>
       <resource>
              <directory>src/main/java</directory>
              <excludes>
                     <exclude>**/*.java</exclude>
              </excludes>
              <includes>
                     <include>**/*.xml</include>
              </includes>
       </resource>
       <resource>
              <directory>src/main/resources</directory>
              <includes>
                     <include>**/*.xml</include>
                     <include>**/*.properties</include>
                     <include>**/*.yml</include>
                     <!-- H2 schema / seed-data scripts -->
                     <include>**/*.sql</include>
              </includes>
       </resource>
</resources>
```

2. Add sqltoy-orm-spring-starter in pom.xml

- Spring Boot

```
<dependency>
       <groupId>com.sagframe</groupId>
       <artifactId>sagacity-sqltoy-spring-starter</artifactId>
       <!-- For JDK 8 use 5.6.96.jre8 (final) -->
       <version>6.0.9</version>
</dependency>
<!-- Cache translate (see the Cache Translate tutorial) needs a cache implementation;
     sqltoy supports ehcache by default (runtime scope so it is packaged into the boot jar) -->
<dependency>
       <groupId>org.ehcache</groupId>
       <artifactId>ehcache</artifactId>
       <scope>runtime</scope>
</dependency>
```

- Solon

```
<dependency>
       <groupId>com.sagframe</groupId>
       <artifactId>sagacity-sqltoy-solon-plugin</artifactId>
       <!-- For JDK 8 use 5.6.96.jre8 (final) -->
       <version>6.0.9</version>
</dependency>
```

3. Create the table: sqltoy_order_info

- Zero-install H2 form: the DDL lives in `src/main/resources/mock/h2/schema-h2.sql` and is executed at startup by `spring.sql.init` (the demo project also ships the organ/dictionary tables used by the cache-translate demos, plus seed data in `data-h2.sql`):

```
DROP TABLE IF EXISTS SQLTOY_ORDER_INFO;
CREATE TABLE SQLTOY_ORDER_INFO (
    ORDER_ID     VARCHAR(32) NOT NULL,
    ORDER_TYPE   VARCHAR(32),
    PRODUCT_CODE VARCHAR(32),
    UOM          VARCHAR(30),
    PRICE        DECIMAL(24, 6),
    QUANTITY     DECIMAL(24, 6),
    TOTAL_AMT    DECIMAL(24, 6),
    STAFF_CODE   VARCHAR(32),
    ORGAN_ID     VARCHAR(32),
    STATUS       INT,
    CREATE_BY    VARCHAR(32),
    CREATE_TIME  TIMESTAMP,
    UPDATE_BY    VARCHAR(32),
    UPDATE_TIME  TIMESTAMP,
    CONSTRAINT PK_ORDER_INFO PRIMARY KEY (ORDER_ID)
);
COMMENT ON TABLE SQLTOY_ORDER_INFO IS 'sqltoy demo order table';
COMMENT ON COLUMN SQLTOY_ORDER_INFO.ORDER_ID IS 'order id';
-- ... remaining column comments: see schema-h2.sql in the demo project
```

- Real-database form (MySQL):

```
DROP TABLE IF EXISTS SQLTOY_ORDER_INFO;
CREATE TABLE SQLTOY_ORDER_INFO(
    `ORDER_ID` VARCHAR(32) NOT NULL  COMMENT 'order id' ,
    `ORDER_TYPE` VARCHAR(32)   COMMENT 'order type' ,
    `PRODUCT_CODE` VARCHAR(32)   COMMENT 'product code' ,
    `UOM` VARCHAR(30)   COMMENT 'unit of measure' ,
    `PRICE` DECIMAL(24,6)   COMMENT 'price' ,
    `QUANTITY` DECIMAL(24,6)   COMMENT 'quantity' ,
    `TOTAL_AMT` DECIMAL(24,6)   COMMENT 'total amount' ,
    `STAFF_CODE` VARCHAR(32)   COMMENT 'sales person' ,
    `ORGAN_ID` VARCHAR(32)   COMMENT 'sales department' ,
    `STATUS` INT   COMMENT 'order status' ,
    `CREATE_BY` VARCHAR(32)   COMMENT 'created by' ,
    `CREATE_TIME` DATETIME   COMMENT 'create time' ,
    `UPDATE_BY` VARCHAR(32)   COMMENT 'updated by' ,
    `UPDATE_TIME` DATETIME   COMMENT 'update time' ,
    PRIMARY KEY (ORDER_ID)
)  COMMENT = 'sqltoy demo order table';
```

4. Configure the quickvo-maven-plugin to generate POJOs and DTOs

- Add quickvo-maven-plugin to pom.xml

```
<plugin>
	<groupId>com.sagframe</groupId>
		<artifactId>quickvo-maven-plugin</artifactId>
		<version>2.0.4</version>
	<configuration>
		<configFile>./src/main/resources/quickvo.xml</configFile>
		<baseDir>${project.basedir}</baseDir>
	</configuration>
	<dependencies>
		<!-- H2 driver: quickvo boots its own in-memory schema via INIT=RUNSCRIPT,
		     so reverse generation works with zero install -->
		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<version>2.4.240</version>
		</dependency>
		<!-- add the driver of your real database when generating against it, e.g. mysql -->
		<dependency>
			<groupId>com.mysql</groupId>
			<artifactId>mysql-connector-j</artifactId>
			<version>${mysql.version}</version>
		</dependency>
	</dependencies>
</plugin>
```

- Create quickvo.xml under src/main/resources (identical to the demo project — the H2 self-bootstrapping form). For the full quickvo.xml reference see: quickvo-maven-plugin

```
<?xml version="1.0" encoding="UTF-8"?>
<quickvo xmlns="http://www.sagframe.com/schema/quickvo"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.sagframe.com/schema/quickvo https://sagframe.github.io/schema/quickvo.xsd">
	<property name="project.version" value="1.0.0" />
	<property name="project.name" value="sqltoy-helloworld" />
	<!-- default package of the project -->
	<property name="project.package" value="com.sqltoy.helloworld" />

	<!-- Points at an in-memory H2 database: INIT=RUNSCRIPT in the url creates the tables
	     from mock/h2/schema-h2.sql when the connection opens. To generate against your own
	     database: change url/driver/schema/username/password (a <property file> import of
	     application.yml with ${spring.datasource.url} references works too) -->
	<datasource name="helloworld" url="jdbc:h2:mem:quickvogen;INIT=RUNSCRIPT FROM 'src/main/resources/mock/h2/schema-h2.sql'"
		driver="org.h2.Driver" schema="PUBLIC" catalog="" username="sa" password="" />

	<!-- dist: where generated java code goes, relative to baseDir in the pom -->
	<tasks dist="src/main/java" encoding="UTF-8">
		<!-- multiple tasks are allowed to generate POJOs into different packages -->
		<task datasource="helloworld" author="zhongxuchen" include="^SQLTOY_\w+" active="true">
			<entity package="${project.package}.entity" substr="Sqltoy"      name="#{subName}" lombok-chain="true" />
			<vo package="${project.package}.dto" lombok-chain="true" substr="Sqltoy" name="#{subName}VO" />
		</task>
	</tasks>

	<!-- type mapping: H2 TIMESTAMP columns would generate java.sql.Timestamp by default;
	     map them to LocalDateTime to stay consistent with the unified-fields handler
	     (SqlToyUnifyFieldsHandler assigns LocalDateTime values) -->
	<type-mapping>
		<sql-type native-types="TIMESTAMP" jdbc-type="TIMESTAMP" java-type="java.time.LocalDateTime" />
	</type-mapping>
</quickvo>
```

5. Run quickvo to generate the POJO and DTO

- In the project root run: mvn quickvo:quickvo
- `src/main/java/com/sqltoy/helloworld/dto/OrderInfoVO.java` is generated
- `src/main/java/com/sqltoy/helloworld/entity/OrderInfo.java` is generated containing `@Entity`, `@Id`, `@Column` annotations that map the object to the table

```
@Data
@Accessors(chain = true)
@Entity(tableName="sqltoy_order_info",comment="sqltoy demo order table",pk_constraint="PRIMARY")
public class OrderInfo implements Serializable {

	/**
	 *
	 */
	private static final long serialVersionUID = 7200696852961513069L;
/*---begin-auto-generate-don't-update-this-area--*/

	/**
	 * Order id
	 */
	@Id(strategy="generator",generator="org.sagacity.sqltoy.plugins.id.impl.NanoTimeIdGenerator")
	@Column(name="ORDER_ID",comment="order id",length=32L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
	private String orderId;

	/**
	 * Order type
	 */
	@Column(name="ORDER_TYPE",comment="order type",length=32L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=true)
	private String orderType;

	// ... remaining fields omitted for brevity (price, quantity, totalAmt, staffCode, organId,
	//     status and the audit fields; createTime/updateTime are generated as LocalDateTime
	//     thanks to the type-mapping above)
/*---end-auto-generate-don't-update-this-area--*/
}
```

> The area between `begin-auto-generate` and `end-auto-generate` is regenerated by quickvo — do not edit it by hand (regeneration overwrites it). Put extended properties/methods outside the markers: that is exactly how the demo project's OrderInfoVO adds its `@Translate` properties.

6. Create a service and a unit test

- 1. The OrderInfoService interface

```
package com.sqltoy.helloworld.service;

import java.util.Map;

import org.sagacity.sqltoy.model.Page;
import com.sqltoy.helloworld.dto.OrderInfoVO;

public interface OrderInfoService {
	/**
	 * Create an order
	 *
	 * @param orderInfoVO
	 */
	public void createOrderInfo(OrderInfoVO orderInfoVO);

	/**
	 * Paginated order search
	 *
	 * @param pageModel
	 * @param queryMap
	 * @return
	 */
	public Page<OrderInfoVO> searchOrderInfo(Page pageModel, Map queryMap);
}
```

- 2. The OrderInfoServiceImpl implementation

```
/**
 * Order service implementation
 *
 * @author zhongxuchen
 * @date 2025/2/5
 */
@Service("orderInfoService")
public class OrderInfoServiceImpl implements OrderInfoService {
	// inject the built-in LightDao
	@Autowired
	LightDao lightDao;

	// all single-table operations are done in an object-oriented way, similar to JPA
	@Override
	@Transactional
	public void createOrderInfo(OrderInfoVO orderInfoVO) {
		// use the built-in dto<-->pojo mapping to create the entity
		OrderInfo orderInfoEntity = lightDao.convertType(orderInfoVO, OrderInfo.class);
		// save
		lightDao.save(orderInfoEntity);
	}

	// complex queries are defined in sql.xml
	@Override
	public Page<OrderInfoVO> searchOrderInfo(Page pageModel, Map queryMap) {
		String sql = """
				select * from SQLTOY_ORDER_INFO t
				where 1=1
				#[and t.status in (:statusAry)]
				#[and t.create_time>=:beginTime]
				#[and t.create_time<=:endTime]
				""";
		return lightDao.findPage(pageModel, sql, queryMap, OrderInfoVO.class);
	}

}
```

- 3. The unit test OrderInfoServiceTest (identical to the demo project, with assertions)

```
@SpringBootTest
public class OrderInfoServiceTest {
	@Autowired
	OrderInfoService orderInfoService;

	@Autowired
	LightDao lightDao;

	@Test
	public void testCreateOrderInfo() {
		String productCode = "P-DEMO-" + System.nanoTime();
		OrderInfoVO orderInfoVO = new OrderInfoVO();
		orderInfoVO.setOrderType("PO");
		orderInfoVO.setOrganId("100007");
		orderInfoVO.setProductCode(productCode);
		orderInfoVO.setPrice(BigDecimal.valueOf(100));
		orderInfoVO.setQuantity(BigDecimal.valueOf(100));
		orderInfoVO.setTotalAmt(BigDecimal.valueOf(10000));
		orderInfoVO.setUom("KG");
		orderInfoVO.setStaffCode("S0001");
		orderInfoVO.setStatus(1);
		// with SqlToyUnifyFieldsHandler configured, audit fields are filled automatically:
		// orderInfoVO.setCreateBy("S0001");
		// orderInfoVO.setCreateTime(LocalDateTime.now());
		orderInfoService.createOrderInfo(orderInfoVO);

		// query it back: the primary key comes from NanoTimeIdGenerator (26 digits),
		// the audit fields from the unified-fields handler
		List<OrderInfo> saved = lightDao.findEntity(OrderInfo.class,
				EntityQuery.create().where("productCode=:pc").values(MapKit.map("pc", productCode)));
		assertEquals(1, saved.size());
		assertEquals(26, saved.get(0).getOrderId().length());
		assertEquals("system-auto", saved.get(0).getCreateBy());
	}

	@Test
	public void testUpdateOrderInfo() {
		// self-contained (no dependency on pre-existing rows): save an order first
		OrderInfo order = new OrderInfo().setOrderType("PO").setProductCode("P-UPD-" + System.nanoTime())
				.setUom("KG").setPrice(BigDecimal.valueOf(100)).setQuantity(BigDecimal.valueOf(100))
				.setTotalAmt(BigDecimal.valueOf(10000)).setStaffCode("S0001").setOrganId("100007").setStatus(1);
		lightDao.save(order);

		// elastic update: null fields on the entity are skipped, only assigned fields are written
		OrderInfo update = new OrderInfo().setOrderId(order.getOrderId());
		update.setQuantity(BigDecimal.valueOf(120));
		lightDao.update(update);

		OrderInfo reloaded = lightDao.load(new OrderInfo().setOrderId(order.getOrderId()));
		assertEquals(0, reloaded.getQuantity().compareTo(BigDecimal.valueOf(120)));
		// price was left null on the update entity — the elastic update does not touch it
		assertEquals(0, reloaded.getPrice().compareTo(BigDecimal.valueOf(100)));
	}

	@Test
	public void testSearchOrderInfo() {
		// dynamic-condition pagination: statusAry/beginTime apply, a null endTime is ignored
		Page<OrderInfoVO> pageModel = orderInfoService.searchOrderInfo(new Page(10, 1),
				MapKit.keys("statusAry", "beginTime", "endTime").values(new Integer[] { 1 },
						LocalDateTime.parse("2024-10-17T00:00:01"), null));
		System.out.println(JSON.toJSONString(pageModel.getRows()));
		assertTrue(pageModel.getRecordCount() >= 3); // the seed data holds 3 status=1 orders
	}

}
```

7. Run and verify

```
git clone https://gitee.com/sagacity/sqltoy-helloworld.git
cd sqltoy-helloworld
mvn test
```

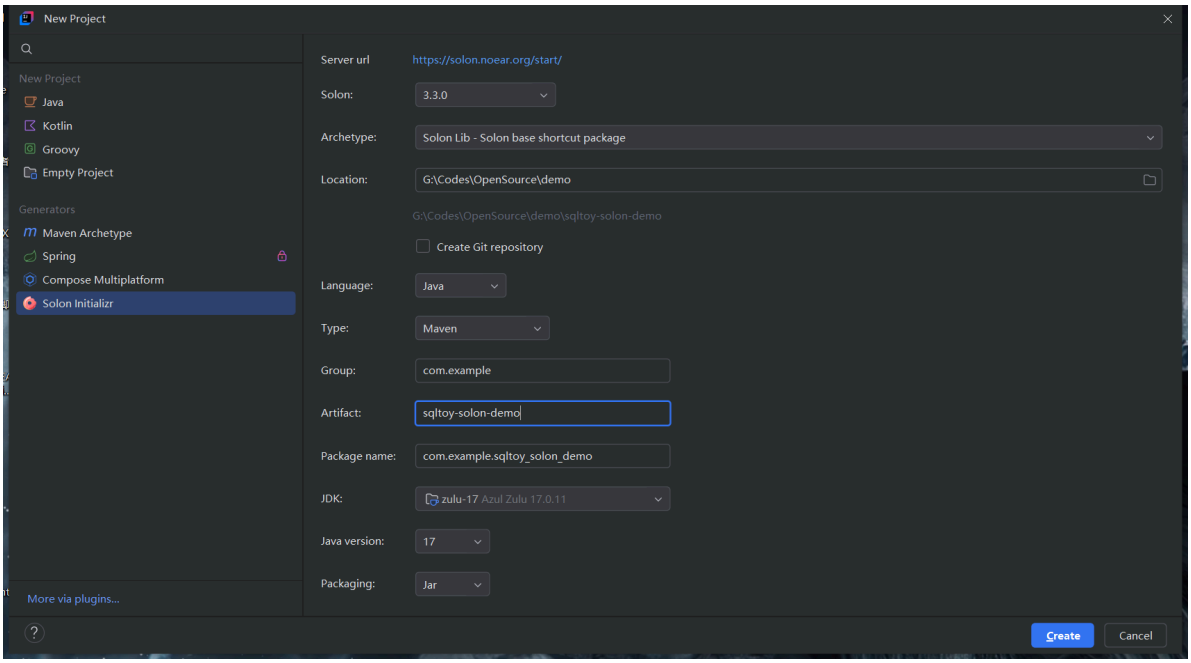
```
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0 — BUILD SUCCESS
```

Besides the CRUD/pagination tests above, the demo project ships `TranslateTest` covering the four cache-translate usages (the xml `<translate>` tag, `QueryExecutor.translates`, `EntityQuery.translates` and the `@Translate` annotation on VO properties — see Cache Translate in Action and Common-Field Auto-Fill), and `LambdaQueryTest` for lambda conditions (entity-getter conditions plus `setExpr` expression updates — see Lambda Queries and Updates).

Integrating SqlToy with Solon

1. Create the project

- Open your IDE (IntelliJ IDEA or Eclipse) and create a Maven project. Here the IDEA Solon plugin is used.
- Demo project: sqltoy-solon-demo



2. Add dependencies

- sqltoy Solon plugin

```
<!--sqltoy-solon-->
<dependency>
    <groupId>com.sagframe</groupId>
    <artifactId>sagacity-sqltoy-solon-plugin</artifactId>
    <version>6.0.9</version>
</dependency>
```

- MySQL connector

```
<!-- mysql connector -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.7.0</version>
    <scope>runtime</scope>
</dependency>
```

3. Configure the data source

- app.yml

```
myapp:
  db1:
    schema: demo
    jdbcUrl: jdbc:mysql://127.0.0.1:3306/demo?useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=Asia/Shanghai&tinyInt1isBit=false
    driverClassName: com.mysql.cj.jdbc.Driver
    username: root
    password: root
```

- Inject the datasource configuration (DbConfig.java)

```
@Configuration
public class DbConfig {
    @Bean(name = "db1", typed = true)
    public DataSource db1(@Inject("${myapp.db1}") HikariDataSource ds) {
        return ds;
    }
}
```

HikariDataSource is used as the connection pool, so one extra dependency is needed:

```
<!-- connection pool -->
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
</dependency>
```

- Then the sqltoy configuration

```
sqltoy:
  # scan path for the sql xml files
  sqlResourcesDir: classpath:sqltoy
  debug: true
```

4. Use the sqltoy API in Solon

1. Create a controller for testing (TestController.java)

```
@Controller
public class TestController {
    @Db
    private LightDao lightDao;

    @Get
    @Mapping("/order/findAll")
    public Object query() {
        return lightDao.find("sqltoy_order_info_find", new HashMap<>());
    }
}
```

2. Create the template sql (sqltoy/sqltoy_order_info.sql.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<sqltoy xmlns="http://www.sagframe.com/schema/sqltoy"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.sagframe.com/schema/sqltoy https://sagframe.github.io/schema/sqltoy.xsd">

    <sql id="sqltoy_order_info_find">
        <value>
            <![CDATA[
            select *
            from sqltoy_order_info
            order by ORDER_ID desc
            ]]>
        </value>
    </sql>

</sqltoy>
```

5. Run and test

1. Run the application (App.java)
2. Open: `http://localhost:8080/order/findAll`

Why common fields need unified assignment

- Regular business logic needs auditing:
  1. when a record is created, maintain the initial created-by and create-time;
  2. every modification maintains updated-by and update-time;
  3. when updating, the front end often sends back the previous values, so a forced overwrite is required. Therefore tables are usually designed with 4 standard columns: create_by / create_time / update_by / update_time. Assigning them by hand in code is repetitive and error-prone — the framework should handle it uniformly! Another key value: when updating, updated-by / update-time are maintained automatically (with forced overwrite of last-update-time where needed), **laying the groundwork for data auditing and update-time-based incremental ETL** — as long as unified assignment is in place, incremental extraction can rely on update_time directly, with no extra bookkeeping.

Implementing the sqltoy common-field handler

- 1. SqlToy provides the standard interface: org.sagacity.sqltoy.plugins.IUnifyFieldsHandler Taking creation as an example, the assignment logic is:
  1. fields that don't exist on the object are skipped automatically (object CRUD knows the real field names);
  2. elastic assignment: if e.g. createBy already has a value it is kept as-is.

```
// logic sketch:
if(entity.getCreateBy()==null){entity.setCreateBy(unifyCreateMap.get("createBy"));}
```

- 2. Register the handler in application.yml

```
spring:
    sqltoy:
        unifyFieldsHandler: com.sqltoy.plugins.SqlToyUnifyFieldsHandler
```

- 3. The SqlToyUnifyFieldsHandler code

```
package com.sqltoy.plugins;

import java.time.LocalDateTime;
import java.util.Map;

import org.sagacity.sqltoy.model.IgnoreCaseSet;
import org.sagacity.sqltoy.model.MapKit;
import org.sagacity.sqltoy.plugins.IUnifyFieldsHandler;

/**
 * @project sqltoy-helloworld
 * @description common-field auto-fill example
 * @author zhongxuchen <a href="mailto:zhongxuchen@gmail.com">contact</a>
 * @version v1.0,Date:2018-01-18
 */
public class SqlToyUnifyFieldsHandler implements IUnifyFieldsHandler {
	private String defaultUserName = "system-auto";

	 /**
	  * Fields and values assigned on creation (elastic: values already set by the caller win; nulls get filled)
	  */
	 public Map<String, Object> createUnifyFields() {
		LocalDateTime nowTime = LocalDateTime.now();
		// get the current user
		String userId = getUserId();
		// non-existent fields are skipped automatically, e.g. you may list both createBy and createdBy
		return MapKit.keys("createBy", "createTime", "updateBy", "updateTime", "enabled").values(userId, nowTime,
				userId, nowTime, 1);
	 }

	/**
	 * Fields and values assigned on update (elastic)
	 */
	@Override
	public Map<String, Object> updateUnifyFields() {
		return MapKit.keys("updateBy", "updateTime").values(getUserId(), LocalDateTime.now());
	}

	/**
	 * Force-updated fields (usually the updateTime property)
	 */
	@Override
	public IgnoreCaseSet forceUpdateFields() {
		IgnoreCaseSet forceUpdateFields = new IgnoreCaseSet();
		forceUpdateFields.add("updateTime");
		return forceUpdateFields;
	}

	/**
	 * Get the current user id
	 */
	private String getUserId() {
		// in real projects the current user is usually kept in a ThreadLocal set by a Filter,
		// e.g. with spring-security
		// return (SpringSecurityUtils.getCurrentUser() != null) ? SpringSecurityUtils.getCurrentUser().getId() : defaultUserName;
		return defaultUserName;
	}
}
```

Unit test verifying the auto-fill

```
@SpringBootTest
public class OrderInfoServiceTest {
	@Autowired
	OrderInfoService orderInfoService;

	@Test
	public void testCreateOrderInfo() {
		OrderInfoVO orderInfoVO = new OrderInfoVO();
		orderInfoVO.setOrderType("PO");
		orderInfoVO.setOrganId("T001");
		orderInfoVO.setProductCode("P0001");
		orderInfoVO.setPrice(BigDecimal.valueOf(100));
		orderInfoVO.setQuantity(BigDecimal.valueOf(100));
		orderInfoVO.setTotalAmt(BigDecimal.valueOf(10000));
		orderInfoVO.setUom("KG");
		orderInfoVO.setStaffCode("S0001");
		orderInfoVO.setStatus(1);
		// with SqlToyUnifyFieldsHandler registered there is no need to set:
		// orderInfoVO.setCreateBy("S0001");
		// orderInfoVO.setCreateTime(LocalDateTime.now());
		// orderInfoVO.setUpdateBy("S0001");
		// orderInfoVO.setUpdateTime(LocalDateTime.now());
		orderInfoService.createOrderInfo(orderInfoVO);
	}
}
```

Configuration

sqltoy Configuration

- Spring projects use the `spring.sqltoy.` prefix;
- Solon projects use the `solon.sqltoy.` prefix;
- In traditional Spring XML mode, set properties with the same names via `<property>` on the `SqlToyContext` bean.

Below are **all configuration parameters** grouped by purpose (based on the spring-boot-starter's `SqlToyContextProperties` and `SqlToyContext`, including default values). The vast majority of projects only need to set 3~5 of them; see Typical Project Usage at the end.

---

1. Resource Loading and Debug Mode

```
# Path to sql.xml resources, comma-separated for multiple values; supports AntPath patterns such as classpath*:com/**/*.sql.xml (5.6.75+)
spring.sqltoy.sqlResourcesDir=classpath:com/nebula/crm/modules
# Directly specify concrete sql.xml file resources (array); use alone or together with sqlResourcesDir
spring.sqltoy.sqlResources[0]=classpath:com/nebula/system.sql.xml
# Entity scan packages (array), based on the @Entity/@SqlToyEntity annotations; no configuration needed by default (loaded on first use)
spring.sqltoy.packagesToScan[0]=com.yourproject.modules
# Extra annotated classes (array); generally no longer necessary
#spring.sqltoy.annotatedClasses[0]=com.yourproject.SomeEntity
# Encoding of sql files
spring.sqltoy.encoding=UTF-8
# Whether to enable debug mode, default false; when true, executed sql and elapsed time are logged, and the sql file change check interval becomes 2
seconds (15 seconds when false)
spring.sqltoy.debug=true
# Interval for detecting sql file script changes (seconds); <0 or >86400 (one day) disables change detection
spring.sqltoy.scriptCheckIntervalSeconds=2
# Delay in seconds before detection of cache updates and sql script updates starts
spring.sqltoy.delayCheckSeconds=300
# Whether to throw an exception and terminate when a duplicate sqlId is found, default true; if false, the later definition overrides the earlier one
spring.sqltoy.breakWhenSqlRepeat=true
# sql log output formatter: defaultSqlFormater or default (uses druid's sql formatting, requires druid on the classpath)
spring.sqltoy.sqlFormater=defaultSqlFormater
```

2. Datasources and Dialects

```
# In multiple-datasource scenarios, specify the default datasource; same effect as Spring's @Primary
spring.sqltoy.defaultDataSource=crmDataSource
# Custom datasource selector (implements DataSourceSelector) for dynamic datasource routing
#spring.sqltoy.dataSourceSelector=com.yourproject.MyDataSourceSelector
# Connection management implementation extension (implements ConnectionFactory); commonly used in pure Java environments
#spring.sqltoy.connectionFactory=org.sagacity.sqltoy.integration.impl.SimpleConnectionFactory
# Database dialect; generally no need to set it (auto-detected by the framework)
#spring.sqltoy.dialect=mysql
# Whether sqlId dialect-variant lookup prefers the real dialect detected from the connection (new in 6.0.3, default false)
# Typical cases: OceanBase configured with the mysql dialect while the real database is oceanbase, proxied databases, etc.;
# when enabled, if the real-dialect variant does not exist the original configured-dialect lookup chain is used as fallback,
# and behavior is unchanged when the real dialect equals the configured one
#spring.sqltoy.realDialectFirst=false
# Database dialect parameter configuration (Map), used to override/supplement dialect behavior
#spring.sqltoy.dialectConfig.xxx=yyy
# Dialect mapping for different databases (Map), e.g. treat a non-built-in database as a certain dialect: OSCAR-->oracle
#spring.sqltoy.dialectMap.OSCAR=oracle
# Cross-database adaptation verification: any query is replayed once on these datasources (array), for productized multi-database testing
#spring.sqltoy.redoDataSources[0]=oracleDB
# Database reserved words, comma-separated (object operations handle them automatically; custom sql adapts accordingly when running across databases)
spring.sqltoy.reservedWords=maxvalue,minvalue,name
# Default query timeout (seconds)
#spring.sqltoy.defaultStatementTimeout=30
```

3. Batch, Pagination and Execution

```
# Batch size for batch operations (saveAll/updateAll etc.), default 200
spring.sqltoy.batchSize=200
# Default number of records fetched on the database side (fetchSize), default -1; generally no need to set it
#spring.sqltoy.fetchSize=1000
# Maximum records per page, default 100000; <0 means unlimited. Prevents full data extraction when pageNo=-1
spring.sqltoy.pageFetchSizeLimit=100000
# Default records per page, default 10
spring.sqltoy.defaultPageSize=10
# For unmatched database types, whether pagination uses the limit ? offset ? pattern or the limit ?,? pattern
#spring.sqltoy.defaultPageOffset=false
# Whether to return to the first page when paging beyond the last page
spring.sqltoy.overPageToFirst=true
# Record-count threshold above which a log reminder is printed for large-volume data modifications, default 2000
spring.sqltoy.updateTipCount=2000
# Split merge into into two steps: updateAll + saveAllIgnoreExist (seata distributed transactions do not support merge)
#spring.sqltoy.splitMergeInto=false
# For change-type sql executed via executeSql, whether blank parameters are converted to null by default
#spring.sqltoy.executeSqlBlankToNull=true
```

4. Cache Translate

```
# Cache translate config file, default classpath:sqltoy-translate.xml;classpath:translates
# (classpath:translates means files under that path ending with .trans.xml/.translates.xml/-translate.xml/-translates.xml)
spring.sqltoy.translateConfig=classpath:sqltoy-translate.xml
# Cache component used by cache translate, default ehcache, caffeine also available
spring.sqltoy.cacheType=caffeine
# Custom cache manager (implements TranslateCacheManager)
#spring.sqltoy.translateCacheManager=com.yourproject.MyTranslateCacheManager
# Implementation for dynamically fetching the cache (usually redis-based), for very large master data; see FIFO dynamic cache for details
#spring.sqltoy.dynamicCacheFetch=com.yourproject.MyDynamicCacheFetch
# Cache manager for dynamically fetched cache; the framework provides a default implementation (FIFODynamicFetchCacheManager)
#spring.sqltoy.dynamicFecthCacheManager=...
# Number of days to cache distributed id keys (usually redis-based, to avoid long-term space usage); expires after that many days
#spring.sqltoy.distributeIdCacheExpireDays=30
```

5. Function Adaptation (Cross-Database)

```
# Enable automatic adaptation of database sql functions; custom functions can also be attached: default,com.yourpackage.Instr
# (with an identical class name, a custom implementation overrides the framework default; close disables it)
spring.sqltoy.functionConverts=default
```

6. Field Processing and Types

```
# Unified field assignment handler (unifyFieldsHandler): fills createBy/createTime/updateBy/updateTime etc. only when they are empty
spring.sqltoy.unifyFieldsHandler=com.sqltoy.plugins.SqlToyUnifyFieldsHandler
# Custom TypeHandler (SPECIAL cases only!). JSON/JSONB, vector, geometry columns
# and enum conversion are all handled by the framework out of the box — no config needed;
# this is only for genuinely special value objects, e.g. Money(CNY,88000) <-> "CNY:88000"
spring.sqltoy.typeHandler=com.sqltoy.plugins.MoneyTypeHandler
# The JSON library behind JSON columns is pluggable: by default it is auto-detected from the
# classpath in the order fastjson2 -> jackson3 -> jackson2. A detected library must also pass a
# smoke parse before it is accepted — being on the classpath is not enough (e.g. jackson3's
# databind needs jackson-annotations 2.20+ and throws NoSuchFieldError when an older one wins
# the version conflict); an unusable candidate is logged as a warning and the next one is tried.
# When none is usable, touching a JSON column fails with a clear pointer instead of a
# NoClassDefFoundError. Pin one explicitly when you need to (e.g. both fastjson2 and jackson are
# on the classpath but you want jackson): the value is either a built-in alias
# (fastjson2 / jackson3 / jackson2) or the fully qualified class name of a custom
# org.sagacity.sqltoy.plugins.json.JsonProvider implementation.
#spring.sqltoy.jsonProvider=jackson2
#spring.sqltoy.jsonProvider=com.xxx.MyJsonProvider
# When queries return List<Map>, whether map labels are converted to camelCase, default true
spring.sqltoy.humpMapResultTypeLabel=true
# Strategy for obtaining ResultSetMetaData column labels: default (no processing)/upper (uppercase)/lower (lowercase)
spring.sqltoy.columnLabelUpperOrLower=default
# Format of LocalDateTime in sql log output; can be set to auto or yyyy-MM-dd HH:mm:ss.SSSSSS etc.
spring.sqltoy.localDateTimeFormat=yyyy-MM-dd HH:mm:ss.SSSSSS
# Format of LocalTime in sql log output; can be set to auto or HH:mm:ss.SSSSSS etc.
spring.sqltoy.localTimeFormat=HH:mm:ss.SSSSSS
# Default locale (e.g. zh_CN, en-US), affects the locale symbols used for date/number formatting and parsing; when null, the JVM default locale is
used
#spring.sqltoy.defaultLocale=zh_CN
# Whether the like query ESCAPE clause uses double backslashes: true=ESCAPE '\\', false=ESCAPE '\', null=decided automatically by dialect
#spring.sqltoy.backslashEscaping=
```

7. Security (Encryption/Decryption and Masking)

```
# RSA keys for field encryption/decryption
spring.sqltoy.securePrivateKey=classpath:mock/rsa_private.key
spring.sqltoy.securePublicKey=classpath:mock/rsa_public.key
# Custom encryption/decryption implementation (RSA by default, no need to set it); interface org.sagacity.sqltoy.plugins.secure.FieldsSecureProvider
#spring.sqltoy.fieldsSecureProvider=com.yourproject.FieldsSecureProvider
# Custom masking implementation (provided by default, no need to set it); interface org.sagacity.sqltoy.plugins.secure.DesensitizeProvider
#spring.sqltoy.desensitizeProvider=com.yourproject.DesensitizeProvider
```

SQL injection validation regexes (`sqlInjectionRegexes`)

```
# Custom regexes (array) for SQL injection detection, replacing the built-in default function/comment/condition/keyword patterns
# Mainly used by the <valid-sqlInjection> parameter validation in <filters>; can also be used with custom interceptors
#spring.sqltoy.sqlInjectionRegexes[0]=^(SELECT|INSERT|UPDATE|DELETE|DROP|ALTER|CREATE|TRUNCATE)\\b
```

> Note: when not configured, the framework's built-in defaults apply (four pattern groups: function calls, comments, always-true conditions, and SQL keywords). The configuration takes effect in the **first initialized SqlToyContext within the JVM** and cannot be weakened by later contexts (security-related). The solon-plugin supports the same property.

8. Automatic DDL Table Creation

```
# Whether to automatically create/update table structures from POJOs, default false; requires packagesToScan
spring.sqltoy.autoDDL=true
# Whether table/column names in generated DDL are converted to lowercase or uppercase (configurable via the spring.sqltoy prefix since 6.0.3)
#spring.sqltoy.ddlLowerOrUpper=lower
# Custom database DDL generator (implements DialectDDLGenerator)
#spring.sqltoy.dialectDDLGenerator=com.yourproject.MyDDLGenerator
# When a single-record save uses the identity/sequence primary-key strategy and returns the primary key value, the case handling of the field name
(Map: lower/upper)
#spring.sqltoy.dialectReturnPrimaryColumnCase.mysql=lower
```

9. Interceptors, Slow SQL and Extensions

```
# sql execution interceptors (array), can modify the current sql, e.g. unified tenant isolation/overreach filtering
spring.sqltoy.sqlInterceptors[0]=org.sagacity.sqltoy.plugins.interceptors.TenantFilterInterceptor
# Custom filter handler (reserved for future use)
#spring.sqltoy.customFilterHandler=...
# Slow sql duration threshold (milliseconds); beyond it the sql is logged and put into the slow sql queue, default 8000 (8 seconds)
spring.sqltoy.printSqlTimeoutMillis=3000
# Slow sql collection handler, default DefaultOverTimeHandler
# Access: lightDao.getSqlToyContext().getOverTimeSqlHandler().getSlowest(100, true)
spring.sqltoy.overTimeSqlHandler=org.sagacity.sqltoy.plugins.overtime.DefaultOverTimeHandler
# Capture of the business code call site (FirstBizCodeTrace), used to trace the business origin of a sql
#spring.sqltoy.firstBizCodeTrace=...
```

Thread Pool (taskExecutor)

The thread pool used by parallel query, parallel pagination, parallel batch, etc., with the prefix `spring.sqltoy.taskExecutor.`:

```
# Specify a thread pool name; once set, that pool becomes the default; default none (uses sqltoy's built-in thread pool)
#spring.sqltoy.taskExecutor.targetPoolName=none
# Thread name prefix, default sqltoyThreadPool
spring.sqltoy.taskExecutor.threadNamePrefix=sqltoyThreadPool
# Core thread count, default CPU cores/2+1
spring.sqltoy.taskExecutor.corePoolSize=8
# Maximum thread count, default CPU cores*3
spring.sqltoy.taskExecutor.maxPoolSize=24
# Buffer queue capacity, default 200
spring.sqltoy.taskExecutor.queueCapacity=200
# Thread idle keep-alive time (seconds), default 60
spring.sqltoy.taskExecutor.keepAliveSeconds=60
# Whether to wait for tasks to complete on shutdown, default true
spring.sqltoy.taskExecutor.waitForTasksToCompleteOnShutdown=true
# Await termination timeout (seconds), default -1
spring.sqltoy.taskExecutor.awaitTerminationSeconds=-1
```

10. NoSQL (Elasticsearch)

ES node configuration uses the prefix `spring.sqltoy.elastic.` (multiple nodes/clusters supported); for full details see Elasticsearch Support:

```
spring.sqltoy.elastic.defaultId=default
spring.sqltoy.elastic.endpoints[0].id=default
spring.sqltoy.elastic.endpoints[0].url=http://192.168.56.101:9200
spring.sqltoy.elastic.endpoints[0].username=elastic
spring.sqltoy.elastic.endpoints[0].password=skyline
spring.sqltoy.elastic.endpoints[0].sqlPath=_nlpcn/sql
```

> MongoDB reuses spring-data's `spring.data.mongodb.*` configuration; see MongoDB Support.

---

11. GraalVM native build-time parameters (since 6.0.8)

These three parameters **only take effect while building a native image**; JVM mode ignores them entirely. They fill in the reflection and proxy needs the framework cannot infer from configuration. Background and verification results are in the GraalVM Native Image Guide.

```
# Reflection registration for non-entity DTO/VOs (packagesToScan only sees @Entity)
# Register by package: every class under the package is scanned recursively
spring.sqltoy.native-reflection-packages[0]=com.yourcorp.project.dto
spring.sqltoy.native-reflection-packages[1]=com.yourcorp.project.vo
# Register by class: a few special classes, classes referenced by name
spring.sqltoy.native-reflection-classes[0]=com.yourcorp.project.report.TreeModel
# Pre-generate proxy subclasses at build time for entities that use updateSaveFetch(EntityUpdateCallback)
# The application must add net.bytebuddy:byte-buddy itself (optional, not transitive, in the starter)
spring.sqltoy.native-entity-proxies[0]=com.yourcorp.project.entity.StockVO
```

Parameter	Value	Purpose
`native-reflection-packages`	array of packages	Registers reflection for every class in the package recursively (plain DTO/VOs, records, enums)
`native-reflection-classes`	array of fully qualified class names	Registers individual classes (DTO/VOs, classes referenced by name)
`native-entity-proxies`	array of fully qualified class names	Pre-generates the `updateSaveFetch` proxy for the listed entities at build time (the entity must not be `final` and needs an accessible no-arg constructor)

> On Solon the prefix is `solon.sqltoy.*` with the same parameter names. For YAML forms and complete examples on both stacks see section 3 of the GraalVM Native Image Guide.

Typical Project Usage

Don't be intimidated by the long list — the vast majority of projects only need to set 3~5 parameters:

```
# Recommended to put complex queries in xml files
spring.sqltoy.sqlResourcesDir=classpath:com/nebula/crm/modules
# Output executed sql, parameters and elapsed time to help locate problems
spring.sqltoy.debug=true
# Unified field assignment for common fields
spring.sqltoy.unifyFieldsHandler=com.sqltoy.plugins.SqlToyUnifyFieldsHandler
# Definition of the duration standard for slow sql
spring.sqltoy.printSqlTimeoutMillis=3000
# Custom TypeHandler (SPECIAL cases only). JSON/vector/geometry/enum are built in;
# only for special value objects — see CustomTypeHandlerTest in sqltoy-showcase
spring.sqltoy.typeHandler=com.sqltoy.plugins.MoneyTypeHandler
```

Multi-Datasource & Transactions

sqltoy itself is not bound to any datasource. For multiple datasources you can directly reuse the mature community **dynamic-datasource** (baomidou) solution: the `@DS` annotation for datasource switching, `@Transactional` for single-database transactions, and `@DSTransactional` to bring operations across multiple datasources into the same transaction.

> This article is adapted from the former `trunk/sqltoy-dynamic-datasource` demo sub-project (since removed from the sqltoy-showcase repository during its single-module restructuring; the code examples in this article are self-contained). For sqltoy's native named multi-datasource approach see `MultiDataSourceTest` in the showcase. The examples in this article target **sqltoy 6.0.4**

1. Add Dependencies

```
<!-- dynamic-datasource (spring-boot3 starter, works under Spring Boot 4.0.1) -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>dynamic-datasource-spring-boot3-starter</artifactId>
    <version>4.5.0</version>
</dependency>
<!-- Connection pool (druid as an example) -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-3-starter</artifactId>
    <version>1.2.28</version>
</dependency>
```

> [!WARNING] When using druid you must **exclude its auto-configuration**, otherwise it conflicts with dynamic-datasource:
> `@SpringBootApplication(exclude = DruidDataSourceAutoConfigure.class)`

2. Configure Multiple Datasources

`primary` specifies the default datasource; individual databases are declared under `datasource`; sqltoy specifies its default database via `defaultDataSource`:

```
spring:
    datasource:
        dynamic:
            primary: datasourceA          # default datasource
            datasource:
                datasourceA:
                    url: jdbc:mysql://127.0.0.1:3306/dyndatasourcea?useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8&useSSL=false
                    username: dyndatasourcea
                    password: dyndatasourcea
                    driver-class-name: com.mysql.cj.jdbc.Driver
                datasourceB:
                    url: jdbc:mysql://127.0.0.1:3306/dyndatasourceb?useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8&useSSL=false
                    username: dyndatasourceb
                    password: dyndatasourceb
                    driver-class-name: com.mysql.cj.jdbc.Driver
            druid:                        # common connection pool settings
                initial-size: 5
                min-idle: 5
                maxActive: 20
                validationQuery: SELECT 1 FROM DUAL
                testWhileIdle: true
    sqltoy:
        # Note the prefix is spring.sqltoy. ; do not write it as sqltoy. or the configuration will not be loaded
        sqlResourcesDir: classpath:com/sqltoy/dyndatasource
        debug: true
        # Specify sqltoy's default datasource
        defaultDataSource: datasourceA
```

3. Switch Datasources with @DS

Use `@DS("datasource name")` on Service methods, and sqltoy operations are routed to the corresponding database:

```
@Service
public class OrderInfoServiceImpl implements OrderInfoService {
	@Autowired
	private LightDao lightDao;

	@Override
	@DS("datasourceA")     // switch to database A
	public long saveLocalAOrders(List<OrderInfoVO> orderInfoVOs) {
		return lightDao.saveAll(orderInfoVOs);
	}

	@Override
	@DS("datasourceB")     // switch to database B
	public long saveLocalBOrders(List<OrderInfoVO> orderInfoVOs) {
		return lightDao.saveAll(orderInfoVOs);
	}
}
```

- The default datasource specified by `primary` does **not** need the `@DS` annotation;
- `@DS` can also be placed on the class, taking effect for all operations of the whole class.

4. Transaction Handling

Single-database transaction: use Spring's `@Transactional` as usual (no extra specification is needed for the primary datasource):

```
@Transactional
public void executeSql(String sql) {
	lightDao.executeSql(sql, MapKit.map());
}
```

Cross-database transaction: use `@DSTransactional` provided by dynamic-datasource to bring operations across multiple datasources into the same transaction — if any of them fails, everything rolls back:

```
@Service
public class DynamicServiceImpl implements DynamicService {

	@Autowired
	private OrderInfoService orderInfoService;

	/**
	 * Operations on databases A and B are brought into the same transaction
	 */
	@Override
	@DSTransactional
	public void saveOrder(List<OrderInfoVO> orderInfoVOAs, List<OrderInfoVO> orderInfoVOBs) {
		orderInfoService.saveLocalAOrders(orderInfoVOAs);   // write to database A
		orderInfoService.saveLocalBOrders(orderInfoVOBs);   // write to database B
	}
}
```

5. Other Multi-Datasource Approaches

Approach	Description
Multiple LightDao beans	Define multiple `LightDao` beans, each injected with a different dataSource, and reference them by bean name in Services
Query-level switching	`QueryExecutor.dataSource(...)` and the chained `.dataSource(...)` (see Link Chain API)
Specified in xml	`<sql dataSource="...">` specifies the datasource for a single SQL (see Dynamic SQL Guide)
Multi-database transaction (JTA)	Implemented in combination with JTA (e.g. Atomikos/Narayana)

> For more common questions (such as why sharding does not support cross-database join pagination, or how to combine sqltoy with other ORM frameworks), see the FAQ.

> 🎬 The full example sources (`DynamicServiceImpl.java` / `OrderInfoServiceImpl.java`) were removed together with the former `trunk/sqltoy-dynamic-datasource` sub-project; the code examples in this article are self-contained. For sqltoy's native named multi-datasource demo see `MultiDataSourceTest` in sqltoy-showcase.

Traditional Spring Setup

- Add the sqltoy dependency (for traditional Spring use sagacity-sqltoy-spring; only Spring Boot projects need the starter)

```
<!-- traditional spring (non-boot) -->
<dependency>
	<groupId>com.sagframe</groupId>
	<artifactId>sagacity-sqltoy-spring</artifactId>
	<!-- For JDK8, use 5.6.96.jre8 (final version, also works with the quickvo-maven-plugin) -->
	    <version>6.0.9</version>
</dependency>
```

- Define the sqltoy context and lightDao as beans based on Spring XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"
	default-autowire="byName" default-lazy-init="true">
	<!-- Define the sqltoy context -->
	<bean id="sqlToyContext" name="sqlToyContext" class="org.sagacity.sqltoy.SqlToyContext" init-method="initialize"
		destroy-method="destroy">
		<!-- Path to the sql.xml files; directories are searched recursively, and multiple paths can be separated by commas; optional property
-->
		<property name="sqlResourcesDir" value="classpath:com/company/project" />
		<!-- Converts functions across different databases; optional property -->
		<property name="functionConverts" value="default" />
		<!-- Unified field assignment; principle: it only fills in missing values — values already set by developers are never overwritten; it
takes effect only when no value has been assigned -->
		<property name="unifyFieldsHandler">
			<!-- Illustrative example here -->
			<bean class="com.company.framework.sqltoy.SqltoyUnifyFieldsHandler" />
		</property>
		<!-- Cache translate manager, optional property (not needed only when cache translation is not used) -->
		<property name="translateConfig" value="classpath:sqltoy-translate.xml" />
		<!-- Default value: false -->
		<property name="debug" value="${sqltoy.debug}" />
		<!-- Default value: 200; the batch size used by sqltoy batch updates -->
		<property name="batchSize" value="${sqltoy.batchSize}" />
		<!-- Default value: 100000; sets the maximum number of records a paginated query may fetch, preventing malicious data extraction from
putting pressure on system memory and safeguarding data security -->
		<property name="pageFetchSizeLimit" value="50000" />
		<!-- Default dataSource -->
		<property name="defaultDataSource" ref="dataSource" />
		<!-- Print the SQL if its execution time exceeds this many milliseconds -->
		<property name="printSqlTimeoutMillis" value="2000" />
		<!-- How long after application startup the check is performed -->
		<property name="delayCheckSeconds" value="300" />
		<property name="breakWhenSqlRepeat" value="false"/>
		<property name="overPageToFirst" value="true"/>
		<!-- The two key configurations: appContext and connectionFactory -->
		<property name="appContext">
			<bean class="org.sagacity.sqltoy.integration.impl.SpringAppContext"/>
		</property>
		<property name="connectionFactory">
			<bean class="org.sagacity.sqltoy.integration.impl.SpringConnectionFactory"/>
		</property>
	</bean>

	<!-- Define lightDao: it is recommended to use LightDao uniformly as the entry point for object operations -->
	<bean id="lightDao" name="lightDao"
		class="org.sagacity.sqltoy.dao.impl.LightDaoImpl" />
	<bean id="sqlToyCRUDService" name="sqlToyCRUDService"
		class="org.sagacity.sqltoy.service.impl.SqlToyCRUDServiceImpl" />
</beans>
```

- Concrete usage

```
@Autowired
private LightDao lightDao;

public void createOrgan(OrganInfo organInfo){
	lightDao.save(organInfo);
}
```

Pure Java Usage

sqltoy is recommended to run within the Spring or Solon framework; a fully standalone framework for independent (non-Spring) usage has not been completely built, and you can optimize and extend it on your own with AI. An example is provided below

- Step 1: Add the pom dependency

```
<dependency>
	<groupId>com.sagframe</groupId>
	<artifactId>sagacity-sqltoy</artifactId>
   <version>6.0.9</version>
</dependency>
```

- Java code: register the DataSource, build the SqlToyContext and lightDao

```
public void doDB() {
	try {
		// Build the sqlToyContext
		SqlToyContext sqlToyContext = new SqlToyContext();
		// sqlResourcesDir points at a directory; for a specific xml file use setSqlResources(List)
		sqlToyContext.setSqlResources(Arrays.asList("classpath:sqltoy/demo.sql.xml"));
		sqlToyContext.initialize();
		Map<String, String> map = new HashMap<>();
		map.put(DruidDataSourceFactory.PROP_URL, "jdbc:mysql://192.168.56.101:3306/java20");
		// Set the driver
		map.put(DruidDataSourceFactory.PROP_DRIVERCLASSNAME, "com.mysql.cj.jdbc.Driver");
		// Set the username
		map.put(DruidDataSourceFactory.PROP_USERNAME, "root");
		// Set the password
		map.put(DruidDataSourceFactory.PROP_PASSWORD, "123456");
		// Create the data source
		DataSource dataSource = DruidDataSourceFactory.createDataSource(map);
		sqlToyContext.setDefaultDataSource(dataSource);
		// The framework provides the DefaultLightDaoImpl implementation by default
		LightDao lightDao = new DefaultLightDaoImpl(sqlToyContext);
		// Non-transactional
		// lightDao.find("select * from staff_info where status=:status",
		// MapKit.map("status", "1"), StaffInfo.class);
		// All of this is illustrative; write it according to your actual logic
		Object result = DBTransUtils.doTrans(lightDao.getDataSource(), () -> {
			// Here you can
			lightDao.updateByQuery(StaffInfo.class,
					EntityUpdate.create().set("sexType", "F").where("staffId=?").values("S0001"));
			return lightDao.find("select * from staff_info where status=:status", MapKit.map("status", "1"),
					StaffInfo.class);
		});
	} catch (Exception e) {

	}
}
```

Solon Integration Reference

sqltoy supports the Solon framework through the standalone `sagacity-sqltoy-solon-plugin` module. The APIs are identical to the Spring version (LightDao / SqlToyLazyDao / SqlToyCRUDService, etc.); only configuration and injection differ. For a hands-on introduction, see the helloworld-solon demo.

1. Dependency and Version Requirements

```
<dependency>
    <groupId>com.sagframe</groupId>
    <artifactId>sagacity-sqltoy-solon-plugin</artifactId>
    <!-- keep the same version number as core -->
    <version>6.0.9</version>
</dependency>
```

- Built against **Solon 4.1.1**, with transitive dependencies on `solon-aot` and `solon-data`;
- Datasources are managed by the Solon container (register connection pools such as HikariCP as beans yourself); sqltoy does not manage them a second time.

2. Configuration (the solon.sqltoy prefix)

The configuration prefix is `solon.sqltoy` (the legacy prefix `sqltoy` is still honored); property names map almost one-to-one to the Spring version's `spring.sqltoy.*` — refer directly to sqltoy Configuration Parameters for their meanings:

```
solon.sqltoy:
  # sql scan path inside the xml (comma-separated for multiple)
  sqlResourcesDir: classpath:sqltoy
  # default classpath:sqltoy-translate.xml
  translateConfig: classpath:sqltoy-translate.xml
  debug: true
  # cross-database function adaptation (on by default)
  #functionConverts: default
```

Two differences from the Spring version:

Difference	Description
`cacheType` defaults to `solon`	Cache translate uses the Solon local cache manager (`SolonTranslateCacheManager`) by default; the Spring version defaults to `ehcache`
debug linkage	When Solon starts in debug mode (`Solon.cfg().isDebugMode()`), sqltoy's debug output is force-enabled

3. @Db Injection

On Solon you do not use `@Autowired`; injection goes through `org.sagacity.sqltoy.solon.annotation.Db` (on fields or method parameters):

Injection target	Description
`javax.sql.DataSource`	Inject the datasource directly
`SqlToyLazyDao`	Full-featured Dao
`LightDao`	Lightweight Dao (recommended for direct use in Services)
`SqlToyCRUDService`	Generic CRUD service

The `@Db` value specifies the datasource bean name; omit it for the default datasource:

```
@Controller
public class TestController {

    // default datasource
    @Db
    private LightDao lightDao;

    // by datasource bean name (matching @Bean(name="db1"))
    @Db("db1")
    private SqlToyLazyDao db1Dao;
}
```

The datasource-selection mechanism (the `datasource` attribute in sql, the lazyDao chained `.dataSource(...)`, the `defaultDataSource` fallback, etc.) is the same as the Spring version — see Multi-Datasource and Transactions.

4. GraalVM Native

The solon-plugin ships `org.sagacity.sqltoy.solon.integration.aot.SqltoyRuntimeNativeRegistrar` (implementing Solon's `RuntimeNativeRegistrar` interface). When building a native image it automatically registers sqltoy's configuration resources (`sqltoy-translate.xml`, etc.) and entity-class reflection metadata — no manual reflect-config additions needed; since 6.0.8 it also registers resource patterns and extension-class reflection dynamically from the configuration, and supports `native-reflection-packages` / `native-reflection-classes` for non-entity DTO/VO reflection.

> For the Spring Boot side of AOT and the full native guide, see the GraalVM Native Image Guide.

XML Schema Guide

All three sqltoy XML configuration files ship with official XSD schemas (hosted on GitHub: sagframe/schema). With the schema configured, IDEA gives you **tag completion, attribute hints and error highlighting** while editing.

The three config files and their schema locations

Config file	Purpose	xmlns (namespace — never changes)	File location in `xsi:schemaLocation`
`*.sql.xml`	sql / mql / eql query definitions	`http://www.sagframe.com/schema/sqltoy`	`https://sagframe.github.io/schema/sqltoy.xsd`
`sqltoy-translate.xml`	Cache-translate configuration	`https://www.sagframe.com/schema/sqltoy-translate`	`https://sagframe.github.io/schema/sqltoy-translate.xsd`
`quickvo.xml`	quickvo code-generation configuration	`http://www.sagframe.com/schema/quickvo`	`https://sagframe.github.io/schema/quickvo.xsd`

> [!IMPORTANT] The `xmlns` is the schema identity — it does **not** need to be reachable and must **never be modified**. The second URI in `xsi:schemaLocation` is only a hint of where the file lives. The framework never downloads the XSD at runtime; it serves IDE completion and validation only.

Legacy address (transition)

Historical versions referenced the `http://www.sagframe.com/schema/sqltoy/` addresses:

Legacy address (still reachable)	New address (recommended)
`http://www.sagframe.com/schema/sqltoy/sqltoy.xsd`	`https://sagframe.github.io/schema/sqltoy.xsd`
`http://www.sagframe.com/schema/sqltoy/sqltoy-translate.xsd`	`https://sagframe.github.io/schema/sqltoy-translate.xsd`
`http://www.sagframe.com/schema/sqltoy/quickvo.xsd`	`https://sagframe.github.io/schema/quickvo.xsd`

For existing files:

- **No runtime impact**: the framework never reads the XSD, so files carrying the legacy address keep working as-is;
- **IDE experience only**: the legacy address is plain http and returns `application/octet-stream`; some network environments or IDEA versions may fail to fetch it, leaving files without completion;
- **Migrate gradually** to the new addresses — no big-bang change needed. The new address is https with a proper XML content type; one Alt+Enter fetch activates it.

Standard file headers (copy & paste)

`*.sql.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<sqltoy xmlns="http://www.sagframe.com/schema/sqltoy"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.sagframe.com/schema/sqltoy https://sagframe.github.io/schema/sqltoy.xsd">

</sqltoy>
```

`sqltoy-translate.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<sagacity xmlns="https://www.sagframe.com/schema/sqltoy-translate"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://www.sagframe.com/schema/sqltoy-translate https://sagframe.github.io/schema/sqltoy-translate.xsd">

</sagacity>
```

`quickvo.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<quickvo xmlns="http://www.sagframe.com/schema/quickvo"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.sagframe.com/schema/quickvo https://sagframe.github.io/schema/quickvo.xsd">

</quickvo>
```

Enabling completion & validation in IDEA

1. Open any config file, put the caret on the highlighted URI and press **Alt+Enter** → choose **Fetch external resource** (once is enough — IDEA caches it globally, no need to repeat per project);
2. From then on you get tag/attribute completion, and typos in tag or attribute names are highlighted immediately;
3. Alternatively, register a URI → XSD file mapping manually under `Settings | Languages & Frameworks | Schemas and DTDs`.

Related pages

- Dynamic SQL Guide: the full `<sql>` element and child-tag specification inside `*.sql.xml`
- Cache Translate in Action: translate.xml configuration and usage
- quickvo Code Generator: key quickvo.xml configuration

Object Operations

Object CRUD

sqltoy's object-oriented CRUD is quite similar to JPA. Its main features:

1. POJOs are generated via quickvo, a tool similar to hibernate-tools
2. Composite primary keys are supported, e.g. for update: lightDao.update(new StaffInfo("S0001","HUAWEI_SH").setName("Li Si"));
3. Two API modes are provided: explicit shortcut APIs and chainable APIs. The chain API exposes more parameters, such as specifying a datasource, enabling parallelism, setting batchSize, e.g. lightDao.update().dataSource(xxx).forceUpdateProps("status").batchSize(500).many(entities);
4. There is no concept of logical delete; in sqltoy it is simply a status update
5. Compared with JPA, sqltoy mainly optimizes update (providing the elastic update) and adds updateSaveFetch and updateFetch (one database interaction completes the locking query, update-or-insert and result return)
6. sqltoy supports cascading in two forms, OneToOne and OneToMany (with some improvements, e.g. loading supports filtering), but only one level of cascading; query results are automatically packaged in layers based on annotations (hierarchy packaging), see Query API
7. sqltoy optimizes cascade loading and cascade deletion under the hood, using id in (:ids) style queries and deletes, organizing the final data algorithmically to reduce database IO and improve efficiency
8. For details see the org.sagacity.sqltoy.dao.LightDao interface, which is thoroughly documented

1. Save a Single Record

- API signature; to explicitly specify a datasource, use the chainable operation lightDao.save().dataSource(xxx).one(entity)

```
/**
 * Save an object and return the primary key value
 * @param entity
 * @return Object returns the primary key value
 */
public Object save(Serializable entity);

/**
 * Chainable mode for saving a collection of objects, e.g.:
 *       lightDao.save().dataSource(xxx).one(entity)
 * @return
 */
public Save save();
```

- Simple example

```
StaffInfo staffInfo = new StaffInfo();
staffInfo.setStaffCode("S2018");
staffInfo.setStaffName("Test Employee 9");
lightDao.save(staffInfo);
```

- Cascade save example

```
// cascade configuration on the main object
@Schema(name="ComplexpkHead",description="complex primary-key cascade operation master table")
@Data
@Accessors(chain = true)
@Entity(tableName="sqltoy_complexpk_head",comment="complex primary-key cascade operation master table",pk_constraint="PRIMARY")
public class ComplexpkHead implements Serializable {

	/**
	 * Child table info associated by primary key
	 */
	@OneToMany(fields={"transDate","transCode"},mappedFields={"transDate","transId"},delete=true)
	private List<ComplexpkItem> complexpkItems=new ArrayList<ComplexpkItem>();

	// OneToOne configuration
	//@OneToOne(fields={"transDate","transCode"},mappedFields={"transDate","transId"},delete=true)
	//private ComplexpkItem complexpkItem;
}

@Test
public void testSaveCascade() {
	// master table record
	ComplexpkHead head = new ComplexpkHead();
	head.setTransDate(LocalDate.parse("2020-09-08"));
	head.setTransCode("S0001");
	head.setTotalCnt(BigDecimal.valueOf(10));
	head.setTotalAmt(BigDecimal.valueOf(10000));

	// child table record 1
	ComplexpkItem item1 = new ComplexpkItem();
	// the id is set manually here for easier demonstration
	item1.setId("S000101");
	item1.setProductId("P01");
	item1.setPrice(BigDecimal.valueOf(1000));
	item1.setAmt(BigDecimal.valueOf(5000));
	item1.setQuantity(BigDecimal.valueOf(5));
	head.getComplexpkItems().add(item1);

	// child table record 2
	ComplexpkItem item2 = new ComplexpkItem();
	item2.setId("S000102");
	item2.setProductId("P02");
	item2.setPrice(BigDecimal.valueOf(1000));
	item2.setAmt(BigDecimal.valueOf(5000));
	item2.setQuantity(BigDecimal.valueOf(5));
	head.getComplexpkItems().add(item2);

	lightDao.save(head);
}
```

2. Batch Save

- API signature; to explicitly specify a datasource, use the chainable operation lightDao.save().dataSource(xxx).many(entities)
- Large-scale parallel save: lightDao.save().parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).many(entities)

```
/**
 * Save objects in batch and return the number of records updated
 * @param <T>
 * @param entities
 * @return Long number of records changed in the database
 */
public <T extends Serializable> Long saveAll(List<T> entities);

/**
 * Save objects in batch, ignoring records that already exist
 * @param <T>
 * @param entities
 * @return Long number of records changed in the database
 */
public <T extends Serializable> Long saveAllIgnoreExist(List<T> entities);

/**
 * Chainable mode for saving a collection of objects, e.g.:
 *       lightDao.save().dataSource(xxx).many(entities)
         lightDao.save().parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).many(entities)
 * @return
 */
public Save save();
```

3. Update a Single Record

- API signature: unlike JPA, sqltoy updates do not require querying and loading the entity first; simply set the primary key and the property values to be modified

```
/**
 * Update data and return the number of records changed in the database (non-forced properties with null values do not participate in the update)
 * @param entity
 * @param forceUpdateProps properties to update forcibly
 * @return Long number of records changed in the database
 */
public Long update(Serializable entity, String... forceUpdateProps);

/**
 * Deep update: all fields are updated forcibly regardless of whether they are null
 * @param entity
 * @return Long number of records changed in the database
 */
public Long updateDeeply(Serializable entity);

/**
 * Chainable mode for updating records
 *       <li>lightDao.update().dataSource(xxx).forceUpdateProps("status").one(entity)</li>
 *       <li>lightDao.update().dataSource(xxx).forceUpdateProps("status").many(entities)</li>
 *       <li>Large-scale parallel:
lightDao.update().parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).forceUpdateProps("status").many(entities)</li>
 * @return
 */
public Update update();

/**
 * Cascade update data and return the number of records changed in the database
 * @param entity
 * @param forceUpdateProps
 * @param forceCascadeClasses when the cascade objects are null or empty, whether this means forcibly deleting the cascade records
 * @param subTableForceUpdateProps sets the properties to update forcibly on the cascade-updated objects
 * @return Long number of records changed in the database
 */
public Long updateCascade(Serializable entity, String[] forceUpdateProps, Class[] forceCascadeClasses,
              HashMap<Class, String[]> subTableForceUpdateProps);
```

- Simple update example

```
// update the name and status of the employee with staff code S0001
lightDao.update(new StaffInfo("S0001").setName("Zhang San").setStatus(1));
// composite primary key (staff code, tenant)
lightDao.update(new StaffInfo("S0001","HUAWEI_SH").setName("Zhang San").setStatus(1));
```

- Cascade update example

```
Map<Class,String[]> subTableForces=new HashMap<>();
subTableForces.put(ComplexpkItem.class,new String[]{"status"});

// for single-record updates, once cascadeForceUpdate or cascadeClasses is involved, cascade behavior is enabled; otherwise update only targets the
object itself
lightDao.update().dataSource(crmDataSource).forceUpdateProps("status").cascadeForceUpdate(subTableForces).one(entity);
```

4. Batch Update

- API signatures

```
/**
 * Batch update, with optional properties to update forcibly (non-forced properties with null values do not participate in the update)
 * @param <T>
 * @param entities
 * @param forceUpdateProps
 * @return Long number of records changed in the database
 */
public <T extends Serializable> Long updateAll(List<T> entities, String... forceUpdateProps);

/**
 * Batch deep update, i.e. all fields participate in the update (including null properties)
 * @param <T>
 * @param entities
 * @return Long number of records changed in the database
 */
public <T extends Serializable> Long updateAllDeeply(List<T> entities);
```

- Large-scale parallel update example

```
lightDao.update().dataSource(crmDataSource).forceUpdateProps("status")
      .parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10))
      .many(entities);
```

5. Update by Query Conditions (updateByQuery)

- Description: no need to load entities first — fields are batch-updated directly based on single-table query conditions, completed in a single database interaction. Suitable for scenarios like "uniformly updating certain fields of all records matching a condition" (logical delete is essentially a status update and can be implemented with it).
- API signature

```
/**
 * Update data based on a single-table query
 * @param entityClass entity class
 * @param entityUpdate organizes the set fields and where conditions via EntityUpdate
 * @return Long number of records changed in the database
 */
public Long updateByQuery(Class entityClass, EntityUpdate entityUpdate);
```

- Usage example

```
// change the gender of employee S0001 to F (set uses object property names; where uses ? placeholders with values passed in order)
lightDao.updateByQuery(StaffInfo.class,
        EntityUpdate.create().set("sexType", "F").where("staffId=?").values("S0001"));

// multi-field update + multiple conditions
lightDao.updateByQuery(OrderInfo.class,
        EntityUpdate.create()
            .set("status", 2)
            .set("updateTime", LocalDateTime.now())
            .where("organId=? and createTime<?")
            .values("T001", LocalDateTime.now().minusMonths(3)));

// specify the datasource; skipNotExistColumn() means set fields that do not exist in the table are skipped automatically
lightDao.updateByQuery(StaffInfo.class,
        EntityUpdate.create().dataSource(crmDataSource)
            .set("status", 0).where("staffId=?").values("S0001")
            .skipNotExistColumn());
```

> `EntityUpdate` chainable methods: `set(property, value)` (callable multiple times to set multiple fields), `where(condition)`, `values(values...)`, `blankToNull(Boolean)`, `skipNotExistColumn()`, `dataSource(...)`, `showSql(Boolean)`. Use `?` placeholders in `where`, with values passed in order via `values(...)`. It pairs with `deleteByQuery` (see the next section): one updates by condition, the other deletes by condition.

- Column self-increment / expression updates (`set field=field+?`)

Besides plain field names, the `set` key can also be written as `field=expression` to perform calculated updates based on a field's current value (self-increment, deduction, etc.). The `?` in the expression is bound by the `set` value; field names (e.g. `totalAmt`) are automatically converted to database column names (`total_amt`).

```
// self update: total_amt = total_amt + 10
// equivalent SQL: update sqltoy_staff_info set total_amt=total_amt+10 where staff_name like 'Zhang'
lightDao.updateByQuery(StaffInfo.class,
        EntityUpdate.create()
            .set("totalAmt=totalAmt+?", 10)
            .where("staffName like ?").values("Zhang"));

// stock deduction: quantity = quantity - 5; plain field updates can be combined at the same time
lightDao.updateByQuery(OrderInfo.class,
        EntityUpdate.create()
            .set("quantity=quantity-?", 5)     // expression update (calculated from the field's current value)
            .set("status", 2)                  // plain field update (they can coexist)
            .where("orderId=?").values("10001"));
```

> Key point: always use `?` placeholders in expressions and pass the value through the `set` value (e.g. `set("totalAmt=totalAmt+?", 10)`); in a `set` key the left side is the field name and the right side is a SQL expression, which may include operators such as `+ - * /` . Expression updates can be mixed with plain `set(field, value)` ; the framework binds parameters automatically in the order "set first, then where".

- In practice: logical delete (status update)

sqltoy has no standalone "logical delete" concept — a logical delete is essentially just updating a status/flag field (see the "Delete Operations" section below). For a single record the elastic update is enough; for batch by condition use `updateByQuery`, done in a single database interaction.

```
// Option 1: single-record logical delete — elastic update, only status is changed (null fields do not participate in the update)
lightDao.update(new OrderInfo("10001").setStatus(0));

// Option 2: batch logical delete by condition — updateByQuery
// mark orders of a department older than 3 months with status 1 as deleted (0), recording the operator/time
lightDao.updateByQuery(OrderInfo.class,
        EntityUpdate.create()
            .set("status", 0)
            .set("updateBy", "S0001")
            .set("updateTime", LocalDateTime.now())
            .where("organId=? and status=? and createTime<?")
            .values("T001", 1, LocalDateTime.now().minusMonths(3)));
```

Exclude logically deleted records when querying (just add the status condition in the SQL):

```
<sql id="find_order">
    <value><![CDATA[
        select * from sqltoy_order_info t
        where t.status<>0          -- exclude logically deleted
        #[and t.organ_id=:organId]
    ]]></value>
</sql>
```

> Tip: if a unified field handler (`unifyFieldsHandler`) is configured, object-style `update` automatically fills in missing `updateBy` / `updateTime` ; with `updateByQuery` it is recommended to explicitly `set` the audit fields as shown above.

6. Delete Operations

- sqltoy has no concept of logical delete
  (a logical delete is essentially updating a status or flag field; use update operations instead)
- API signatures

```
/**
 * Chainable mode for delete operations, examples:
 *       <li>lightDao.delete().dataSource(xxxx).one(entity);</li>
 *       <li>lightDao.delete().batchSize(1000).autoCommit(true).many(entities);</li>
 *       <li>lightDao.delete().parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).many(entities);</li>
 * @return
 */
public Delete delete();

/**
 * Delete a single object and return the number of records affected in the database
 * @param entity
 * @return Long number of records changed in the database (number of deleted records)
 */
public Long delete(final Serializable entity);

/**
 * Delete objects in batch and return the number of records affected in the database
 * @param entities
 * @return Long number of records changed in the database (number of deleted records)
 */
public <T extends Serializable> Long deleteAll(final List<T> entities);

/**
 * Batch delete by a collection of ids
 * @param entityClass
 * @param ids
 * @return
 */
public Long deleteByIds(Class entityClass, Object... ids);

/**
 * Delete based on a single-table query, providing a shortcut operation in code
 * @param entityClass
 * @param entityQuery e.g.: lightDao.deleteByQuery(DictDetail.class,EntityQuery.create().where("status=?").values(0));
 * @return Long number of records changed in the database (number of inserted records)
 */
public Long deleteByQuery(Class entityClass, EntityQuery entityQuery);
```

- Usage example

```
// composite primary key
lightDao.delete(new StaffInfo("S0001","HUAWEI_SH"));
// single primary key, delete multiple records
lightDao.deleteByIds(OrderInfo.class,"10001","10002");
// delete data matching the conditions
lightDao.deleteByQuery(DictDetail.class,EntityQuery.create().dataSource(xxxx).where("status=?").values(0));
```

7. saveOrUpdate Operations

- Description
  1. The saveOrUpdate logic: if the record exists it is updated; if it does not exist it is created (a null primary key value always means creation)
  2. MySQL-family databases use update first then insert ignore; others use the merge into mode
- API signatures

```
/**
 * Chainable mode for saving a collection of objects, e.g.:
 *    <li>lightDao.save().dataSource(xxx).saveMode(SaveMode.UPDATE).many(entities)</li>
 * @return
 */
public Save save();

/**
 * Save or update data and return the number of records changed in the database
 * @param entity
 * @param forceUpdateProps fields to update forcibly
 * @return Long number of records changed in the database
 */
public Long saveOrUpdate(Serializable entity, String... forceUpdateProps);

/**
 * Batch save or update (updates when the record already exists)
 * @param <T>
 * @param entities
 * @param forceUpdateProps fields to update forcibly
 * @return Long number of records changed in the database
 */
public <T extends Serializable> Long saveOrUpdateAll(List<T> entities, String... forceUpdateProps);
```

- Usage example

```
// simple batch save or update
lightDao.saveOrUpdateAll(entities,"name","status","quantity");

// SaveMode has three values: APPEND\UPDATE\IGNORE
lightDao.save().dataSource(xxx).saveMode(SaveMode.UPDATE)
       .parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).many(entities);
```

8. Object Loading

- API signatures

```
/**
 * Chainable mode for object loading operations
 *    <li>lightDao.load().dataSource(xxxx).lock(LockMode.UPGRADE_NOWAIT).one(new StaffInfo("S0001"));</li>
 *    <li>lightDao.load().parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).many(entities);</li>
 *    <li>Load the main objects and cascade-load the child objects: lightDao.load().cascade(OrderItem.class,OrderDeliveryPlan.class).many(entities);
</li>
 *    <li>Cascade-load the child objects only by the main objects' primary keys:
lightDao.load().cascade(OrderItem.class,OrderDeliveryPlan.class).onlyCascade().many(entities);</li>
 * @return
 */
public Load load();

/**
 * Load the details of an object by its primary key value
 * @param entity
 * @return entity
 */
public <T extends Serializable> T load(final T entity);

/**
 * Load a single entity object by primary key
 * @param <T>
 * @param entityClass
 * @param id
 * @return
 */
public <T extends Serializable> T loadById(final Class<T> entityClass, Object id);

/**
 * Load an object by primary key, with a read lock setting
 * @param entity
 * @param lockMode LockMode.UPGRADE or LockMode.UPGRADE_NOWAIT, etc.
 * @return entity
 */
public <T extends Serializable> T load(final T entity, final LockMode lockMode);

/**
 * Load an object while specifying the child classes to load, achieving cascade loading
 * @param entity
 * @param lockMode
 * @param cascadeTypes
 * @return entity
 */
public <T extends Serializable> T loadCascade(final T entity, final LockMode lockMode, final Class... cascadeTypes);

/**
 * Load entity details by the primary keys in the collection (optimized under the hood with batch loading, while handling the in-1000 issue)
 * @param entities
 * @return entities
 */
public <T extends Serializable> List<T> loadAll(List<T> entities);

/**
 * Batch loading with record locking
 * @param <T>
 * @param entities
 * @param lockMode
 * @return
 */
public <T extends Serializable> List<T> loadAll(List<T> entities, final LockMode lockMode);

/**
 * Batch load objects by a collection of ids
 * @param <T>
 * @param entityClass
 * @param ids
 * @return
 */
public <T extends Serializable> List<T> loadByIds(final Class<T> entityClass, Object... ids);

/**
 * Batch load objects by a collection of ids, with locking
 * @param <T>
 * @param entityClass
 * @param lockMode
 * @param ids
 * @return
 */
```

```
public <T extends Serializable> List<T> loadByIds(final Class<T> entityClass, final LockMode lockMode,
              Object... ids);

/**
 * Selectively load child table information
 * @param entities
 * @param cascadeTypes
 * @return
 */
public <T extends Serializable> List<T> loadAllCascade(List<T> entities, final Class... cascadeTypes);

/**
 * Lock the main table records and cascade-load the child table data
 * @param <T>
 * @param entities
 * @param lockMode
 * @param cascadeTypes
 * @return
 */
public <T extends Serializable> List<T> loadAllCascade(List<T> entities, final LockMode lockMode,
              final Class... cascadeTypes);
```

- Large-scale parallel loading example

```
lightDao.load().parallelConfig(ParallelConfig.create().groupSize(5000).maxThreads(10)).many(entities);
```

- Cascade loading

```
lightDao.load().cascade(OrderItem.class,OrderDeliveryPlan.class).many(entities);

lightDao.loadCascade(new OrderInfo("S0001"), null, OrderItem.class,OrderDeliveryPlan.class);
```

- Cascade-load the child objects only (child objects are constructed onto the corresponding properties of the main objects)

```
// .onlyCascade() queries the child objects only via the primary keys of the pojos in the entities collection; the main objects are no longer queried
lightDao.load().cascade(OrderItem.class,OrderDeliveryPlan.class).onlyCascade().many(entities);
```

9. updateSaveFetch (locking query → update if exists / insert if not → return the result, in one interaction)

- Description
  1. Unique to sqltoy (JPA does not have it): **a single database interaction** completes the locking query + update-if-exists / insert-if-not-exists + returning the updated or inserted record
  2. Applicable scenarios: high-concurrency, strongly transactional ledgers such as inventory and customer fund accounts — the "update if exists, initialize-and-insert if not" upsert semantics naturally avoids the concurrent first-query race (when two requests both find the record missing, only one insert succeeds; the other falls back to updating)
  3. **Prefer it over updateFetch for such scenarios**: it locates the record by entity + primary/unique key, no SQL needed; the insert happens automatically when the record does not exist, so callers never check existence first. updateFetch (next section) targets scenarios where the record **must already exist** and the query conditions and lock strategy need to be freely authored
- API (three common overloads; uniqueProps can be omitted to query by primary key)

```
/**
 * Designed for high-concurrency, strongly transactional scenarios such as inventory ledgers and customer fund accounts; a single database
interaction accomplishes:
 * 1. locking query; 2. update if the record exists; 3. insert if the record does not exist; 4. return the updated or inserted record information
 */
public <T extends Serializable> T updateSaveFetch(final T entity, final UpdateRowHandler updateRowHandler,
        final String... uniqueProps);

// entity-callback form (recommended): inside the callback you only touch entity getters/setters, no ResultSet
public <T extends Serializable> T updateSaveFetch(final T entity, final EntityUpdateCallback<T> callback,
        final String... uniqueProps);

// form with lock wait timeout (in seconds)
public <T extends Serializable> T updateSaveFetch(final T entity, final EntityUpdateCallback<T> callback,
        final int lockWaitTimeout, final String... uniqueProps);
```

- Usage example

```
// add the dependency class (the entity-callback form relies on byte-buddy to back the proxied entity)
<dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.18.15</version>
</dependency>
```

```
@Test
public void testUpdateSaveFetch() {
    StaffInfoVO staffInfo = new StaffInfoVO();
    staffInfo.setStaffId("S0001");
    staffInfo.setBeginDate(LocalDate.parse("2019-01-01"));
    staffInfo.setEndDate(LocalDate.now());
    staffInfo.setStaffName("Chen");
    lightDao.updateSaveFetch(staffInfo, (entity, rowIndex) -> {
        // the entity getter here actually goes through a proxy, reading the value via rs.getString("tel_no")
        String telNo = entity.getTelNo();
        if (telNo != null) {
            // the setter here actually maps to rs.updateString("tel_no", telNo.substring(0, 3) + "#**#" +
            // telNo.substring(7));
            entity.setTelNo(telNo.substring(0, 3) + "#**#" + telNo.substring(7));
        }
    });
}
```

> [!NOTE] The `entity` inside the `EntityUpdateCallback` is a **proxied object**: getters read the current database value (`rs.getXxx`) and setters update the row directly (`rs.updateXxx`) — no manual ResultSet synchronization. Throwing inside the callback rolls the whole operation back.

- Notes
    1. Avoid identity / sequence primary keys on the entity (when the record does not exist and an insert happens, carrying the key in the entity is the most straightforward)
    2. For business-unique-key scenarios (e.g. a ledger located by "product + date" rather than primary key), pass the property names via `uniqueProps` ; omit it to query by primary key
    3. Under fierce contention prefer the `lockWaitTimeout` overload to set the lock wait explicitly (in seconds); a lock wait timeout throws and rolls back
    4. Under GraalVM native images the `EntityUpdateCallback` form relies on byte-buddy proxy subclasses: configure `native-entity-proxies` to pre-generate them at build time (entities not listed are unusable; the insert branch is not affected); see the GraalVM Native guide

10. updateFetch (SQL-authored locking query → validate and update → return results, in one interaction)

- Description
    1. Unique to sqltoy (JPA does not have it): **a single database interaction** completes query + locking + logical validation + update + returning the updated results
    2. Applicable scenarios: strongly transactional steps where the record **must already exist** and the query conditions need to be freely authored (multi-table joins, complex conditions, explicit lock strategy) — the purpose of the query is to **lock the records for logical validation** (e.g. the value must not drop below 0 after deduction); the update happens only after validation passes
    3. For ledger scenarios where the record may not exist and is located by primary/unique key, prefer `updateSaveFetch` above (automatic upsert)

> [!WARNING] The core purpose of updateFetch is **data logical validation** (e.g. the value must not drop below 0 after deduction). **Do not use this method for display-oriented page queries.**

- API

```
/**
 * Fetch and lock data, then update it (usually a simple query: lock the records, validate logically, then update)
 * @param queryExecutor    query executor, defining the query SQL, condition parameters and lock strategy
 * @param updateRowHandler row update callback handler that validates and modifies the row data, then submits the update
 * @return the updated row records
 */
public List updateFetch(final QueryExecutor queryExecutor, final UpdateRowHandler updateRowHandler);
```

- Usage example (stock deduction: lock → validate sufficient stock → deduct → return the latest data)

```
List result = lightDao.updateFetch(
        // simple query + pessimistic lock strategy
        new QueryExecutor("sqltoy_stock_find")
                .names("productId").values("P0001")
                .lock(LockMode.UPGRADE),
        // Note: UpdateRowHandler's methods are all default; it is not a functional interface, so override via an anonymous class
        new UpdateRowHandler() {
            @Override
            public void updateRow(ResultSet rs, int index, BiConsumer<String, Object> setVal) throws Exception {
                double quantity = rs.getDouble("quantity");
                if (quantity < 1) {
                    // if logical validation fails, throw directly; the whole update rolls back
                    throw new IllegalArgumentException("Insufficient stock!");
                }
                // set the updated value via the callback: <property name, new value>
                setVal.accept("quantity", quantity - 1);
            }
        });
```

> `UpdateRowHandler` provides three overridable methods: `updateRow(rs,index)`, `updateRow(rs,index,BiConsumer<property,value>)`, `updateRow(rs,index,ThreeBiConsumer<property,forced update fields[],value>)` (the third one allows specifying forced-update fields).
>
> It complements `updateSaveFetch` in the previous section: `updateSaveFetch` locates the record by entity and inserts automatically when missing; `updateFetch` updates **existing** records, with query conditions freely authored in SQL.

Tree-Table Route Builder

For **tree tables** such as organization structures, administrative divisions and product categories, a node route field (e.g. `node_route`) is usually kept redundantly so that an entire subtree can be fetched **non-recursively** with `like 'route%'`, avoiding the database pressure caused by layer-by-layer recursive queries. `wrapTreeTableRoute` builds and maintains these redundant fields: **node route (nodeRoute), hierarchy level (nodeLevel), leaf flag (leaf)**.

1. API

```
/**
 * Builds the required information of a tree table, such as the node route, hierarchy level and leaf flag
 * @param treeTableModel tree table model object, providing the hierarchical structure configuration such as the id field and the pid parent field
 * @return returns true on success, false on failure
 */
public boolean wrapTreeTableRoute(final TreeTableModel treeTableModel);
```

Equivalent chainable form: `lightDao.treeTable().treeModel(treeTableModel).submit()`

2. TreeTableModel Configuration

Two construction modes: **entity-based** (the table name is taken from the `@Entity` annotation) or **table-name-based** full-argument construction `(tableName, pidValue, idField, pidField, nodeRouteField, nodeLevelField, leafField, isChar, idLength)`.

Configuration	Description
`idField` / `pidField`	primary key field / parent field
`rootId` (`pidValue`)	parent value of the root node (e.g. `0`)
`nodeRouteField`	node route field, in the form `,100008,100009,`
`nodeLevelField`	hierarchy level field
`isLeafField` (`leafField`)	leaf flag field
`idLength` / `idTypeIsChar`	id length / whether character type (for fixed-length hierarchical concatenation)
`setConditions`	additional filter conditions
`appendZero`	whether to pad with zeros
`splitSign`	route separator

3. Usage Example

```
// Mode 1: entity object based (the table name is taken from the @Entity annotation)
lightDao.wrapTreeTableRoute(new TreeTableModel(organInfo)
        .idField("organCode").pidField("organPid")
        .nodeRouteField("nodeRoute").nodeLevelField("nodeLevel")
        .isLeafField("isLeaf"));

// Mode 2: table-name-based full-argument construction
// (tableName, root parent value pidValue, idField, pidField, nodeRouteField, nodeLevelField, leafField, isChar, idLength)
lightDao.wrapTreeTableRoute(new TreeTableModel("sqltoy_organ_info", "0",
        "organ_id", "organ_pid", "node_route", "node_level", "leaf_field", true, 4));
```

Once built, the route field can be used to efficiently query an entire subtree:

```
<sql id="sqltoy_treeTable_search">
    <value><![CDATA[
        select * from sqltoy_organ_info t
        where t.node_route like :nodeRoute
    ]]></value>
</sql>
```

```
// query all children under a node (nodeRoute in the form ",100008,%" — the trailing % matches the whole subtree)
List<OrganInfoVO> subOrgans = lightDao.find("sqltoy_treeTable_search",
        MapKit.map("nodeRoute", ",100008,%"), OrganInfoVO.class);
```

4. Notes

- When a unified field handler (`unifyFieldsHandler`) is configured, the route cascade and leaf-flag update statements **automatically append** the common update fields from `updateUnifyFields()` (e.g. last modified by, last modified time); columns are matched by entity property names, and fields not present in the entity are ignored automatically.
- For the **query-side sorting and layer-by-layer summary** of tree tables (the `<tree-sort>` tag), see Complex Query & Analytics; combining the two yields a complete tree solution of "route building + sorting and summary".
- For declaring object cascade relationships (`@OneToMany` / `@OneToOne`), see Object CRUD.

SQL Query

Dynamic SQL Guide

sqltoy's dynamic SQL design goal: **keep SQL clean and directly runnable in a database client**, while stripping out of the SQL the dynamic logic (such as `<if>`) that mybatis intermixes into it.

- SQL can be written in `*.sql.xml` files (recommended: easy to maintain, supports hot reloading), passed in directly from Java code, or expressed as conditions with the Lambda condition API (built in since 6.0.8; the former third-party sqltoy-plus has been merged into core).
- For simple scenarios, use the explicit `lightDao` interfaces directly; for complex ones, organize parameters with `findByQuery` / `findPageByQuery` together with `QueryExecutor` or `EntityQuery`. All explicit interfaces are built on `QueryExecutor` underneath — understand it once and you can use them all flexibly.

---

1. How the `#[]` Condition Fragment Works

sqltoy uses `#[ ... ]` to mark a **dynamic condition fragment**. The core rules:

1. `#[ ... ]` marks the condition inside as dynamic; **required conditions do not need** `#[]`.
2. If **any parameter inside the fragment is null (or blank)**, the entire fragment is removed automatically.
3. **Multi-level nesting** is supported: `#[t.type=:type #[and t.amt>:amt and t.quantity>:quantity]]`.
   - **Fragments are dropped level by level, independently**: each `#[...]` looks only at its own direct parameters. When an inner parameter is null, only the inner fragment is removed while the outer fragment and its parentheses survive:

     ```
     where t.type=:type #[and (t.amt>:amt #[or t.qty>:qty])]
     -- amt=100, qty=null → where t.type=:type and (t.amt>:amt)
     -- amt=100, qty=5    → where t.type=:type and (t.amt>:amt or t.qty>:qty)
     ```

   - **When an outer fragment's own direct parameter is null, the whole outer fragment is removed together with everything nested inside** — even if the inner parameters all have values. Nesting therefore naturally expresses **hierarchical / master-detail conditions**: the outer level is the precondition for the inner level to participate. The same applies recursively at three or more levels:

     ```
     where #[t.a=:a #[and t.b=:b #[and t.c=:c]]]
     -- b null → where t.a=?
     ```

   - **Parameters work exactly as in a single fragment**; all of the following forms apply inside nested levels: named parameters `:paramName`; array indexing where elements of the same array are referenced separately (e.g. `:starEndTime[0]` and `:starEndTime[1]` forming a date range); tuple multi-parameters `(t.tech_group,t.prod_group) in (:techGroups,:prodGroups)`; collection expansion `t.id in (:ids)` (arrays/collections expand into multiple placeholders). The same parameter may appear at multiple levels, each level checking null on its own.
   - **Parenthesized groups and multiple members**: several `#[...]` members inside `and ( ... )` are dropped independently (e.g. `#[and ( #[t.a=:a] or #[t.b=:b] )]`) — a member survives as long as its own parameters have values, and the whole group is removed when every member is null. Dangling `and` / `or` connectors left behind by dropped members are cleaned automatically (since 6.0.7; a connector right after `(` or right before `)` is stripped, while legitimate keywords such as `( order by ... )` are unaffected), so no manual workaround is needed.
4. When a required condition (without `#[]`) gets a null parameter, it is converted automatically to `is null` / `is not null`.
5. sqltoy enables `blank-to-null="true"` by default: blank strings are treated as null automatically.
6. `#` happens to be the comment marker in MySQL and similar databases, so **SQL containing `#[]` can be copied straight into a client and executed**, which makes debugging easy.

---

2. The XML Form

> [!TIP] For the XSD schema location of `*.sql.xml` files and IDE completion setup, see the XML Schema Guide.

```
<!-- blank-to-null="true" is the default; equivalent to <sql id="show_case" blank-to-null="true"> -->
<sql id="show_case">
    <filters>
        <!-- Pre-normalize condition parameters via filters -->
        <eq params="status" value="-1" />
        <!-- Two ways to turn off blank-to-null:
             1. blank-to-null="false"
             2. Apply the blank tag once to any parameter (including nonexistent ones), which switches off the default automatically -->
        <blank params="anyParamNameIncludeNotExist"/>
    </filters>
    <value><![CDATA[
        select *
        from sqltoy_device_order_info t
        where
              -- when orderType is null it converts automatically to t.order_type is null
              t.order_type=:orderType
              and t.ORGAN_ID in (:authedOrganIds)
              #[and t.status=:status]
              #[and t.ORDER_ID=:orderId]
              #[and t.TRANS_DATE>=:beginAndEndDate[0]]
              #[and t.TRANS_DATE<:beginAndEndDate[1]]
              #[and (t.TECH_GROUP,t.PROD_GROUP) in (:techGroups,:prodGroups)]
    ]]></value>
</sql>
```

Common attributes of the `<sql>` tag: `id` (required), `type` (`search` / `insert` / `update` / `delete` ; `select` is a legacy-compatible alias), `blank-to-null` (default `true`), `debug` (whether to output execution logs), `query-timeout` (query timeout in seconds), `dataSource` (specifies a data source name), `dialect` (new in 6.0.3 — declares that this sql is parsed and solidified in the specified dialect form; see Dialect Self-Adaption & Function Extension), `union-all-count` (in union pagination, computes the count branch-by-branch to avoid unnecessary execution). For NoSQL scenarios use the `eql` (Elasticsearch) / `mql` (MongoDB) nodes instead — their attributes differ, see Elasticsearch and MongoDB.

---

3. The Java Form (Anything XML Can Do, Code Can Do Too)

```
String whereString = """
        orderType=:orderType
        and organId in (:authedOrganIds)
        #[and status=:status]
        #[and orderId=:orderId]
        #[and transDate>=:beginAndEndDate[0]]
        #[and transDate<:beginAndEndDate[1]]
        #[and (techGroup,prodGroup) in (:techGroups,:prodGroups)]
        """;
List result = lightDao.findEntity(DeviceOrderVO.class, EntityQuery.create()
        // .blankNotNull()  // turn off blank-to-null
        .where(whereString)
        .values(MapKit.keys("orderType", "authedOrganIds", "status", "orderId", "beginAndEndDate", "techGroups", "prodGroups")
                .values("PO", authedOrganIds, "1", "S0001", beginAndEndTime, techGroups, prodGroups))
        .filters(new ParamsFilter("status").eq("-1")));
```

---

4. Complete Reference for the `filters` Parameter Filters

`<filters>` pre-normalizes condition parameters before the SQL executes (to-null conversion, type conversion, formatting, reverse cache matching, etc.) and is the core of sqltoy's dynamic SQL. In Java it corresponds to `ParamsFilter` , passed in via `.filters(...)` .

> **Design philosophy — pre-normalization**: this is where sqltoy's simplicity comes from — filters trim condition parameters into shape before they enter the SQL, **keeping interference with the SQL body itself to a minimum**. `#[]` retains only one intrinsic form — "if the parameter has a value, splice it in" — while all other condition logic (setting null, formatting, splitting/merging) is moved up front into filters, so the SQL always keeps an intuitive, concise, native structure.

4.1 To-Null Filters (set the parameter to null when the condition is met, thereby removing the corresponding `#[]` fragment)

Tag	Attributes	Description
`<blank>`	`params` (default `*`), `excludes`	Converts the parameter to null if it is blank
`<eq>`	`params`, `value`, `excludes`	Converts the parameter to null if it equals the given value (a match on any value within an array converts)
`<neq>`	`params`, `value`	Converts the parameter to null if it does not equal the given value
`<gt>` / `<gte>`	`params`, `value`	Converts to null if the parameter is greater than / greater than or equal to the given value
`<lt>` / `<lte>`	`params`, `value`	Converts to null if the parameter is less than / less than or equal to the given value
`<between>`	`params`, `start-value`, `end-value`, `excludes`	Converts to null if the parameter falls within the range

4.2 LIKE Fuzzy-Match Handling

Tag	Attributes	Description
`<l-like>`	`params`, `append-str`	Appends `%` on the right when the parameter is non-blank (i.e. `xxx%`)
`<r-like>`	`params`, `append-str`	Prepends `%` on the left when the parameter is non-blank (i.e. `%xxx`)
`<escapeLike>`	`params`	Escapes special characters in like parameters

4.3 Type Conversion and Formatting

Tag	Attributes	Description
`<to-number>`	`params`, `data-type` (decimal/integer/double/float/long)	Converts the parameter to a numeric type
`<to-array>`	`params`, `data-type` (string/decimal/...)	Converts the parameter to an array (a single value becomes an array of length 1)
`<to-string>`	`params`, `add-quote` (none/single/double)	Converts the parameter to a string, optionally quoted
`<to-date>`	`params`, `format`, `type`, `increment-time`, `increment-unit`	Converts to a date; `format` supports `yyyy-MM-dd`, `FIRST_OF_MONTH`, `LAST_OF_YEAR`, `FIRST_OF_WEEK`, etc.; `increment-time` accepts dynamic parameters such as `${incrementDays}`
`<split>`	`params`, `split-sign` (default `,`), `data-type`	Splits a concatenated string into an array (commonly used for in conditions)
`<default>`	`params`, `value`, `data-type`, `split-sign`, `is-array`	Sets a default value for the parameter; `value` supports expressions such as `sysdate()-2d`, `first_of_month`
`<replace>`	`params`, `regex`, `value`, `is-first`	Regex replacement on the parameter
`<to-in-arg>`	`params`, `single-quote` (default true)	Concatenates an array into a single string used as the value of an `in (:args)` condition
`<remove-null>`	`params`, `remove-blank` (default true)	Removes null or blank entries from arrays/collections
`<date-format>`	`params`, `format`	Formats a date parameter (rarely used)

4.4 Advanced Filters

Tag	Attributes	Description
`<primary>`	`param`, `excludes`	**Primary parameter**: e.g. when an order number has a value it signals a precise query; apart from permission conditions excluded via `excludes`, no other conditions participate in filtering
`<exclusive>`	`param`, `set-params`, `compare-type`, `compare-values`, `set-value`	Mutually exclusive parameter handling
`<clone>`	`param`, `as-param`	Copies a parameter's value and stores it as another parameter
`<cache-arg>`	`param`, `cache-name`, `cache-key-index`, `cache-mapping-indexes`, `cache-mapping-max`, `alias-name`, `unmatched-return-self`, etc.	**Reverse cache matching**: matches a name into an array of corresponding codes (e.g. inputting "Zhang Ming" matches out the array of employee IDs containing that name) which then participates in the query, replacing inefficient like. May contain the `<filter cache-index compare-param compare-type split-sign>` child tag
`<custom-handler>`	`params`, `type`	Custom data handler
`<valid-sqlInjection>`	`params`, `level` (STRICT_WORD/RELAXED_WORD/SQL_KEYWORD)	SQL injection parameter validation

> **Reverse cache matching (cache-arg)** is a signature sqltoy capability; see Cache Translate in Action and Large-Scale Cache (FIFO).

4.5 Complete Example of Parameter Forms

A single `<sql>` that condenses all parameter forms sqltoy supports (single value, collection expansion, array index range, tuple multi-parameter, object property path, hierarchical nesting, bracket groups), combined with filters preprocessing, forming the typical business shape of "**query conditions separated from the concrete SQL**":

```
<!-- 1. condition-value handling is separated from the concrete SQL -->
<!-- 2. conditions are preprocessed up-front via the generic filters methods (most need no extra handling) -->
<sql id="show_case">
    <filters>
        <!-- as long as statusAry contains -1 (meaning "all"), set it to null so the condition is skipped -->
        <eq params="statusAry" value="-1" />
    </filters>
    <value><![CDATA[
        select t.*
        from sqltoy_device_order_info t
        where #[t.status in (:statusAry)]                              -- collection auto expansion
        #[and t.ORDER_ID=:orderId]                                     -- single value
        #[and t.ORGAN_ID in (:authedOrganIds)]                         -- collection (data permission)
        #[and t.STAFF_ID in (:staffIds)]                               -- collection (multi-select)
        #[and t.TRANS_DATE>=:beginAndEndDate[0]]                       -- array index, first element
        #[and t.TRANS_DATE<:beginAndEndDate[1]]                        -- array index, second element (time range)
        #[and (t.tech_group,t.prod_group) in (:techGroups,:prodGroups)] -- tuple multi-parameter
        #[and t.staff_name like :staffName]                            -- fuzzy match
        #[and t.creator_id=:xxxVo.creatorId]                           -- object property path (single-object passing, see below)
        #[and t.priority=:priority #[and t.level>=:level]]             -- hierarchical nesting: outer null drops inner too
        #[and ( #[t.order_id=:orderId2] or #[t.biz_no=:bizNo] )]       -- bracket group: members judge null independently, dangling or auto-cleaned
    ]]></value>
</sql>
```

Parameter forms at a glance:

Form	Syntax	Description
Single value	`:orderId`	Binds a single value
Collection / array expansion	`t.status in (:statusAry)`	Arrays / collections automatically expand into multiple `?` placeholders
Array index	`:beginAndEndDate[0]` / `:beginAndEndDate[1]`	Refer to elements of the same array separately, e.g. building a time range
Tuple multi-parameter	`(t.tech_group,t.prod_group) in (:techGroups,:prodGroups)`	Two collections pair by position into `(a1,b1),(a2,b2)`
Fuzzy match	`t.staff_name like :staffName`	Can combine with the filters' `l-like` / `r-like` / `escapeLike`
Object property path	`:xxxVo.creatorId`	**Single-object passing**: values carries only one VO/Map object; the SQL parameter name is the object property, resolved by reflection; supports nested property cascading `xxxVo.name` and `a.b[index]` (case-insensitive; matches `BeanUtil.reflectBeanToAry` semantics; also used by `updateByQuery` etc.)
Hierarchical nesting	`#[t.priority=:priority #[and t.level>=:level]]`	Each level judges null independently; an outer null drops the inner level too
Bracket group	`#[and ( #[t.order_id=:orderId2] or #[t.biz_no=:bizNo] )]`	Members judge null independently; when all are null the whole group is dropped and dangling `and` / `or` are cleaned automatically

Passing parameters from Java (matching `names` order one-to-one; pass null for parameters not needed):

```
// Style 1: multi-value passing (names and values map one-to-one)
List rows = lightDao.findByQuery(
        new QueryExecutor("show_case")
            .names("statusAry", "orderId", "authedOrganIds", "staffIds", "beginAndEndDate",
                   "techGroups", "prodGroups", "staffName", "priority", "level",
                   "orderId2", "bizNo")
            .values(
                    new Integer[] { 1, 2, -1 },        // statusAry: contains -1 -> eq sets null, segment dropped
                    "O0001",                            // orderId: single value
                    new String[] { "A01", "A02" },      // authedOrganIds
                    new String[] { "S001", "S002" },    // staffIds
                    new LocalDateTime[] { start, end }, // beginAndEndDate: range
                    new String[] { "T001", "T002" },    // techGroups
                    new String[] { "P001", "P002" },    // prodGroups
                    "%张%",                             // staffName
                    1, 2,                               // priority, level (hierarchical nesting)
                    null, "BZ2026001")                  // orderId2=null -> bracket group keeps only the or branch
            .resultType(OrderVO.class))
        .getRows();

// Style 2: single-object passing (parameter name = objectName.propertyName, :xxxVo.creatorId uses the property path)
StaffInfoVO xxxVo = new StaffInfoVO();
xxxVo.setCreatorId("U0001");   // corresponds to #[and t.creator_id=:xxxVo.creatorId]
List rows2 = lightDao.findByQuery(
        new QueryExecutor("show_case")
            .names("xxxVo")     // pass a single object
            .values(xxxVo)
            .resultType(OrderVO.class))
        .getRows();
```

---

5. Complete `<sql>` Specification Quick Reference

Below is a `<sql>` element listing **all available child tags** (validated against `sqltoy.xsd`). This is a "full landscape" reference — **in practice, just pick what you need; the vast majority of SQL only needs `<value>` plus a handful of tags** (see the minimal example at the end of this section). Note that some tags are mutually exclusive (e.g. `pivot` and `unpivot`).

```xml
<?xml version="1.0" encoding="utf-8"?>
<sqltoy xmlns="http://www.sagframe.com/schema/sqltoy"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.sagframe.com/schema/sqltoy https://sagframe.github.io/schema/sqltoy.xsd">
<!--
  id: recommended to name as moduleName+functionName to avoid duplicates across modules
  type: search/insert/update/delete (select is a legacy-compatible alias, same as search)
  debug: whether to output execution logs for this sql (defaults to the global debug value)
  blank-to-null: whether blank strings are converted to null, default true; once <blank params="specificParam"/> appears in filters, the default is
switched off automatically
  query-timeout: statement timeout for a single sql (seconds)
  dataSource: specify a data source name (special scenarios)
-->
<sql id="sqltoy_sql_specs" type="search" debug="false" blank-to-null="true">
    <!-- sql description for readability and maintenance; not executed; must be used together with <value> -->
    <remarks>Staff information query demo</remarks>

    <!-- ===== filters: pre-normalizes the parameter values participating in query/execution (see section 4 of this page) ===== -->
    <filters>
        <!-- common: eq, to-date; targeted: cache-arg; fallback: primary -->
        <!-- Converts the parameter to null when it equals the given value (e.g. a dropdown "All" sends -1 so the condition is not applied) -->
        <eq params="organType" value="-1" />
        <!-- Date conversion: format can be a custom pattern like yyyy-MM-dd, or
FIRST_OF_MONTH/LAST_OF_MONTH/FIRST_OF_YEAR/LAST_OF_YEAR/FIRST_OF_WEEK/LAST_OF_WEEK;
             increment-time adds/subtracts time, increment-unit defaults to days -->
        <to-date params="beginDate" format="yyyy-MM-dd" increment-time="1" increment-unit="days"/>
        <!-- Copies a parameter value into a new parameter (often combined with to-date to build a date range from a single date) -->
        <clone param="beginDate" as-param="endDate"/>
        <to-number params="amt" data-type="decimal" />
        <!-- Convert to string; add-quote: none/single/double -->
        <to-string params="staffName" add-quote="single" />
        <!-- Escape special characters in like parameters -->
        <escapeLike params="staffName"/>
        <!-- Prepend/append % (skipped if the parameter already contains %) -->
        <l-like params="staffName"/>
        <r-like params="staffName"/>
        <!-- Reverse cache matching: matches names via like into the corresponding codes used as conditions, enabling precise queries -->
        <cache-arg param="staffName" cache-name="staffCache" alias-name="staffIds" prior-match-equal="false">
            <!-- Filters the cache, e.g. by personally authorized organizations or active status; cache-index refers to which cache column -->
            <filter compare-param="1" cache-index="4"/>
        </cache-arg>
        <!-- Primary parameter: e.g. when an order number is entered precisely, all conditions except the permission conditions excluded via excludes
are set to null -->
        <primary param="orderId" excludes="organIds" />
        <!-- Concatenates an array into an in-condition string wrapped in single quotes -->
        <to-in-arg params="organIds" />
        <!-- Formats a date into a string (combine with to-number to get numeric months like 202112) -->
        <date-format params="bizMonth" format="yyyyMM"/>
        <!-- Blank to null (all blanks are converted to null by default; usually no configuration needed) -->
        <blank params="*" excludes="staffName" />
        <!-- Converts to null when the range/magnitude comparison is satisfied -->
        <between params="amt" start-value="0" end-value="9999" excludes="" />
        <lte params="" value="" />
        <lt params="" value="" />
        <gte params="" value="" />
        <gt params="" value="" />
        <!-- Removes null or blank entries from arrays/collections -->
        <remove-null params="organIds" remove-blank="true"/>
        <!-- Character replacement (regex replaces all occurrences by default; is-first=true replaces only the first) -->
        <replace params="" regex="" value="" is-first="false" />
        <!-- Default values: sysdate()-1d (d=days/h=hours/w=weeks/m=months/y=years); first_of_month-3d/first_of_year/last_of_month... -->
        <default params="beginDate" data-type="localDate" value="sysdate()-1d" />
        <!-- Exclusive parameter: when a parameter holds a specific value, other parameters are set to specific values -->
        <exclusive param="" compare-type="eq" compare-values="" set-params="" set-value="" />
        <!-- Custom parameter handler (implement FilterHandler, configured via spring.sqltoy.customFilterHandler) -->
        <custom-handler params="" type=""/>
    </filters>

    <!-- ===== Cache Translate: code→name, supports A,B concatenated translation (split-sign/link-sign) ===== -->
    <!-- uncached-template: supplementary display when no match is found; ${value} stands for the original key value, e.g. [${value} undefined] -->
    <translate cache="dictCache" cache-type="POST_TYPE" columns="POST_TYPE" cache-indexs="1" uncached-template="" />

    <!-- ===== Security: masking and decryption ===== -->
    <!-- Masking type: tel/name/id-card/bank-card/address/email/public-account/discrete-rate/other -->
    <secure-mask columns="telNo" type="tel" head-size="3" tail-size="4" mask-code="*" mask-rate="50" />
    <!-- Decrypts encrypted stored fields at query time -->
    <secure-decrypt columns="idCard"/>
```

```
    <!-- ===== Database/Table sharding ===== -->
    <sharding-datasource strategy="multiDataBase" />
    <sharding-table tables="sys_order" strategy="hisRealTable" params="beginDate" />

    <!-- ===== Pagination Optimize: caches count results for identical conditions ===== -->
    <!-- alive-max: how many sets of counts for different conditions are kept per sql; alive-seconds: survival seconds; parallel: whether count and
the data query run in parallel -->
    <page-optimize alive-max="100" alive-seconds="600" parallel="false" />

    <!-- ===== Result-set formatting ===== -->
    <date-format columns="createTime" format="yyyy-MM-dd HH:mm:ss" />
    <!-- Number formats: #,###.00, capital (Chinese uppercase), capital-rmb (uppercase RMB amount), capital-en; supports rounding-
mode/locale/currency -->
    <number-format columns="totalAmt" format="capital-rmb" />

    <!-- ===== Tree sorting, sorting within levels, level-by-level aggregation ===== -->
    <tree-sort id-column="organ_id" pid-column="organ_pid" sum-columns="staff_cnt" level-order-column="staff_cnt" order-way="desc">
        <!-- Rows with status 0 are excluded from aggregation -->
        <sum-filter column="status" compare-type="neq" compare-values="0"/>
    </tree-sort>

    <!-- ===== SQL body (required) ===== -->
    <value>
    <![CDATA[
    -- comments can be written directly in the sql
    select t1.*,t2.ORGAN_NAME from
    @fast(select * from sys_staff_info t
          where #[t.sexType=:sexType]
            #[and t.JOIN_DATE>:beginDate]
            #[and t.STAFF_NAME like :staffName]
            -- @if() for logical decisions
            #[@if(:isVirtual==true||:isVirtual==0) and t.IS_VIRTUAL=1]
          ) t1,sys_organ_info t2
     where t1.ORGAN_ID=t2.ORGAN_ID
    ]]>
    </value>

    <!-- Custom count statement (only effective for pagination; sqltoy already optimizes count intelligently — hand-write it only in extreme
performance scenarios) -->
    <count-sql><![CDATA[]]></count-sql>

    <!-- ===== Result-set analysis algorithms ===== -->
    <!-- Grouped totals/averages -->
    <summary sum-columns="staff_cnt" average-columns="amt" average-radix-sizes="2" reverse="false" sum-site="left" average-skip-null="false">
        <global sum-label="Total" label-column="organ_name" />
        <!-- order-column: sort column within groups; order-with-sum defaults to true; order-way desc/asc -->
        <group sum-label="Subtotal" label-column="organ_name" group-column="organ_id" order-column="staff_cnt"/>
    </summary>
    <!-- Grouped concatenation (replaces group_concat/WM_CONCAT): id-columns are the grouping columns, columns are the concatenated columns
(required), result-type can be ARRAY/LIST/SET -->
    <link id-columns="organ_id" columns="staff_name" sign="," distinct="true"/>
    <!-- Rows to columns (mutually exclusive with unpivot) -->
    <pivot category-columns="order_month" group-columns="fruit_name" start-column="sale_count" end-column="total_amt" default-value="0" />
    <!-- Columns to rows: columns-to-rows format "column:label,column:label" -->
    <unpivot columns-to-rows="chinese_score:Chinese,math_score:Math" new-columns-labels="subject_name,score_value" />
    <!-- Column chain-relative (horizontal comparison among same-type columns): group-size is the number of metrics per group, relative-indexs are
the indexes of the metrics being compared, start-column is the starting column -->
    <cols-chain-relative group-size="3" relative-indexs="1,2" start-column="1" format="#.00%" />
    <!-- Row chain-relative (vertical comparison across rows): group-column is the grouping column, relative-columns are the columns being compared -
->
    <rows-chain-relative group-column="sale_date" relative-columns="sales_amt" format="#.00%" reduce-one="true" multiply="100" insert="true" />
</sql>
</sqltoy>
```

Child Tag Quick Reference

Child tag	Purpose	See
`<remarks>`	sql description (not executed)	This page
`<filters>`	Condition parameter pre-normalization (to-null / type conversion / reverse cache matching, etc.)	Section 4 of this page
`<translate>`	Cache Translate (code→name)	Cache Translate in Action
`<secure-mask>` / `<secure-decrypt>`	Result-set masking / field decryption	Data Masking & Encryption
`<sharding-datasource>` / `<sharding-table>`	Database / table sharding strategies	Sharding
`<page-optimize>` / `<count-sql>`	Pagination count cache optimization / custom count	Pagination Optimize
`<date-format>` / `<number-format>`	Result-set date / number formatting	This page
`<summary>`	Grouped totals / averages	Complex Query & Analytics
`<pivot>` / `<unpivot>`	Rows to columns / columns to rows	Complex Query & Analytics
`<cols-chain-relative>` / `<rows-chain-relative>`	Column / row period-over-period comparison	Complex Query & Analytics
`<tree-sort>`	Tree sorting and level-by-level aggregation	Complex Query & Analytics
`<link>`	Grouped concatenation (replaces group_concat)	Complex Query & Analytics
`<reverse>`	Reverses the result-set order	Complex Query & Analytics
`<value>`	SQL body (required); supports `#[]`, `@if`, `@fast`, etc.	This page / Tags & Expressions

Minimal Everyday Example

Don't be intimidated by the "full landscape" above — everyday SQL is usually just this simple. For example, suppose the page has a gender dropdown whose "All" option carries `value="-1"`; an `eq` filter turns `-1` into null, and that condition automatically drops out of the query:

```
<select name="sexType">
    <option value="-1">All</option>
    <option value="F">Female</option>
    <option value="M">Male</option>
</select>
```

```
<sql id="sys_staff_find">
    <filters>
        <!-- When sexType=-1 (All) it converts to null and the whole #[and t.sex_type=:sexType] fragment is removed -->
        <eq params="sexType" value="-1" />
    </filters>
    <translate cache="dictCache" cache-type="POST_TYPE" columns="postType" />
    <value><![CDATA[
        select * from sys_staff_info t
        where 1=1
        #[and t.sex_type=:sexType]
        #[and t.staff_name like :staffName]
    ]]></value>
</sql>
```

6. Related Advanced Syntax

- Macro directives such as `@if()` / `@elseif()` / `@else`, `@fast`, `@blank()`, `@value()`, `@loop()` / `@secure-loop()`, `@include(sqlId)`, `@include(:scriptParam)`, `@joinFields()` (safe field-list joining for page-configurable reports), `@split()` (splits a string parameter into an array for `in` expansion) are covered in Tags & Expressions.
- More practical patterns (dynamic `@include`, configuration-driven SQL integration, slow SQL statistics, etc.) are covered in SQL Showcase.
- Query APIs (findOne/find/findPage/findTop/findRandom/getValue/isUnique, etc.) are covered in Query API.

Query API

lightDao API

Prerequisite: Template SQL

```
<sql id="sqltoy_staff_findlist">
       <!-- Pagination optimizer: when the query conditions stay identical, caches the total record count for a period of time so the total count
does not need to be queried on every request -->
       <!-- parallel: whether to query the total record count and the single page of data in parallel; cache optimization is disabled when alive-
max=1 -->
       <!-- alive-max: how many counts for different query conditions can be kept at most; alive-seconds: how long a query-condition count stays
alive (e.g. 120 seconds; beyond the threshold it is queried again) -->
       <page-optimize parallel="true" alive-max="100" alive-seconds="120" />
       <value><![CDATA[
              select t1.*,t2.ORGAN_NAME
              -- @fast() paginates first and takes 10 rows (the exact number is determined by pageSize), then joins
              from  @fast(
                    select t.*  from sqltoy_staff_info t
                        where t.STATUS=1
                          #[and t.STAFF_NAME like :staffName]
                        order by t.ENTRY_DATE desc
                        ) t1
              left join sqltoy_organ_info t2 on  t1.organ_id=t2.ORGAN_ID
         ]]>
       </value>
       <!-- For extremely special cases, a custom count-sql is provided here to achieve targeted performance optimization -->
       <!-- <count-sql></count-sql> -->
</sql>
```

findOne: Find a Single Record

```
StaffInfo staffInfo = lightDao.findOne("sqltoy_staff_findlist", new StaffInfo(), StaffInfo.class);
```

find: Find a List

```
List<StaffInfo> list = lightDao.find("sqltoy_staff_findlist", new StaffInfo(), StaffInfo.class);
```

findEntity: Entity Query

```
List<SysLog> list = lightDao.findEntity(SysLog.class, EntityQuery.create().names("operator").values("112233"));
```

findPage: Paginated Query

```
Page<StaffInfo> page = lightDao.findPage(new Page<>(10, 1), "sqltoy_staff_findlist", new StaffInfo(), StaffInfo.class);
```

findPageEntity: Paginated Entity Query

```
Page<SysLog> page = lightDao.findPageEntity(new Page<>(10, 1), SysLog.class, EntityQuery.create().names("operator").values("112233"));
```

findTop: Find Top-N Records

```
List<StaffInfo> list = lightDao.findTop("sqltoy_staff_findlist", new StaffInfo(), StaffInfo.class, 12);
```

findRandom: Find N Random Records

```
List<StaffInfo> list = lightDao.findRandom("sqltoy_staff_findlist", new StaffInfo(), StaffInfo.class, 12);
```

getValue, getCount, isUnique: Shortcut Queries

- api

```
// getValue: returns the value of the first row and first column of the query result (typically used for select max(x), aggregate values, etc.)
public Object getValue(final String sqlOrSqlId, final Map<String, Object> paramsMap);
public <T> T getValue(final String sqlOrSqlId, final Map<String, Object> paramsMap, final Class<T> resultType);

// getCount: returns the total record count (available for both SQL and single-table entity query forms)
public Long getCount(final String sqlOrSqlId, final Map<String, Object> paramsMap);
public Long getCount(Class entityClass, EntityQuery entityQuery);

// isUnique: checks whether the entity's property values are unique in the table (commonly used for uniqueness validation of codes)
public boolean isUnique(Serializable entity, String... paramsNamed);
```

- usage examples

```
// get an aggregate value (with an explicit return type)
BigDecimal maxAmt = lightDao.getValue("sqltoy_order_max_amt", paramsMap, BigDecimal.class);

// total record count
Long total = lightDao.getCount("sqltoy_order_find", paramsMap);
Long cnt = lightDao.getCount(StaffInfo.class, EntityQuery.create().where("status=?").values(1));

// uniqueness check (returns true when unique; paramsNamed takes entity property names, not database column names)
boolean unique = lightDao.isUnique(staffInfo, "staffCode");
// fluent form: lightDao.unique().entity(entity).fields("staffCode").submit(), see Link Chain API
```

> A single record can also be fetched with `loadByQuery(new QueryExecutor(sql).names(...).values(...)...)` (returns Object); for common scenarios `findOne` is recommended. Top / random queries also have QueryExecutor forms: `findTopByQuery(queryExecutor, topSize)` and `findRandomByQuery(queryExecutor, randomSize)`, see the findTop / findRandom sections above.

fetchStream: Stream Data Fetching

- For extremely large data volumes that cannot be held directly in a List or other collection, a callback approach is provided so developers can write the data row by row into a specific file or other storage themselves

```
/**
 * fetch query results as a stream
 * @param queryExecutor       use queryExecutor.showsql() to toggle SQL log output
 * @param streamResultHandler
 */
//public void fetchStream(final QueryExecutor queryExecutor, final StreamResultHandler streamResultHandler);

List result = new ArrayList();
// the sql can be defined in xml; here it is shown inline for demonstration
String sql = "select * from sqltoy_staff_info";
lightDao.fetchStream(new QueryExecutor(sql).resultType(StaffInfoVO.class),
    new StreamResultHandler() {
        @Override
        public void consume(Object row, int rowIndex) {
            result.add(row);
        }

        // end is typically used for flushing files, etc.
        @Override
        public void end() {
            System.err.println("Execution completed");
        }
    }
);

for (Object item : result) {
    System.err.println(JSON.toJSONString(item));
}
```

> 🎬 Full runnable example: `FetchStreamTest.java` in the demo project `sqltoy-showcase`.

findByQuery: Hierarchy Packaging (parent and child objects returned in one query)

Scenario: the parent-child tables are fetched in a single join (e.g. data dictionary types + dictionary details), and based on the `@OneToMany` / `@OneToOne` annotations on the result VO, the framework automatically **groups and packages the flat result set** into parent-child objects (1..n levels) — no second query, no manual assembly.

Step 1: declare the parent-child relationship on the result VO (`notNullField` is the key)

```
/**
 * child table info associated by primary key
 * notNullField: when packaging the hierarchy, whether this child-table field is null decides whether the joined record has child-table data (must be
set for left-join scenarios)
 */
@OneToMany(fields = { "dictType" }, mappedFields = { "dictType" }, delete = true,
        orderBy = "showIndex desc", notNullField = "dictKey")
private List<DictDetailVO> dictDetailVOs = new ArrayList<DictDetailVO>();
```

`@OneToMany` / `@OneToOne` attributes:

Attribute	Description
`fields`	Parent-table association columns (multiple allowed, corresponding to composite keys)
`mappedFields`	Corresponding association properties on the child object
`notNullField`	Non-null field used to decide whether child-table data exists during hierarchy packaging (for hiberarchy)
`orderBy`	Child-table ordering for **cascade load**, e.g. `showIndex desc`; in hierarchy-packaging (hiberarchy) scenarios the child order follows the SQL's `ORDER BY`
`load`	Custom load sql (e.g. `enable=1` or a full sql: `select * from t where fkField=:fkField`)
`delete`	Whether to cascade delete
`update`	Custom operation on the child table during cascade save/update (`delete` means delete first, `status=0` means disable)

Step 2: findByQuery + hiberarchy packages the hierarchy

```
@Test
public void testOneToMany() {
    // one join sql fetches the dictionary parent table and the dictionary details together
    String sql = "select sdt.DICT_TYPE_NAME, sdt.COMMENTS dictTypeComments, "
            + "sdt.status dictTypeStatus, sdd.*, sdd.STATUS dictStatus from sqltoy_dict_type sdt "
            + " left join sqltoy_dict_detail sdd on sdt.DICT_TYPE = sdd.DICT_TYPE ";
    List<DictTypeVO> result = lightDao
            .findByQuery(new QueryExecutor(sql).resultType(DictTypeVO.class)
                    // enable hierarchy packaging (not needed if hiberarchyFieldsMap is already set)
                    .hiberarchy(true)
                    // when parent and child objects have same-named properties (e.g. both have status), map result-set labels to object properties:
                    // key is the query result's label (alias), value is the actual column/property name
                    .hiberarchyFieldsMap(DictTypeVO.class,
                            MapKit.keys("dictTypeComments", "dictTypeStatus").values("COMMENTS", "status"))
                    .hiberarchyFieldsMap(DictDetailVO.class,
                            MapKit.keys("dictStatus").values("status")))
            .getRows();
    // the JSON output is a 1..n hierarchy: each dictionary type carries a dictDetailVOs list
    for (DictTypeVO dictType : result) {
        System.err.println(JSON.toJSONString(dictType));
    }
}
```

The packaged result looks like:

```
{ "comments": "Device Type", "dictDetailVOs": [ { "dictKey": "NET", "dictName": "Network Device", "dictType": "DEVICE_TYPE", "showIndex": 2,
"status": 1 } ] }
```

> [!NOTE]
>
> - The method names are `hiberarchy` / `hiberarchyFieldsMap` (a historical spelling in the framework, not hierarchy).
> - `hiberarchyFieldsMap` **automatically enables** hierarchy packaging and registers the involved classes, so once it is set there is no need to call `hiberarchy(true)` again.
> - `hiberarchyFieldsMap` only targets the scenario where **parent and child objects have same-named properties**; without name conflicts, `hiberarchy(true)` alone is enough.
> - Hierarchy packaging means "packaging directly from a single join sql", which is complementary to the cascading load in Object CRUD (batch secondary queries by primary key).

> 🎬 Full runnable example: `HierarchyPackagingTest.java` in the demo project `sqltoy-showcase` (its `DictTypeVO` has `notNullField = "dictKey"` configured on @OneToMany; the child order follows the join SQL's `ORDER BY`).

Stored Procedure Calls

For the complete description and examples of stored procedure calls (`executeStore` / `executeMoreResultStore` , the fluent `store()` , out parameters, **multiple result sets**), see Stored Procedures.

```
// quick example: call a stored procedure and get the result set (pass null for outParamsType when there are no out parameters)
List result = lightDao.executeStore("{call storeName(?,?)}",
        new Object[]{ value1, value2 }, null, VO.class).getRows();
```

Link Chain API

Besides explicit interfaces such as `save` / `update` / `find` / `findPage`, `LightDao` also provides a set of **fluent (Link) entry points** that organize "set parameters → submit and execute" into fluent chained calls. When you need to dynamically specify the **datasource, lock, timeout, batch size, parallel configuration, force-update fields**, etc., the fluent style is clearer and more flexible.

1. Fluent Entry Overview

Entry	Returns	Purpose	Submit Methods
`lightDao.query()`	`Query`	SQL/entity query	`find()` / `getOne()` / `getValue()` / `count()` / `findPage()` / `findTop()` / `findRandom()`
`lightDao.save()`	`Save`	Save objects	`one(entity)` / `many(entities)`
`lightDao.update()`	`Update`	Update objects	`one(entity)` / `many(entities)`
`lightDao.delete()`	`Delete`	Delete objects	`one(entity)` / `many(entities)`
`lightDao.load()`	`Load`	Load objects (with cascade/lock)	`one(entity)` / `many(entities)`
`lightDao.unique()`	`Unique`	Uniqueness check	`submit()`
`lightDao.execute()`	`Execute`	Execute insert/update/delete SQL	`submit()` / `insertReturnPrimaryKey()`
`lightDao.batch()`	`Batch`	Batch execution	`submit()`
`lightDao.store()`	`Store`	Stored procedure call	`submit()`
`lightDao.treeTable()`	`TreeTable`	Tree-table node route building	`submit()`
`lightDao.elastic()`	`Elastic`	Elasticsearch query	`find()` / `getOne()` / `findPage()` / `findTop()`
`lightDao.mongo()`	`Mongo`	MongoDB query	`find()` / `getOne()` / `findPage()` / `findTop()`
`lightDao.tableApi()`	`TableApi`	Table metadata/truncate/drop	Direct call

> Almost all fluent objects support `.dataSource(DataSource)` to **temporarily switch the datasource**, as well as `.autoCommit(Boolean)`, `.batchSize(int)`, `.parallelConfig(ParallelConfig)`, etc.

2. Query: query()

```
// query a list: sql can be a sqlId defined in xml, or a direct sql
List<StaffInfoVO> list = lightDao.query()
        .sql("sys_staff_find")
        .names("staffName", "status")
        .values("Zhang San", 1)
        .resultType(StaffInfoVO.class)
        .find();

// pass parameters via an entity (the framework maps parameter names in the sql to entity properties to take values)
StaffInfoVO one = (StaffInfoVO) lightDao.query()
        .sql("sys_staff_find")
        .entity(staffInfoVO)
        .resultType(StaffInfoVO.class)
        .getOne();

// get a single value
Long count = lightDao.query().sql("select count(1) from sqltoy_staff_info").getValue(Long.class);

// pagination (Page is constructed as new Page(pageSize, pageNo))
Page<StaffInfoVO> page = (Page) lightDao.query()
        .sql("sys_staff_find").resultType(StaffInfoVO.class)
        .findPage(new Page(10, 1));

// top-N / random
List top = lightDao.query().sql("sys_staff_find").resultType(StaffInfoVO.class).findTop(10);

// lock, timeout, fetch size for stream fetching
lightDao.query().sql(...).lock(LockMode.UPGRADE).queryTimeout(30).fetchSize(1000).find();
```

`Query` supports fluent settings: `sql`, `names`, `values`, `entity`, `resultType`, `dataSource`, `fetchSize`, `maxRows`, `lock`, `lockWaitTimeout`, `queryTimeout`, `humpMapLabel` (whether Map result labels use camelCase).

---

3. Save: save()

```
// save a single object and return the primary key
Object pk = lightDao.save().one(entity);

// batch save and return the number of successfully saved rows
Long rows = lightDao.save().many(entities);

// deep cascade save + save mode + batch size + parallel
lightDao.save()
        .deeply(true)                // cascade save associated objects
        .saveMode(SaveMode.IGNORE)   // APPEND(normal)/UPDATE(saveOrUpdate)/IGNORE(ignore if exists)
        .batchSize(500)
        .parallelConfig(parallelConfig)
        .many(entities);
```

---

4. Update: update()

```
// flexible update: by default null fields are excluded from the update, done in a single database interaction
Long rows = lightDao.update().one(entity);

// force update specific fields (updated even when the value is null)
lightDao.update().forceUpdateFields("photo", "remark").one(entity);

// deep cascade update / specify cascade classes
lightDao.update().deeply(true).one(entity);
lightDao.update().cascadeClasses(OrganInfo.class).one(entity);

// batch update
lightDao.update().batchSize(500).many(entities);
```

`Update` supports fluent settings: `deeply`, `updateFields`, `forceUpdateProps`, `forceUpdateFields`, `cascadeClasses`, `cascadeForceUpdate`, `batchSize`, `autoCommit`, `dataSource`, `parallelConfig`.

---

5. Delete: delete() and Load: load()

```
// delete
lightDao.delete().one(entity);
lightDao.delete().many(entities);

// load a single object and cascade load associated objects
StaffInfo staff = lightDao.load().cascade(OrganInfo.class).one(entity);
// load all cascades
lightDao.load().cascadeAll().one(entity);
// locked load (pessimistic lock)
lightDao.load().lock(LockMode.UPGRADE).lockWaitTimeout(10).one(entity);
// batch load
List list = lightDao.load().many(entities);
```

`LockMode`: `UPGRADE` (normal pessimistic lock), `UPGRADE_NOWAIT` (no waiting), `UPGRADE_SKIPLOCK` (skip already-locked rows).

---

6. Uniqueness: unique()

```
// check whether the entity's specified properties are unique in the database (property names, not database column names)
boolean unique = lightDao.unique()
        .entity(entity)
        .fields("staffCode", "tenantId")
        .submit();
```

---

7. Execute SQL: execute() and Batch: batch()

```
// execute update/insert/delete/merge statements and return the affected row count
Long rows = lightDao.execute()
        .sql("update sqltoy_staff_info set status=:status where staff_id=:staffId")
        .names("status", "staffId").values(0, "S0001")
        .submit();

// insert and return the primary key
Object pk = lightDao.execute().sql("sys_staff_insert").entity(entity)
        .insertReturnPrimaryKey("staffId");

// batch execution (dataSet is List<Map> or List<VO>)
Long batchRows = lightDao.batch()
        .sql("sys_staff_insert")
        .dataSet(dataList)
        .batchSize(500)
        .submit();
```

---

8. Stored Procedure: store()

```
// call a stored procedure and get the result
StoreResult result = lightDao.store()
        .sql("{call storeName(?,?)}")
        .inParams(value1, value2)
        .resultType(StaffInfoVO.class)
        .submit();
List rows = result.getRows();

// with out parameters / multiple result sets
StoreResult more = lightDao.store()
        .sql("{call storeName(?,?,?)}")
        .inParams(value1, value2)
        .outTypes(java.sql.Types.INTEGER)
        .moreResult(true)
        .resultTypes(StaffInfoVO.class)
        .submit();
Object[] outValues = more.getOutResult();
List[] allResults = more.getMoreResults();
```

---

9. Tree Table: treeTable()

```
// build tree-table node route (node_route), node level, leaf flag, etc.
boolean ok = lightDao.treeTable()
        .treeModel(new TreeTableModel(OrganInfo.class)
                .idField("organId").pidField("organPid")
                .nodeRouteField("nodeRoute").nodeLevelField("nodeLevel")
                .isLeafField("isLeaf"))
        .submit();
```

---

10. NoSQL: elastic() and mongo()

```
// Elasticsearch (sql mode or native json mode)
List list = lightDao.elastic().sql("es_sql_id").names(...).values(...).resultType(...).find();
Page page = lightDao.elastic().sql("es_sql_id").findPage(new Page(10, 1));

// MongoDB
List mList = lightDao.mongo().sql("mql_id").entity(queryVO).resultType(...).find();
```

See Elasticsearch Support and MongoDB Support for details.

---

11. Table Metadata: tableApi()

```
// get table/column metadata
List<TableMeta> tables = lightDao.tableApi().getTables(null, null, "sqltoy%");
List<ColumnMeta> columns = lightDao.tableApi().getTableColumns(null, null, "sqltoy_staff_info");
// truncate a table / drop a table (supports table names or entity classes)
lightDao.tableApi().truncate("sqltoy_staff_info");
lightDao.tableApi().drop(StaffInfo.class);
```

Lambda Conditions

> Available since 6.0.8. Build query, update and delete conditions from **entity property method references** (e.g. `StaffInfo::getStatus`), so a misspelled property is caught at compile time. Single-table and multi-table joins share one builder — whether a query is a join is decided by whether you call a join step.

0. Positioning and division of labour

sqltoy offers three ways to organize a query; pick by scenario:

Approach	Good for	Entry point
Lambda conditions (this page)	Everyday conditional queries over one or more joined tables; conditional update/delete with conditions expressed as property references	`lightDao.lambdaQuery/lambdaUpdate/lambdaDelete`
EntityQuery/EntityUpdate	Single-table string conditions (where text + named parameters)	`lightDao.findEntity/updateByQuery` etc.
SQL mode (sqlId + `#[...]` segments + `@if` macros)	Complex dynamic SQL, multi-branch condition combinations, dialect variants, configurable reports	`lightDao.find/findPage` + sqlId, see Dynamic SQL Guide

Dynamism in the lambda layer is expressed with the `IfPresent` sugar (see Dynamic conditions); **do not try to pile complex dynamic logic into the lambda layer — switch to SQL mode instead**, where segments, macros, dialect variants and hot reload already exist.

On the **single-table channel** a lambda query converts conditions into `EntityQuery` / `EntityUpdate` and inherits every entity-level capability (sharding, cache translate, encryption/masking, tenant filtering, unified update fields). On the **join channel** it assembles full SQL and goes through the native SQL path, so entity-level enhancements no longer apply (a multi-table SQL cannot use single-entity enhancements by nature).

---

1. Entry points and builders

```
LambdaQuery<T>  lightDao.lambdaQuery(Class<T> entityClass);   // query (single table or join)
LambdaUpdate<T> lightDao.lambdaUpdate(Class<T> entityClass);  // update (single table or join)
LambdaDelete<T> lightDao.lambdaDelete(Class<T> entityClass);  // delete (single table or join)
```

A builder is a single mutable object; every method returns itself, so **chained and imperative styles can be mixed freely**:

```
LambdaQuery<StaffInfo> query = lightDao.lambdaQuery(StaffInfo.class);
if (status != null) {
    query.eq(StaffInfo::getStatus, status);
}
List<StaffInfo> list = query.orderByDesc(StaffInfo::getEntryTime).find();
```

> **Note**: the `if` above only demonstrates that "the builder is a single mutable object, so imperative control flow (if/for) mixes freely with chaining" — it is **not** a recommendation to null-check by hand. The normal way to say "skip this condition when the value is null" is `eqIfPresent`, one line, no hand-written `if` (see Dynamic conditions later):

```
List<StaffInfo> list = lightDao.lambdaQuery(StaffInfo.class)
        .eqIfPresent(StaffInfo::getStatus, status)   // skipped automatically when status is null
        .orderByDesc(StaffInfo::getEntryTime)
        .find();
```

You can also build first and execute later: a builder created with `LambdaQuery.create(...)` is not bound to a dao, so terminals are unavailable until you call `bind(lightDao)` (handy when a service layer assembles conditions and execution happens after crossing layers):

```
LambdaQuery<StaffInfo> query = LambdaQuery.create(StaffInfo.class).eq(StaffInfo::getStatus, 1);
List<StaffInfo> list = query.bind(lightDao).find();
```

> `LightDao` exposes only three lambda factories (`lambdaQuery/lambdaUpdate/lambdaDelete`); **all execution is driven by builder terminals**, the dao offers no execution overloads.

Query terminals at a glance:

Terminal	Semantics
`find()`	All matching rows
`getOne()`	A single row: **0 rows returns null, more than 1 throws** `DataAccessException` (multiple matches mean the condition is defective; the first row is never taken silently)
`findPage(Page)`	Pagination, `new Page(pageSize, pageNo)`
`findTop(n)` / `findTop(0.2)`	First n rows / first 20% (greater than 1 means a row count, less than 1 a ratio), ordered by `orderBy`
`findRandom(n)` / `findRandom(0.2)`	n random rows / a random ratio
`count()` / `exists()`	Row count / whether anything matches

Choosing a datasource (multiple datasources / read-write splitting)

Builders offer `.dataSource(DataSource)` to pin the operation to a datasource; without it the dao's default datasource is used. Consistent with existing sqltoy practice (`EntityQuery` / `QueryExecutor` / `lightDao.update().dataSource(...)`), it **works for every terminal on both the single-table and join channels — query, count, update and delete**:

```
// single-table query against a read-only replica
List<StaffInfo> list = lightDao.lambdaQuery(StaffInfo.class)
        .dataSource(readOnlyDs).eq(StaffInfo::getStatus, 1).find();

// join update against a specific database
lightDao.lambdaUpdate(StaffInfo.class).dataSource(ds2)
        .leftJoin(Depart.class).on(StaffInfo::getDeptId, Depart::getId)
        .set(StaffInfo::getStatus, 2).eq(Depart::getDeptName, "R&D").update();
```

> The single-table channel passes the datasource through `EntityQuery/EntityUpdate` ; the join channel passes it through sqltoy's existing `query()` / `execute()` chain builders (query/count use `query().dataSource(ds)` , update/delete use `execute().dataSource(ds)` — both already supported datasource selection); for a join delete with several targets the transaction runs on the datasource you specified. A null `dataSource` behaves exactly as if it were not set.

Pagination optimization (parallel count + keep-alive cache)

`findPage` can reuse sqltoy's existing pagination optimization through `.pageOptimize(PageOptimize)` (when unset, the global `SqlToyConfig.pageOptimize` applies). Two common knobs: `parallel(true)` runs the count and the data query in parallel; `aliveSeconds(n)` caches the count for n seconds (keep-alive), so paging skips the count query while the cache is warm:

```
Page<StaffInfo> page = lightDao.lambdaQuery(StaffInfo.class)
        .eq(StaffInfo::getStatus, 1)
        .pageOptimize(new PageOptimize().parallel(true).aliveSeconds(120))
        .findPage(new Page(20, 1));
```

`PageOptimize` also offers `aliveMax` (cache entry limit), `skipZeroCount` (when a cached count of 0 is hit, skip that cache entry and re-fetch the total count) and `parallelMaxWaitSeconds` (parallel wait limit). Both channels are covered (single table through `EntityQuery` , join through `QueryExecutor`).

---

2. Single-table queries

Condition methods

Every condition method has a `(boolean condition, ...)` overload; when `condition` is false the condition does not participate in the SQL:

Method	SQL shape
`eq / ne / gt / ge / lt / le`	`column = :p1`, `<>`, `>`, `>=`, `<`, `<=`
`between / notBetween`	`column BETWEEN :p1 AND :p2`
`like / notLike`	`column LIKE :p1` / `NOT LIKE :p1`; the `%value%` wildcards are wrapped into the parameter value (contains match)
`likeLeft / likeRight`	`LIKE :p1` with the value as `%value` (ends with) / `value%` (starts with)
`likeExact / likeExactLeft / likeExactRight`	Same three directions, but `%` inside the value is first escaped as a literal before the directional `%` is padded — user input containing `%` matches as an ordinary character instead of acting as a wildcard (searching `"50%"` won't hit arbitrary `5xx…` strings)
`isNull / isNotNull`	`column IS NULL` / `IS NOT NULL`
`in(Collection)` / `in(values...)`	`column IN (:p1)`; empty collections/arrays are skipped automatically
`notIn(Collection)` / `notIn(values...)`	`column NOT IN (:p1)`
`inSql(column, subQuery[, paramMap])`	`column IN (subQuery)`; sqltoy named parameters of the sub-query come from the Map
`inb(colA, colB, tupleCollection)`	Multi-column tuple IN: a single column degenerates to `column IN (:p1,:p2…)`; several columns expand to OR-of-ANDs `((a=:p1 AND b=:p2) OR (a=:p3 AND b=:p4)…)`, avoiding row-value IN dialect differences on SQLServer/DB2 etc.

```
List<StaffInfo> list = lightDao.lambdaQuery(StaffInfo.class)
        .eq(StaffInfo::getStatus, 1)
        .like(StaffInfo::getStaffName, "Smith")
        // % inside the value matches as an ordinary character: searching "50%" won't treat % as a wildcard hitting any "5…" string
        .likeExactRight(StaffInfo::getStaffCode, "50%")
        .between(StaffInfo::getEntryTime, start, end)
        .in(StaffInfo::getOrganId, organIds)
        .orderByDesc(StaffInfo::getEntryTime)
        .find();
```

exists / notExists correlated sub-queries

A sub-query is built with the same lambda style; to reference an outer column use `OuterRef.of(outerField)` (type-safe, no hand-written table name or alias):

```
// single-table channel: there exists an organ row that is enabled and matches the staff's organ id
lightDao.lambdaQuery(StaffInfo.class)
        .exists(OrganInfo.class, sub -> sub
                .eq(OrganInfo::getStatus, 1)
                .eq(OrganInfo::getOrganId, OuterRef.of(StaffInfo::getOrganId)))  // references the outer column
        .find();
// ... WHERE ... AND EXISTS (SELECT 1 FROM organ_table sq1 WHERE sq1.STATUS = :p2 AND sq1.ORGAN_ID = staff_table.ORGAN_ID)

// the join channel works the same way; the outer column is rendered with the outer alias (e.g. s1.ORGAN_ID)
lightDao.lambdaQuery(StaffInfoVO.class).selectAll()
        .from(StaffInfo.class).leftJoin(OrganInfo.class).on(StaffInfo::getOrganId, OrganInfo::getOrganId)
        .notExists(BlackList.class, sub -> sub.eq(BlackList::getStaffId, OuterRef.of(StaffInfo::getStaffId)))
        .find();
```

Rules and boundaries:

- `OuterRef` only supports the six comparison methods `eq/ne/gt/ge/lt/le` (passing it to between/like/in throws); using it outside a sub-query also throws;
- On the single-table channel the outer column renders as `table.column`, on the join channel as `alias.column` — inside the sub-query it must be qualified, otherwise a same-named column becomes ambiguous;
- Sub-query table aliases use the `sq` prefix (`sq1`, `sq2` …), so they never collide with the outer single-letter aliases; nested exists supports several levels;
- Sub-query parameters share the named-parameter sequence with the outer query (`:p1, :p2…` are globally unique);
- The sub-query entity must match the entity that owns the sub-condition fields, otherwise assembly throws.

The `exists()` terminal (does any matching row exist) is implemented as `findTop(1)` and stops at the first hit — no full count aggregation.

Logical composition and nesting

```
// and/or connectors, nested parentheses, not negation (one grouping level)
lightDao.lambdaQuery(StaffInfo.class)
        .eq(StaffInfo::getStatus, 1)
        .or(n -> n.eq(StaffInfo::getOrganId, "O001").eq(StaffInfo::getStaffName, "Zhang San"))
        .not(n -> n.isNull(StaffInfo::getMobile))
        .find();
// → status = :p1 OR (organ_id = :p2 AND staff_name = :p3) AND NOT (mobile IS NULL)

// multi-level grouping: the argument n of nested/and/or/not is the same builder type and nests further, unlimited depth
lightDao.lambdaQuery(StaffInfo.class)
        .nested(outer -> outer
                .nested(inner -> inner.eq(StaffInfo::getOrganId, "O001").or().eq(StaffInfo::getOrganId, "O002"))
                .eq(StaffInfo::getStatus, 1))
        .or().eq(StaffInfo::getStaffId, "S009")
        .find();
// → ((organ_id = :p1 OR organ_id = :p2) AND status = :p3) OR staff_id = :p4

// accept: reuse a block of conditions merged into the current level (no parentheses); nested: an independent group (parenthesized)
lightDao.lambdaQuery(StaffInfo.class).eq(StaffInfo::getStatus, 1)
        .accept(q -> q.eq(StaffInfo::getOrganId, "O001").eq(StaffInfo::getStaffName, "Zhang San")).find();
// → status = :p1 AND organ_id = :p2 AND staff_name = :p3
lightDao.lambdaQuery(StaffInfo.class).eq(StaffInfo::getStatus, 1)
        .nested(q -> q.eq(StaffInfo::getOrganId, "O001").eq(StaffInfo::getStaffName, "Zhang San")).find();
// → status = :p1 AND (organ_id = :p2 AND staff_name = :p3)

// last appends a raw SQL fragment (injection risk — use with care)
lightDao.lambdaQuery(StaffInfo.class).eq(StaffInfo::getStatus, 1).last("limit 10").find();
```

Adjacent conditions get an automatic `AND`; adjacent explicit `and()/or()` connectors are merged keeping the last one; leading/trailing dangling connectors are stripped; every nesting level is wrapped in parentheses, **with no depth limit** (in practice constrained by JVM stack depth and the target database's expression nesting limit — 1~3 levels is sufficient for common needs, and more complex dynamic grouping should use SQL mode). **Empty groups are dropped automatically**: when every condition inside a group is skipped by `condition=false` or `IfPresent`, the whole group disappears together with its surplus connectors, so illegal `()` SQL is never rendered — dynamic groups are therefore safe to write, and a group simply stops participating when its parameters are missing.

Grouping, sorting and aggregate filtering

```
lightDao.lambdaQuery(StaffInfo.class)
        .groupBy(StaffInfo::getOrganId)
        .having("count(*) > :min", MapKit.map("min", 5))
        .orderByAsc(StaffInfo::getOrganId)
        .orderByDesc(StaffInfo::getEntryTime)   // several orderBy calls merge into one clause
        .find();
```

With grouping/sorting but no where condition, `1 = 1` is added as the where clause automatically.

Selecting specific columns

```
lightDao.lambdaQuery(StaffInfo.class)
        .select(StaffInfo::getStaffId, StaffInfo::getStaffName)   // property form
        .eq(StaffInfo::getStatus, 1)
        .find();
```

Dynamic conditions

The recommended idiom is the `IfPresent` sugar: the condition is skipped when the value is null (a blank string for strings, an empty collection for collections):

Method	Skipped when
`eqIfPresent / neIfPresent / gtIfPresent / geIfPresent / ltIfPresent / leIfPresent`	the value is null
`likeIfPresent`	a blank string
`inIfPresent(Collection)` / `inIfPresent(values...)` / `notInIfPresent(Collection)`	the collection/array is null or empty
`betweenIfPresent(v1, v2)` / `betweenIfPresent(array)`	both are null; a half-open range degrades automatically: only a lower bound becomes `ge`, only an upper bound becomes `le`

```
// the typical page-query style: include the condition only when the parameter is present
lightDao.lambdaQuery(StaffInfo.class)
        .eqIfPresent(StaffInfo::getStatus, status)
        .likeIfPresent(StaffInfo::getStaffName, name)
        .inIfPresent(StaffInfo::getOrganId, organIds)
        .betweenIfPresent(StaffInfo::getEntryTime, start, end)
        .find();
```

A skipped condition never resolves its column name during assembly, so it has no side effects. **For more complex dynamic combinations (multiple branches, dialect differences, configurability) use SQL mode.**

Value filtering

Null values are filtered by default; an instance-level strategy can be set before adding conditions:

```
lightDao.lambdaQuery(StaffInfo.class)
        .valueFilter(v -> v != null && !"".equals(v))   // instance-level, not a global static
        .eq(StaffInfo::getStaffName, name)
        .find();
```

---

3. Single-table updates

```
lightDao.lambdaUpdate(StaffInfo.class)
        .set(StaffInfo::getStatus, 2)
        .setExpr(StaffInfo::getLoginCount, "loginCount + 1")   // computed from the current value
        .eq(StaffInfo::getStaffId, staffId)
        .update();
```

setExpr expression updates

Form	Example	Notes
Literal expression	`.setExpr(U::getLoginCount, "loginCount + 1")`	Fields inside the expression are written as **property names** and converted to database columns at execution time; other columns of the same table may be referenced
`?` placeholder binding	`.setExpr(U::getSalary, "salary * ?", rate)`	A single `?` binds the second argument
`:name` named parameter	`.setExpr(U::getSalary, "salary * :rate", rate)`	Supported on the single-table channel (reusing the expression-set capability of updateByQuery)

The `=` character is **not allowed inside the expression** (the assignment equals sign is assembled by the framework); passing one throws.

Default value filtering skips `set(col, null)`. To set a column to null explicitly use `setNull`:

```
lightDao.lambdaUpdate(StaffInfo.class).setNull(StaffInfo::getMobile).eq(StaffInfo::getStaffId, id).update();
```

Safety boundaries

- **An update/delete with no where condition throws `DataAccessException` by default**; an explicit `.matchAll()` is required to allow it — this prevents accidental whole-table writes;
- When the where clause is a literal (no named parameters) `1=:DEFAULT_VALUE` is added internally as a fallback, sidestepping the parameter-count validation invisibly to the caller;
- **Whole-entity updates (by primary key) are not offered in the lambda layer**; use the existing path: `lightDao.update(entity)` / `lightDao.update(entity, forceUpdateProps)`.

---

4. Single-table deletes

```
lightDao.lambdaDelete(StaffInfo.class)
        .eq(StaffInfo::getStatus, 0)
        .delete();

// an unconditional delete must be explicitly allowed
lightDao.lambdaDelete(StaffInfo.class).matchAll().delete();
```

The delete builder only supports where conditions (no select/orderBy), keeping its responsibilities separate from the query builder.

---

5. Multi-table join queries

Calling a join step on the same `LambdaQuery` switches it to the join channel — **a single-table query is merely the degenerate form of a join**:

```
public class StaffInfoVO {
    private String staffId;
    private String staffName;
    private String organName;    // from the joined table
    // getters/setters
}

List<StaffInfoVO> rows = lightDao.lambdaQuery(StaffInfoVO.class)
        .selectAll(ExcludeCols.of(StaffInfo::getStatus))          // all main-table columns except status
        .from(StaffInfo.class)                                    // required when the result type is a VO
        .leftJoin(OrganInfo.class).on(StaffInfo::getOrganId, OrganInfo::getOrganId)
        .select(JoinCol.of(OrganInfo::getOrganName).as("organName"))  // extra column + result alias
        .eqIfPresent(StaffInfo::getStatus, status)
        .orderByAsc(StaffInfo::getStaffId)
        .find();
```

Generated SQL (columns qualified by their entity alias; the `AS` aliases map to result object properties):

```
SELECT s1.STAFF_ID AS staffId, s1.STAFF_NAME AS staffName, o2.ORGAN_NAME AS organName
FROM sqltoy_staff_info s1 LEFT JOIN sqltoy_organ_info o2 ON s1.ORGAN_ID = o2.ORGAN_ID
WHERE s1.STATUS = :p1
ORDER BY s1.STAFF_ID ASC
```

Join steps

Method	Notes
`from(entityClass)`	Specifies the main table; **may be omitted** — in a query it defaults to the result type (it must be given when the result type is a non-entity such as a VO)
`leftJoin / rightJoin / innerJoin(entityClass)`	Starts a join and returns the on step; the first call switches to the join channel
`on(fieldA, fieldB)`	Equality join; two field pairs may be passed to one join (combined with AND); returns the query builder so further joins or conditions can follow

Table aliases are the entity class initial in lower case plus a sequence number (`s1` , `o2` ...). A join without `on()` , or a condition referencing an entity that does not participate in the join, throws during assembly.

Forms of select

Form	Semantics
no select	When the result type participates in the join, all of its columns are selected (`column AS property`); otherwise `*`
`selectAll()`	Explicitly declares all main-table columns; may coexist with extra `select(JoinCol...)` columns (main-table columns first)
`selectAll(Entity.class)`	Merges all columns of a **joined entity** into select as `alias.column AS property` (columns duplicating the main table are skipped); the entity must participate in the join, otherwise assembly throws
`selectAll(ExcludeCols.of(A::getX, A::getY))`	All main-table columns **excluding** the given ones; excluded fields must belong to the main entity; on the single-table channel it maps to `EntityQuery.unselect`
`select(JoinCol...)`	Explicit columns: `JoinCol.of/count/sum/avg/min/max(field).as(alias)` ; its presence routes to the join channel, and it is also used for single-table grouping/aggregation (with `from(mainTable)` + `groupBy` + `having`)
`select(JoinCol.raw("..."))`	**Raw SQL fragment column**: shapes a typed select cannot express (window functions such as `ROW_NUMBER() OVER(..)` , database-specific functions, `CASE WHEN` , arithmetic expressions). `JoinCol.raw(fragment).as(alias)` , or inline `... AS alias` inside the fragment; it is spliced into SELECT verbatim and triggers the join channel (results map to a VO field with that alias)
`distinct()`	Distinct results: the single-table channel maps to `EntityQuery.distinct()` , the join channel renders `SELECT DISTINCT` (DISTINCT is a quantifier, not a function, so no column-function form is provided)

`JoinCol.raw(...)` fills the blind spots of a typed select (window functions, for instance); the fragment references **real database column names / table aliases** (unqualified column names are fine for a single table, multi-table queries use the generated aliases such as `j1`):

```
// window function: rank by salary, rn maps to the VO's rankNo field
List<StaffRankVO> ranks = lightDao.lambdaQuery(StaffRankVO.class)
        .select(JoinCol.of(StaffInfo::getStaffName).as("staffName"),
                JoinCol.raw("ROW_NUMBER() OVER(ORDER BY SALARY DESC)").as("rankNo"))
        .from(StaffInfo.class)
        .find();
// → SELECT s1.STAFF_NAME AS staffName, ROW_NUMBER() OVER(ORDER BY SALARY DESC) AS rankNo FROM sqltoy_staff_info s1
```

> `JoinCol.raw` and `last(sql)` are both **raw SQL escape hatches**: the framework splices them verbatim without injection screening, so **never splice end-user input into them**; when working across databases, mind the portability of the fragment itself (window function syntax differs between databases).

Grouping and aggregation (sum/avg/min/max/count)

Aggregate columns use `JoinCol`'s function factories together with `groupBy` / `having`; **single-table grouping and aggregation needs no join** — `JoinCol` switches the query to the SQL channel, where `from(mainTable)` names the aggregation subject:

```
public class OrganSalaryStatVO {
    private String organId;
    private BigDecimal totalSalary;
    private BigDecimal avgSalary;
    private Long staffCount;
    // getters/setters
}

List<OrganSalaryStatVO> stats = lightDao.lambdaQuery(OrganSalaryStatVO.class)
        .select(JoinCol.of(StaffInfo::getOrganId).as("organId"),
                JoinCol.sum(StaffInfo::getSalary).as("totalSalary"),
                JoinCol.avg(StaffInfo::getSalary).as("avgSalary"),
                JoinCol.count(StaffInfo::getStaffId).as("staffCount"))
        .from(StaffInfo.class)
        .eqIfPresent(StaffInfo::getStatus, status)
        .groupBy(StaffInfo::getOrganId)
        .having("sum(SALARY) > :min", MapKit.map("min", 10000))   // having is raw SQL; named params via Map
        .find();
```

Aggregating in a join scenario is written the same way, with one extra `leftJoin(...).on(...)`; the `having` text is raw SQL (write real column names), and rows whose grouping key is null take part in the aggregation as well (standard SQL behaviour). To sort aggregate results, sort by the alias:

```
.groupBy(StaffInfo::getOrganId)
.orderByDesc("staffCount")   // sort by the alias defined via "as" in select; only effective on the join channel
```

The single-table channel has no column aliases, so calling an alias sort there throws during assembly; aliases must be plain identifiers (anti-injection).

Result mapping and LEFT JOIN

A column maps when its `AS` alias equals a property name of the result type; columns the result type does not have are ignored. With LEFT JOIN, properties of the joined table are null when there is no matching row.

Terminals and limits of the join channel

`find/getOne/findPage/findTop/findRandom/count/exists` are all available; **the join channel does not inherit entity-level enhancements** (sharding, cache translate, masking, etc.) — an inherent boundary of multi-table SQL.

---

6. Multi-table join update and delete

Join updates/deletes use a **portable EXISTS-over-derived rewrite** (not the MySQL multi-table DML syntax): the join and where conditions go into a derived table, and the outer statement restricts rows by correlating the primary key through EXISTS. That preserves LEFT JOIN semantics in full and covers databases such as Oracle/PostgreSQL which have no multi-table `UPDATE ... JOIN` / `DELETE ... JOIN` syntax at all:

```
UPDATE target_table SET column = :p1 WHERE EXISTS (
    SELECT 1 FROM (SELECT alias.pk AS LPK1 FROM main_table alias LEFT JOIN ... WHERE conditions) sqd
    WHERE sqd.LPK1 = target_table.pk)

DELETE FROM target_table WHERE EXISTS (
    SELECT 1 FROM (SELECT alias.pk AS LPK1 FROM main_table alias LEFT JOIN ... WHERE conditions) sqd
    WHERE sqd.LPK1 = target_table.pk)
```

```
// join update
lightDao.lambdaUpdate(StaffInfo.class)
        .leftJoin(OrganInfo.class).on(StaffInfo::getOrganId, OrganInfo::getOrganId)
        .set(StaffInfo::getStatus, 2)
        .eq(OrganInfo::getOrganName, "R&D")
        .update();
// UPDATE sqltoy_staff_info SET STATUS = :p1 WHERE EXISTS (
//   SELECT 1 FROM (SELECT s1.STAFF_ID AS LPK1 FROM sqltoy_staff_info s1
//                  LEFT JOIN sqltoy_organ_info o2 ON s1.ORGAN_ID = o2.ORGAN_ID
//                  WHERE o2.ORGAN_NAME = :p2) sqd
//   WHERE sqd.LPK1 = sqltoy_staff_info.STAFF_ID)

// join delete: deleteTables names the delete target (defaults to the main table)
lightDao.lambdaDelete(StaffInfo.class)
        .leftJoin(OrganInfo.class).on(StaffInfo::getOrganId, OrganInfo::getOrganId)
        .deleteTables(StaffInfo.class)
        .eq(OrganInfo::getOrganName, "Marketing")
        .delete();
// DELETE FROM sqltoy_staff_info WHERE EXISTS (SELECT 1 FROM (SELECT s1.STAFF_ID AS LPK1 FROM ... ) sqd WHERE sqd.LPK1 = sqltoy_staff_info.STAFF_ID)
```

- The **main table of a join update/delete is implicitly the builder entity**, so `from()` is unnecessary;
- **The target entity must have an `@Id` primary key** (a composite key correlates through `LPK1…LPKn`): the EXISTS rewrite joins back to the target table by primary key, and an entity without one throws `DataAccessException` during assembly;
- The rewrite is verified against real relational databases (mysql/postgresql/sqlserver/oracle/db2 — part of the release multi-DB gate); clickhouse/doris/starrocks and other databases without transactional row-level DML are not supported and must use SQL mode instead;
- `setExpr` expressions in a join update are **emitted verbatim** (the single-table channel's automatic property-to-column conversion does not apply), so write real column names; a single `?` can still bind a value;
- The join channel is likewise subject to the unconditional-write guard (`matchAll()` allows it).

6b. Composed queries: CTE (WITH AS) / @fast pagination pushdown / UNION

lambda can compose **several lambdaQueries** into one big SQL statement: CTE (`WITH ... AS`), @fast page pushdown derived tables, UNION/UNION ALL. The branches **share the parameter sequence** (inner branches are assembled first, the outer statement continues the numbering), so parameter names are globally unique.

CTE (WITH AS)

```
// inner CTE body: an independent join lambdaQuery that is not executed but only contributes SQL
LambdaQuery<CteDept> inner = lightDao.lambdaQuery(CteDept.class).from(CteDept.class)
        .select(JoinCol.of(CteDept::getDeptId).as("deptId"), JoinCol.of(CteDept::getDeptName).as("deptName"))
        .eq(CteDept::getStatus, 1);

// outer query: register the CTE with with and reference it via fromCte; the outer type maps to a VO
List<CteDeptVO> rows = lightDao.lambdaQuery(CteDeptVO.class)
        .with("d", inner)
        .fromCte("d")
        .eq(CteDeptVO::getDeptName, "R&D Dept")
        .find();
// → WITH d AS (SELECT c1.DEPT_ID AS deptId, c1.DEPT_NAME AS deptName FROM cte_dept c1 WHERE c1.STATUS = :p1)
//   SELECT c2.DEPTNAME AS deptName, c2.DEPTID AS deptId FROM d c2 WHERE c2.DEPTNAME = :p2
```

- **Several CTEs / a CTE referencing another CTE**: chain `.with("a", qa).with("b", qb)` and let `qb` use `fromCte("a")` → `WITH a AS (..), b AS (..)`.
- **Recursive CTE**: `.withRecursive(name, q, "id", "pid", "name")` → `WITH RECURSIVE name(id, pid, name) AS (..)`; **the column list is strongly recommended** — recursive CTEs on H2 and others require it, and PG/MySQL/SQLServer support it as standard. Oracle/SQLServer recursive CTEs omit the `RECURSIVE` keyword, which the framework adapts by target database type. The recursive branch must join the CTE to itself with **innerJoin** (a LEFT JOIN produces a row with no child and makes the recursion never terminate), and the join direction is "child.pid = current node.id" walking downwards.
- **PostgreSQL-family materialization hint**: `.with(name, q, Boolean.TRUE/FALSE)` renders `AS MATERIALIZED` / `AS NOT MATERIALIZED` on the PostgreSQL family (postgresql/postgresql14 and same-kernel openGauss/Kingbase/Vastbase/GAUSSDB/HighGo — whitelist in

`SqlWithAnalysis.supportsCteMaterialized`); **other databases ignore it** (decided by dbType — not rendered, no error). `AS [NOT] MATERIALIZED` inside SQL templates follows the same convention: stripped at parse time when the target database does not support it (since 6.0.9).

- **Position constraint on WITH**: the generated `WITH` is always at position 0 — sqltoy detects WITH textually (`SqlWithAnalysis` / `hasWith`) to strip it outside the wrapper for count/pagination, and a leading semicolon would break that detection; a single SQLServer statement starting with WITH needs no leading semicolon anyway (the semicolon requirement only applies to batches with an unterminated statement before the WITH).
- The outer column resolution uses the **result VO's EntityMeta** (a CTE is not a real table and has no EntityMeta), so the outer query must be typed to a VO whose fields align with the CTE output columns; databases without CTE support must be handled by the developer (the framework does not degrade).

@fast pagination pushdown with derived tables

`@fast` semantics: **first paginate the inner @fast block to pageSize rows, then join those rows to outer tables** (avoiding a full join followed by pagination); the pagination count defaults to the inner @fast SQL (the simplified form). lambda expresses this as a derived table:

```
LambdaQuery<StaffInfo> inner = lightDao.lambdaQuery(StaffInfo.class).from(StaffInfo.class)
        .eq(StaffInfo::getStatus, 1).orderByDesc(StaffInfo::getEntryDate);   // inner: filter + sort (pushdown subject)

Page<StaffOrganVO> page = lightDao.lambdaQuery(StaffOrganVO.class)
        .fromFast(inner, "t1")                                  // FROM @fast(inner sql) t1
        .leftJoin(OrganInfo.class).on(StaffOrganVO::getOrganId, OrganInfo::getOrganId)
        .findPage(new Page(10, 1));
// → select t1.*, t2.ORGAN_NAME from @fast(select ... from staff t where t.STATUS=:p1 order by t.ENTRY_DATE desc) t1
//   left join organ t2 on t1.organ_id=t2.ORGAN_ID   (limit/offset is pushed into the @fast inner block by sqltoy)
```

`@fast` can also be nested inside a CTE body: `.with("tmp", q)` where `q` internally uses `fromFast(...)` → `WITH tmp AS (SELECT * FROM @fast(..) t1 ...)`.

An even leaner count: `.countSql("select count(1) from staff where status=:p1")` supplies a hand-written count that is leaner than the inner @fast count (effective only for `findPage` on the join/SQL channel), corresponding to sqltoy's `count-sql` configuration.

UNION / UNION ALL

```
LambdaQuery<NameVO> q1 = lightDao.lambdaQuery(NameVO.class).from(Staff.class)
        .select(JoinCol.of(Staff::getName).as("name")).eq(Staff::getStatus, 1);
LambdaQuery<NameVO> q2 = lightDao.lambdaQuery(NameVO.class).from(Staff.class)
        .select(JoinCol.of(Staff::getName).as("name")).eq(Staff::getStatus, 2);

q1.unionAll(q2).find();          // (q1) UNION ALL (q2)  no dedup, faster
q1.union(q2).count();            // (q1) UNION (q2)     dedup
q1.unionAll(q2).findPage(page);  // wrapped as a sub-query then paginated; count likewise
```

- Both sides must map to **the same result type** with identical select columns/aliases, and both are join/SQL channel queries (requiring `from` / `select`).
- **Every query terminal applies to the composed SQL** (`find/getOne/findPage/findTop/findRandom/count/exists` all take the composed SQL from `getSql()`, so CTE/UNION work with all of them); UNION ALL counts reuse sqltoy's union-all-count optimization (summing the branches).

> Composed queries (CTE/@fast/UNION) are all join/SQL channel and **do not inherit entity-level enhancements** (sharding/translate/masking/tenant); the raw SQL fragments and aliases in CTE/@fast/UNION are the developer's responsibility for dialect portability.

---

7. Parameter and SQL-shape conventions

- Condition parameters are uniformly named `:p1、:p2…`, unique across the whole builder (nested conditions and the set/where parts of a join share the sequence), which keeps SQL logs readable;
- External named parameters of `having` and `inSql` should **avoid the `p` + pure-digit form** to prevent collisions with internal parameter names;
- Nested conditions are parenthesized as a whole; when only grouping/sorting is present, `1 = 1` is used to complete the where clause;
- lambda extraction only supports **getter method references** (`User::getName`, `User::isActive`); an expression lambda (`u -> u.getName()`) cannot yield a property name and fails fast.

7b. Safety boundaries and usage conventions

- **Raw SQL escape hatches are not injection-screened**: `last(sql)`, `having(sql, params)`, `inSql(col, sql, params)` and `setExpr(col, expr)` accept raw SQL text (at the same trust level as sqlId text), so **never concatenate end-user input**; always bind parameters by name;

- **Parameter namespaces**: internal condition parameters are `:p1`、`:p2`…; external parameter names of `having` / `inSql` may not use the `p` + pure-digit form, validated at call time;
- **Select alias uniqueness**: a user alias (`JoinCol.as(...)` or the property form of select) that duplicates any existing alias throws (result-mapping ambiguity); duplicate column names among framework dumps (`selectAll()` / `selectAll(Class)`) are **skipped in declaration order with the main table winning** (e.g. when both the main table and a joined table have a `status` column, the joined one is skipped) — use `JoinCol.as` to rename explicitly when you need the joined column;
- **Self-joins are not supported**: registering the same entity twice (including `from` followed by joining the same class) throws, because the lambda join cannot generate two aliases for one entity;
- **When the result type is a VO**, the property form `select(VO::getX)` is unavailable on the join channel (the VO does not participate in the join, so no alias can qualify it) and throws; for VOs use `JoinCol...as()` or `selectAll/selectAll(Class)`;
- **Channel restriction of `last(sql)`**: on the join channel it is appended to the end of the SQL normally; on the single-table channel (where the `EntityQuery` where clause cannot carry trailing raw SQL) calling it throws — use `findTop/findPage` or the join channel instead;
- **External parameter namespace**: two `inSql` / `having` sites sharing one named parameter with **different values** throws (silent overwriting would produce wrong results); sharing it with the same value is allowed;
- **Proxy lambdas are unsupported**: lambdas produced by the IDEA Evaluate window or Mockito mocks cannot have their metadata resolved under JDK module encapsulation and throw, with a hint to add `--add-opens java.base/java.lang.invoke=ALL-UNNAMED` or switch to a method reference;
- **Builders are single-use and not thread-safe**: one builder instance corresponds to one assembly; calling structural mutators such as `from/leftJoin/on` after a terminal has executed (assembled) throws, and further conditions have no effect; create a new builder per request under concurrency;
- **Fields must be mapped properties**: properties referenced in conditions/select/set must be `@Column`-mapped fields; unmapped properties (helper getters, `getClass`, etc.) throw during assembly instead of silently producing `null = :p1`;
- **Update/delete builders reject grouping and sorting**: `groupBy/having/orderBy` are supported only by `LambdaQuery`; calling them on `LambdaUpdate/LambdaDelete` throws during assembly;
- **`not(...)` and the preceding condition get an automatic AND**: `eq(a).not(n -> ...)` renders as `a AND NOT (...)`; a dangling NOT from an empty `not(...)` is stripped;
- **Delete-channel fallback for a parameterless where**: same rule as update (conditions without named parameters such as `isNull`, and the `matchAll` case, get `1=:DEFAULT_VALUE` automatically), so `deleteByQuery`'s empty-value rejection is never triggered;
- **The primary key may be set explicitly**: an explicit `set/setExpr` on the primary key counts as intent (natural-key correction, etc.) and is not blocked; setting several rows to the same key is rejected by the database constraint; whole-entity update's primary-key safety is guaranteed by `lightDao.update(entity)` (the key goes into the where clause).

8. Error semantics at a glance

Scenario	Behaviour
`getOne()` matches more than one row	Throws (including the match count), hinting at checking condition uniqueness or using `find()`
Update/delete with no where condition	Throws; an explicit `matchAll()` is required
Condition/select/set field belongs to an entity outside the join	Throws: the entity does not participate in the current query/join
A join without `on()`	Throws during assembly
Join update/delete without a main table	Throws during assembly (a query may default the main table to the result type)
`setExpr` expression contains `=`	Throws at call time
`setExpr` expression contains a quoted literal `?`	Throws at call time (the update channel replaces `?` textually, so a quoted `?` would be mistaken for a parameter slot)
`ExcludeCols` field not belonging to the main entity	Throws during assembly
`selectAll(ExcludeCols...)` excludes every column	Throws during assembly
Terminal called on a builder not bound to a dao	Throws, hinting at how it must be created

9. Migrating from sqltoy-plus

sqltoy-plus	lambda conditions
`Wrappers.lambdaQuery(entityClass)`	`LambdaQuery.create(entityClass)` or `lightDao.lambdaQuery(entityClass)`
`LambdaQueryWrapper / LambdaUpdateWrapper`	`LambdaQuery / LambdaUpdate`
`SqlToyPlusDao.findList/findOne/findPage/count/exists/delete/update(wrapper)`	Builder terminals: `lambdaQuery(...).find()/getOne()/findPage()/count()/exists()`, `lambdaUpdate(...).update()`, `lambdaDelete(...).delete()`
`lambdaQueryChain/lambdaUpdateChain`	`lightDao.lambdaQuery/lambdaUpdate` (the factory binds the dao; go straight to a terminal)
`Wrappers.lambdaMultiWrapper(toClass).select(...).from(...)`	`lightDao.lambdaQuery(toClass)[.selectAll(...)].from(...).leftJoin(...).on(...)`
`LambdaColumn`	`JoinCol`
`lambdaUpdateMultiWrapper / lambdaDeleteMultiWrapper`	`lightDao.lambdaUpdate/lambdaDelete` + join steps
`SFunction`	`LambdaFn`
`FiledValueFilterStrategy` (global static)	builder-instance `valueFilter(...)`
parameter names of the `type.valuePairs.MPG1` form	`:p1…`
string-column wrappers (`QueryWrapper` etc.)	no longer provided: use `EntityQuery` for single-table string conditions and SQL mode for complex dynamic SQL
`@EnableSqlToyPlus` / mapping-strategy beans	no configuration needed; the capability ships with `LightDao`

10. Boundaries (what it does not do)

- **String condition DSL**: the lambda layer accepts method references only; use `EntityQuery/EntityUpdate` for single-table string conditions and SQL mode for complex dynamic SQL;
- **Entity-level enhancements do not apply on the join channel** (sharding/translate/masking/tenant);
- **Join update/delete on OLAP databases**: clickhouse/doris/starrocks and other databases without transactional row-level DML are out of scope (a database capability boundary, at the developer's discretion); relational databases use the portable EXISTS-over-derived rewrite, verified on real mysql/postgresql/sqlserver/oracle/db2 databases;
- **Whole-entity merged updates**: update a whole object by primary key with `lightDao.update(entity)`; the lambda layer only offers explicit `set/setExpr` fields;
- **Interceptor/plugin system**: lambda conditions ultimately run through sqltoy's existing execution path, so plugins (`SqlInterceptor` etc.) keep working, but the lambda layer itself offers no extension points.

Pagination Optimize

sqltoy provides a multi-layer pagination mechanism: **intelligent count optimization, cached pagination, fast pagination, and parallel pagination**, and also supports custom count SQL and skipping count. Developers do not need to worry about pagination dialect differences across databases — sqltoy adapts automatically.

Overall flow:

```
flowchart TD
    A[Pagination request findPage] --> B[Intelligently rewrite the count SQL<br/>strip order by · simplify select]
    B --> C{page-optimize configured?}
    C -->|Yes| D{Cache hit?}
    D -->|Hit| F[Reuse the cached count result]
    D -->|Miss| G[Execute count and write it into the cache]
    C -->|No| H[Execute the optimized count directly]
    F --> I[Run the single-page data query in parallel]
    G --> I
    H --> I
    I --> J[Assemble the Page and return<br/>rows + recordCount]
```

1. Basic Pagination API

```
// 1) paginated query with Map parameters
Page<StaffInfo> page = lightDao.findPage(new Page(10, 1), "sqltoy_staff_find", paramsMap, StaffInfo.class);

// 2) flexible pagination with QueryExecutor (datasource, cache translate, formatting, pagination optimize, etc. can be specified)
QueryResult result = lightDao.findPageByQuery(new Page(10, 1),
        new QueryExecutor("sqltoy_staff_find").names("status").values(1).resultType(StaffInfo.class));

// 3) single-table entity pagination
Page<SysLog> page2 = lightDao.findPageEntity(new Page(10, 1), SysLog.class,
        EntityQuery.create().where("status=:status").names("status").values(1));
```

> `Page` is constructed as `new Page(pageSize, pageNo)`; for example `new Page(10, 1)` means 10 rows per page, page 1.

Common `Page` methods:

Method	Description
`getRows()`	Data collection of the current page
`getRecordCount()`	Total record count
`getPageNo()` / `getPageSize()`	Current page number / rows per page
`getTotalPage()` / `getLastPage()`	Total pages / last page number
`getNextPage()` / `getPriorPage()` / `getFirstPage()`	Next page / previous page / first page
`getStartIndex()` / `getNextIndex()` / `getPreviousIndex()`	Start / next / previous index
`setSkipQueryCount(Boolean)`	Whether to skip the total count query (see Section 6)
`setOverPageToFirst(Boolean)`	Whether to return to the first page when paging goes beyond the last page

---

2. Intelligent Count Optimization (Automatic)

Pagination is essentially a two-step query: ① **query the total record count,** ② **query the data of a specific page**. The count query is often the performance bottleneck of pagination on large tables.

sqltoy's count is **not** a blunt `select count(1) from (your sql) as t`; instead it intelligently rewrites:

- **Strips unnecessary** `order by` clauses according to the SQL logic;
- Detects whether the `select` columns and function expressions before `from` in the original SQL can be **directly cut away**, replaced with `select count(1) from ...`, avoiding pointless function evaluation;
- Automatically adapts to the pagination dialect of each database (limit / rownum / offset fetch / top, etc.).

In the vast majority of scenarios, this automatic optimization already significantly reduces the count overhead with no extra configuration.

---

3. Cached Pagination (page-optimize)

For scenarios where **the query conditions are identical and paging happens frequently**, re-running count on every page turn is wasteful. `<page-optimize>` caches "identical query conditions → count result" so that, within a certain period, paging directly reuses the count without querying the total again.

```
<sql id="sqltoy_staff_find">
    <!-- parallel: whether to execute the count and the single-page data query in parallel;
         alive-max: how many groups of counts for different query conditions can be cached at most; alive-seconds: cache survival seconds -->
    <page-optimize parallel="true" alive-max="100" alive-seconds="120" />
    <value><![CDATA[
        select t.* from sqltoy_staff_info t
        where t.STATUS=1
        #[and t.STAFF_NAME like :staffName]
        order by t.ENTRY_DATE desc
    ]]></value>
</sql>
```

`<page-optimize>` attributes:

Attribute	Default	Description
`parallel`	`false`	Whether to execute the count and the result query in parallel for better efficiency (cache optimization is disabled when `alive-max=1`)
`parallel-maxwait-seconds`	—	Maximum wait seconds for the parallel query, to avoid running too long
`alive-max`	`100`	Maximum number of count records cached for different query conditions
`alive-seconds`	`90`	Survival seconds in memory for each group of query conditions and its count; when expired it is queried again
`skip-zero-count`	`false`	When a cached count of 0 is hit, whether to skip that cache entry and re-fetch the total count

The Java counterpart is `QueryExecutor.pageOptimize(PageOptimize)`:

```
lightDao.findPageByQuery(new Page(10, 1),
        new QueryExecutor("sqltoy_staff_find").pageOptimize(new PageOptimize(100, 120)));
```

---

4. Fast Pagination (@fast / @fastPage)

Paginate first, then join: when the main query needs to join other tables (especially one-to-many), the usual approach is to join into a large result set first and then paginate, which is expensive. `@fast()` wraps the part that "determines the result set" in a subquery, **first takes the paginated records by pageSize, then joins with the other tables**, greatly reducing the amount of joined data.

```
<sql id="sqltoy_staff_findlist">
    <value><![CDATA[
        select t1.*,t2.ORGAN_NAME
        -- @fast() first takes pageSize rows by pagination, then joins
        from @fast(
              select t.* from sqltoy_staff_info t
              where t.STATUS=1
                #[and t.STAFF_NAME like :staffName]
              order by t.ENTRY_DATE desc
             ) t1
        left join sqltoy_organ_info t2 on t1.organ_id=t2.ORGAN_ID
    ]]></value>
</sql>
```

Key points:

- `@fast` and `@fastPage` are **equivalent**.
- `@fast()` is written directly in the SQL; for **non-paginated queries the framework automatically removes** `@fast`, so it does not affect other queries such as find/findTop.
- It also applies to `findTop` and `findRandom` (getRandomResult).
- It can be combined with extracting 1..n times the data: e.g. with `pageSize=10`, take 50 rows and then perform row-to-column pivoting and other secondary computation.

---

5. Parallel Pagination (parallel)

With `<page-optimize parallel="true">`, sqltoy **executes the count query and the single-page data query in parallel**, compressing the originally serial two steps into the duration of one, suitable for scenarios where both the count and the data query are heavy. Use `parallel-maxwait-seconds` to limit the maximum parallel wait time.

---

6. Custom count-sql and Skipping Count

Custom count SQL: in extremely special cases, you can hand-write the optimal count statement:

```
<sql id="sqltoy_staff_find">
    <value><![CDATA[ select ... complex query ... ]]></value>
    <count-sql><![CDATA[ select count(1) from sqltoy_staff_info t where t.STATUS=1 ]]></count-sql>
</sql>
```

Skipping count: for scenarios that do not need the total, such as scroll loading and infinite dropdowns, the count query can be skipped to improve performance:

```
Page page = new Page(20, 1);
page.setSkipQueryCount(true);   // do not query the total record count
Page result = lightDao.findPage(page, "sqltoy_staff_find", paramsMap, StaffInfo.class);
```

> For pagination-related source code, refer to `org.sagacity.sqltoy.dialect.PageOptimizeUtils` (the count optimization and fast pagination engine) and the `Dialect` implementations of the individual databases.

Complex Query & Analytics

sqltoy provides a set of analytics capabilities that **perform secondary computations on result sets in memory**: multi-dimensional group summary, pivot/unpivot, YoY & MoM comparisons, tree sort with layer-by-layer rollup, group concatenation, and more.

These algorithms deliver two standout benefits:

1. **Offloads the database**: computations such as row-column conversion and layer-by-layer rollup would cause huge CPU and memory overhead if done by the database; sqltoy processes them in a single linear pass in application memory with high efficiency.
2. **Cross-database compatibility**: the algorithms do not depend on any database dialect at all. The same SQL runs on MySQL and Oracle and migrates smoothly to domestic and mainstream databases such as Dameng (DM), GaussDB, and PostgreSQL.

All analytics features can be declared with tags in `*.sql.xml` or built fluently in Java with `QueryExecutor`; the two approaches are fully equivalent.

> The examples below all use a fruit sales record table `sqltoy_fruit_order` for demonstration.

1. Multi-Dimensional Group Summary (summary)

1.1 Use Case

In financial reports or sales statistics, subtotals, grand totals, or averages are usually appended at the bottom of the table or at the end of each group.

- **Traditional pain point**: SQL `GROUP BY ROLLUP` is not only cumbersome to write, its dialects also differ greatly across Oracle, MySQL, and SQL Server, making it extremely hard to maintain.
- **The sqltoy way**: the SQL only fetches the base detail rows, while sqltoy automatically completes multi-dimensional summarization and average calculation in memory.

1.2 XML Configuration Example

```
<sql id="group_summary_case">
    <value><![CDATA[
        select t.fruit_name,t.order_month,t.sale_count,t.sale_quantity,t.total_amt
        from sqltoy_fruit_order t
        order by t.fruit_name,t.order_month
    ]]></value>
    <!-- sum-columns: columns to sum; average-columns: columns to average; reverse:true puts summary rows on top -->
    <summary sum-columns="sale_count,sale_quantity,total_amt" reverse="true">
        <!-- global: grand total; label-column: which column holds the total label; sum-label: total label text -->
        <global label-column="fruit_name" sum-label="Total" />
        <!-- group: subtotal per group; group-column: the grouping column -->
        <group group-column="fruit_name" label-column="fruit_name" sum-label="Subtotal" />
    </summary>
</sql>
```

1.3 Attributes

`<summary>` tag attributes:

Attribute	Description
`sum-columns`	Columns to sum, comma-separated
`average-columns`	Columns to average
`average-radix-sizes`	Decimal places kept for each average column, e.g. `1,3,2`; a single value applies to all
`average-rounding-modes`	Rounding modes: `HALF_UP` / `HALF_DOWN` / `ROUND_DOWN` / `ROUND_UP`
`average-skip-null`	Whether to exclude null values when averaging, default `false`
`sum-site`	Position of the summary row: `top` / `bottom` / `left` / `right`, default `top`
`reverse`	Whether the data is arranged in reverse order, default `false`
`link-sign`	Concatenation symbol when sum and average are shown in the same row, default `/`
`skip-single-row`	Whether single-row data also participates in summarization, default `false`
`has-grouped`	Whether the data has already been grouped and organized, default `true`

`<global>` (grand total): `label-column` (required), `sum-label` (required), `average-label`, `reverse`.

`<group>` (group subtotal, multiple allowed): `group-column` (required), `label-column` (required), `sum-label` (required), `average-label`, `order-column` (column used to order the groups), `order-with-sum` (default `true`), `order-way` (`desc` / `asc`).

1.4 Effect

Category	Sale Month	Sales Count	Sale Quantity (tons)	Sale Amount (10K CNY)
Total		71	12600	14900
Subtotal		36	5900	7500
Apple	2019-05	12	2000	2400
Apple	2019-04	11	1900	2600
Apple	2019-03	13	2000	2500
Subtotal		35	6700	7400
Banana	2019-05	10	2000	2000
Banana	2019-04	12	2400	2700
Banana	2019-03	13	2300	2700

---

2. Dynamic Pivot

2.1 Use Case

Converts a "narrow table" in the database (multiple row records) into a "wide table" for front-end display (multiple columns). For example, row data of "month, product, sales volume" is expanded horizontally into "product, March sales, April sales, May sales".

- **Traditional pain point**: the database `PIVOT` syntax cannot cope with dynamic, non-fixed columns; once months or product types change dynamically, pure SQL breaks down.
- **The sqltoy way**: combining its core algorithms, sqltoy supports fully dynamic pivot with just a few declarative lines.

2.2 XML Configuration Example

```
<sql id="pivot_case">
    <value><![CDATA[
        select t.fruit_name,t.order_month,t.sale_count,t.sale_quantity,t.total_amt
        from sqltoy_fruit_order t
        order by t.fruit_name,t.order_month
    ]]></value>
    <!-- group-columns: row grouping columns; category-columns: horizontal category columns;
         start-column~end-column: range of metric columns to rotate (3 columns from sale_count to total_amt) -->
    <pivot group-columns="fruit_name" category-columns="order_month"
           start-column="sale_count" end-column="total_amt" />
</sql>
```

`<pivot>` attributes: `group-columns` (required), `category-columns` (required), `start-column` (required), `end-column` (required), `category-sql` (generate the category columns dynamically via SQL), `default-value`, `default-type`.

> [!TIP] For a plain pivot that meets the default regular criteria, you can simply write `<pivot/>` without the lengthy attributes.

2.3 Effect

Category	2019-03			2019-04			2019-05		
	Count	Quantity	Total Amount	Count	Quantity	Total Amount	Count	Quantity	Total Amount
Banana	13	2300	2700	12	2400	2700	10	2000	2000
Apple	13	2000	2500	11	1900	2600	12	2000	2400

---

3. Unpivot

3.1 Use Case

The opposite of `pivot`. Raw data imported from Excel is often a wide table (e.g. student ID, Chinese score, math score, English score); for easier cleansing, loading, or fine-grained analysis, it needs to be converted into a narrow table (student ID, subject, score).

3.2 XML Configuration Example

```
<sql id="sys_student_score_unpivot">
    <value><![CDATA[
        select t.student_no,t.chinese_score,t.math_score,t.english_score
        from sys_student_score t
    ]]></value>
    <!-- columns-to-rows: "column name:metric name", comma separated; new-columns-labels: headers of the two newly created columns -->
    <unpivot columns-to-rows="chinese_score:Chinese,math_score:Math,english_score:English"
             new-columns-labels="subject_name,score_value" />
</sql>
```

`<unpivot>` attributes: `columns-to-rows` (required, format `col1:metricName1,col2:metricName2`), `new-columns-labels` (required, sets the headers of the 2 newly created columns so they can be mapped to VO properties, e.g. `indexName,indexValue`).

---

4. YoY and MoM Calculations

4.1 Use Case

In BI dashboards, merely displaying sales figures is not enough; you often need to show the **MoM growth rate** (this month vs. last month) or the **YoY growth rate**. sqltoy provides two tags — `<cols-chain-relative>` (column-level MoM, horizontal comparison among like columns) and `<rows-chain-relative>` (row-level MoM, vertical comparison between rows) — which compute and output percentage/per-mille values automatically.

4.2 Column-Level MoM (pivot first, then ratio)

```
<sql id="cols_relative_case">
    <value><![CDATA[
        select t.fruit_name,t.order_month,t.sale_count,t.sale_quantity,t.total_amt
        from sqltoy_fruit_order t
        order by t.fruit_name,t.order_month
    ]]></value>
    <!-- pivot first: each month has three metrics: count/quantity/total amount -->
    <pivot group-columns="fruit_name" category-columns="order_month"
           start-column="sale_count" end-column="total_amt" />
    <!-- ratio between columns: group-size=3 means 3 metrics per group; relative-indexs=1,2 means computing
         the MoM ratio for the 2nd and 3rd metrics; start-column=1 starts from the 1st column -->
    <cols-chain-relative group-size="3" relative-indexs="1,2" start-column="1" format="#.00%" />
</sql>
```

`<cols-chain-relative>` attributes: `group-size` (required), `relative-indexs` (required), `start-column` (required), `end-column` (negative values count from the end), `reduce-one` (whether to subtract 1: `(B-A)/A` instead of `B/A`), `multiply` (`1` decimal / `100` percent / `1000` per-mille), `radix-size` (decimal places kept, default 3), `rounding-mode`, `format` (`#.00%` / `#.00‰`), `defaultValue` (default value for the first group where no comparison value exists).

Effect (excerpt):

Category	2019-03					2019-04					2019-05				
	Count	Quantity	vs. Last Month	Total Amount	vs. Last Month	Count	Quantity	vs. Last Month	Total Amount	vs. Last Month	Count	Quantity	vs. Last Month	Total Amount	
Banana	13	2300		2700		12	2400	4.30%	2700	0.00%	10	2000	-16.70%	2000	
Apple	13	2000		2500		11	1900	-5.00%	2600	4.00%	12	2000	5.30%	2400	

4.3 Row-Level MoM

```
<sql id="sales_rows_relative">
    <value><![CDATA[
        select t.sale_date,t.sales_amt from sales_table t order by t.sale_date asc
    ]]></value>
    <!-- group-column: grouping column (use -1 if there is no grouping); relative-columns: columns to compare;
         insert:true means the MoM value is added as a separate new column -->
    <rows-chain-relative group-column="sale_date" relative-columns="sales_amt"
                         format="#.00%" reduce-one="true" multiply="100" insert="true" />
</sql>
```

`<rows-chain-relative>` attributes: `group-column` (required, a column name or numeric index; use `-1` for a single-type comparison without grouping), `relative-columns` (required), `relative-indexs`, `start-row`, `end-row` (negative values count from the end), `reduce-one`, `reverse` (default `true`), `multiply`, `radix-size`, `rounding-mode`, `format`, `insert` (default `true`, whether the MoM value automatically adds a new column), `defaultValue`.

---

5. Tree Sort & Rollup (tree-sort)

5.1 Use Case

For tree tables with hierarchical relationships (organization structures, account categories, administrative divisions), the front end usually requires rows to be ordered by tree depth and lineage (Head Office → East China Branch → Shanghai Office ...), and **parent nodes automatically roll up the data of all their descendant nodes**.

5.2 XML Configuration Example

```
<sql id="treeTable_sort_sum">
    <value><![CDATA[
        select t.area_code,t.pid_area,sale_cnt from sqltoy_area_sales t
    ]]></value>
    <!-- builds the tree hierarchy, rolls up child node values layer by layer to parent nodes,
         and orders nodes within the same level by sale_cnt descending -->
    <tree-sort id-column="area_code" pid-column="pid_area" sum-columns="sale_cnt"
               level-order-column="sale_cnt" order-way="desc" />
</sql>
```

`<tree-sort>` attributes: `id-column` (required), `pid-column` (required), `sum-columns` (columns rolled up layer by layer), `level-order-column` (column used for ordering within each level), `order-way` (`desc` / `asc`, default `desc`). It can also contain the child tag `<sum-filter column="..." compare-type="eq|neq|gt|gte|lt|lte|between|in|out" compare-values="..."/>` to filter which nodes participate in the rollup.

5.3 Effect

Region	Parent Region	Sales Quantity
Shanghai	China	300
Songjiang	Shanghai	120
Yangpu	Shanghai	116
Pudong	Shanghai	64
Jiangsu	China	270
Nanjing	Jiangsu	110
Suzhou	Jiangsu	90
Wuxi	Jiangsu	70

---

6. Group Concatenation (link, replacing group_concat)

`group_concat` / `listagg` / `string_agg` are written differently across databases; sqltoy performs group concatenation in memory with the `<link>` tag, achieving consistent behavior across databases.

```
<sql id="organ_staff_link">
    <value><![CDATA[
        select t.organ_id,t.staff_name from sqltoy_staff_info t order by t.organ_id
    ]]></value>
    <!-- id-columns: grouping columns; columns: columns to concatenate; sign: separator; distinct: deduplicate or not -->
    <link id-columns="organ_id" columns="staff_name" sign="," distinct="true" />
</sql>
```

`<link>` attributes: `columns` (required, multiple columns allowed, e.g. `staff_id,staff_name`), `id-columns` (grouping columns, multiple allowed; make sure the SQL sorts properly), `sign` (separator, default comma), `distinct` (default `false`), `result-type` (`ARRAY` / `LIST` / `SET`, only applicable to a single field, presenting the result as a collection instead of a concatenated string).

> The concatenation order follows the result-set order — **make sure the SQL is sorted by the `id-columns` grouping columns**, otherwise the concatenated result is unpredictable.

Multiple columns + custom separator (with multiple `columns` they are concatenated in order; `sign` accepts any character):

```
<sql id="order_items_link">
    <value><![CDATA[
        select t.order_id, t.product_name, t.qty
        from sqltoy_order_item t
        order by t.order_id
    ]]></value>
    <!-- group by order, concatenate product names and qty into one row -->
    <link id-columns="order_id" columns="product_name,qty" sign="|" />
</sql>
```

Collection result (`result-type` applies to a single field only; returns an `ARRAY` / `LIST` / `SET` collection instead of a string):

```
<sql id="staff_tags_link">
    <value><![CDATA[
        select t.organ_id, t.staff_tag from sqltoy_staff_tag t order by t.organ_id
    ]]></value>
    <link id-columns="organ_id" columns="staff_tag" result-type="ARRAY" />
</sql>
```

Java form (QueryExecutor.groupConcat):

```
QueryResult result = lightDao.findByQuery(
        new QueryExecutor("organ_staff_link")
            .groupConcat(new GroupConcat()
                .group("organ_id")        // equivalent to id-columns
                .concat("staff_name")     // equivalent to columns
                .separator(",")           // equivalent to sign
                .distinct(true)));        // equivalent to distinct
List rows = result.getRows();
```

6b. Reversing the Result Set (reverse)

`<reverse>` reverses the result set **in place** (the whole set by default, or only a given row range). The typical case is "the SQL takes the most recent N rows in descending time order, but the page must display them in ascending order" — no extra sub-query needed.

```
<sql id="recent_orders">
    <value><![CDATA[
        select t.order_id,t.trans_date from sqltoy_order_info t
        order by t.trans_date desc
    ]]></value>
    <!-- reverse only the first 10 rows (0-based): "latest 10, displayed in ascending order" -->
    <reverse start-row="0" end-row="9" />
</sql>
```

Attributes:

Attribute	Notes
`start-row`	First row to reverse, **0-based**, defaults to 0
`end-row`	Last row (inclusive), defaults to the last row; **a negative value counts from the end** (-1 is the last row)

> With fewer than 2 rows, a `start-row` beyond the last row, or an invalid range, the result is returned unchanged — no exception.

---

7. Java API (QueryExecutor)

All of the XML analytics tags above have corresponding `QueryExecutor` fluent methods that can be built dynamically in code: `summary()`, `pivot()`, `unpivot()`, `colsChainRatio()`, `rowsChainRatio()`, `treeSort()`, `groupConcat()`, plus formatting `numFmt()` / `dateFmt()`, masking `secureMask()`, and more.

```
@Autowired
private LightDao lightDao;

public void getComplexReport() {
    QueryResult result = lightDao.findByQuery(
        new QueryExecutor("group_summary_case")
            .names("startDate").values(LocalDate.now().minusMonths(3))
            // group summary: equivalent to the XML <summary>
            .summary(new Summary()
                .sumColumns("sale_count", "sale_quantity", "total_amt")
                .reverse(true)
                .summaryGroups(new SummaryGroup("fruit_name")
                    .labelColumn("fruit_name").sumTitle("Subtotal")))
            // column-level MoM: equivalent to <cols-chain-relative>
            .colsChainRatio(new ColsChainRatio()
                .groupSize(3).relativeIndexs(1, 2).startColumn("1").format("#.00%"))
            // number/date formatting
            .numFmt("#,###.00", RoundingMode.HALF_UP, "total_amt")
            .dateFmt("yyyy-MM-dd", "order_month")
    );

    List rows = result.getRows();          // final result set after the in-memory computation
    long executeTime = result.getExecuteTime(); // SQL execution time
}
```

> Tree rollup and group concatenation work the same way: `.treeSort(new TreeSort().idColumn("area_code").pidColumn("pid_area").sumColumns("sale_cnt").levelOrderColumn("sale_cnt").orderWay("desc"))`, `.groupConcat(new GroupConcat().group("organ_id").concat("staff_name").separator(",").distinct(true))`.

Parallel Query

1. Applicable Scenarios

Parallel query is used to **execute several SQL statements that have no dependencies on each other at the same time**, turning sequential execution into parallel execution to improve overall efficiency (e.g. a page that needs to load multiple independent statistics blocks at once).

Notes:

- **Do not use it in the middle of a transaction.**
- Multiple SQL statements **share the same parameter conditions** (the union of several query parameters); sqltoy automatically extracts the parameters each SQL actually needs.
- Supports paginated queries, non-paginated queries, and a mix of both: setting a `Page` on a `ParallelQuery` marks that query as paginated.

2. API

```
// returns the QueryResult of each query, in the same order as the given ParallelQuery list
public <T> List<QueryResult<T>> parallelQuery(List<ParallelQuery> parallelQueryList, Map<String, Object> paramsMap);
```

`ParallelQuery` is built fluently: `create()`, `sql()`, `page()`, `topSize()`, `randomSize()`, `names()`, `values()`, `paramsMap()`, `resultType()`, `dataSource()`, `showSql()`, `contextData()`.

3. Usage Example

```
List<ParallelQuery> queries = new ArrayList<>();
// plain query
queries.add(ParallelQuery.create().sql("sys_staff_find").resultType(StaffInfoVO.class));
// paginated query (setting page makes it a paginated query)
queries.add(ParallelQuery.create().sql("sys_order_find").page(new Page(10, 1)));
// top N query
queries.add(ParallelQuery.create().sql("sys_sale_top").topSize(10));

// shared parameter conditions; sqltoy automatically extracts the parameters each sql needs
Map<String, Object> paramsMap = MapKit.keys("status", "organId").values(1, "T001");
List<QueryResult> results = lightDao.parallelQuery(queries, paramsMap);

List<StaffInfoVO> staff = results.get(0).getRows();
Page orderPage = results.get(1).getPageResult();
List saleTop = results.get(2).getRows();
```

Each query can also **set its own parameters individually** (overriding/adding to the shared parameters):

```
queries.add(ParallelQuery.create().sql("sys_order_find")
        .names("status").values(1)
        .page(new Page(10, 1)));
```

> Common `QueryResult` methods: `getRows()` (result set), `getPageResult()` (pagination result), `getExecuteTime()` (execution time).

> **Renamed since 6.0**: `parallQuery` / `ParallQuery` are renamed to `parallelQuery` / `ParallelQuery`; use the old names in version 5.6.x.

> 🎬 Full runnable example: `ParallelQueryTest.java` in the demo project `sqltoy-showcase` (three forms: basic parallel execution, `ParallelConfig` thread configuration, and mixed pagination).

Stored Procedures

sqltoy supports calling database stored procedures and can retrieve query results, out parameters, and multiple result sets.

1. LightDao API

```
// call without result sets (in parameters are passed in ? order)
StoreResult executeStore(String storeSqlOrKey, Object[] inParamValues);

// with out parameters / a specified result type
// outParamsType can be null; resultType can be a VO, Map.class, LinkedHashMap.class, Array.class, or null (a two-dimensional List)
StoreResult executeStore(String storeSqlOrKey, Object[] inParamValues,
                         Integer[] outParamsType, Class resultType);

// stored procedure with multiple result sets
StoreResult executeMoreResultStore(String storeSqlOrKey, Object[] inParamValues,
                                   Integer[] outParamsType, Class... resultTypes);
```

`storeSqlOrKey` can be a sqlId defined in the xml, or directly `{call storeName(?,?)}`.

2. Fluent Style: store()

```
StoreResult result = lightDao.store()
        .sql("{call storeName(?,?)}")
        .inParams(value1, value2)
        .outTypes(java.sql.Types.INTEGER)   // out parameter types; can be omitted
        .resultType(StaffInfoVO.class)
        .moreResult(false)                  // multiple result sets or not
        .timeout(30)
        .submit();
```

3. The StoreResult

Method	Description
`getRows()`	Main result set (List)
`getMoreResults()`	All result sets (`List[]` , used for multiple result sets)
`getOutResult()`	Array of out parameter values (`Object[]`)
`getUpdateCount()`	Number of affected rows
`getLabelsList()`	Column names of each result set

4. Call Examples

```
// 1) call without results
lightDao.executeStore("{call storeName(?,?)}", new Object[]{value1, value2});

// 2) stored procedure query (pass null as outParamsType when there are no out parameters)
List result = lightDao.executeStore("{call storeName(?,?)}",
        new Object[]{value1, value2}, null, VO.class).getRows();

// 3) multiple result sets
StoreResult more = lightDao.executeMoreResultStore("{call storeName(?,?,?)}",
        new Object[]{value1, value2}, new Integer[]{java.sql.Types.INTEGER},
        VO1.class, VO2.class);
List[] allResults = more.getMoreResults();
Object[] outValues = more.getOutResult();
```

5. Complete Example: Stored Procedure with Multiple Result Sets

When a stored procedure returns **multiple result sets** (e.g. a master table plus a detail table, or several groups of statistics), use `executeMoreResultStore` or the fluent `store().moreResult(true)` , and retrieve all result sets via `getMoreResults()` .

Suppose the stored procedure `proc_order_report` takes `organId` and `beginDate` as inputs, returns two result sets (an order list and order details), and has an out parameter `totalCount` that returns the total record count.

```
// calls {call proc_order_report(?,?,?)}: the first two are in parameters, the last one is an out parameter
StoreResult storeResult = lightDao.executeMoreResultStore(
        "{call proc_order_report(?,?,?)}",
        new Object[]{ "T001", LocalDate.parse("2026-01-01") },  // in parameters (in ? order)
        new Integer[]{ java.sql.Types.INTEGER },                // out parameter type
        OrderInfoVO.class,        // mapping type of the 1st result set
        OrderItemVO.class);       // mapping type of the 2nd result set

// 1) get all result sets: List[], each element corresponds to one result set
List[] allResults = storeResult.getMoreResults();
List<OrderInfoVO> orders = allResults[0];        // first result set (orders)
List<OrderItemVO> orderItems = allResults[1];    // second result set (details)

// 2) get out parameter values
Object[] outValues = storeResult.getOutResult();
Integer totalCount = (Integer) outValues[0];

// 3) getRows() returns the main result set (i.e. the first result set)
List<OrderInfoVO> mainRows = storeResult.getRows();

System.out.println("Order count=" + orders.size()
        + ", detail count=" + orderItems.size()
        + ", total records=" + totalCount);
```

The equivalent fluent style:

```
StoreResult result = lightDao.store()
        .sql("{call proc_order_report(?,?,?)}")
        .inParams("T001", LocalDate.parse("2026-01-01"))
        .outTypes(java.sql.Types.INTEGER)
        .moreResult(true)
        .resultTypes(OrderInfoVO.class, OrderItemVO.class)
        .submit();

List[] all = result.getMoreResults();              // all[0]=orders, all[1]=details
Integer total = (Integer) result.getOutResult()[0]; // out parameter
```

> Note: `resultTypes` specifies the mapping type of each result set in the order the result sets appear (can be a VO, `Map.class`, `LinkedHashMap.class`, `Array.class`, or `null` for a two-dimensional List); `getMoreResults()` returns `List[]` (all result sets), `getRows()` returns the main (first) result set, and `getOutResult()` returns the array of out parameter values.

SQL Showcase

1. Shared SQL fragment @include(sqlId)

```
<sql id="showcase_1">
    <value><![CDATA[
        select * from table1 t where t.name like :name
    ]]></value>
</sql>

<sql id="showcase_2">
    <value><![CDATA[
        @include("showcase_1")
        #[and t.status=:status]
    ]]></value>
</sql>
```

2. Dynamic fragment @include(:scriptParam)

- A variant of @include(sqlId): the SQL fragment is passed in dynamically as a parameter.

```
String sql="select * from sqltoy_fruit_order where status=:status @include(:sqlScript)";
List result = lightDao.find(sql,MapKit.keys("status", "saleCount", "sqlScript").values(1, 12, "and sale_count>:saleCount"));
```

3. Usage of @if(), @elseif(), @else

- sqltoy reserves a super-powerful usage for SQL handling in special business scenarios:

```
@Test
public void testMultiInnerIfElse1() throws Exception {
    String sql = """
        select * from table where 1=1
        #[@if(:flag==1) and name like :name]
        #[@elseif(:flag==2)
            #[@if(:operateType==1) and status=:status]
            #[@elseif(:operateType==2) and saleType is not :saleType]
            #[@else and saleType is :saleType]
        ]
        #[@else and orderType=:orderType]
        #[@if(:tenantId==4) and tenant=1]
        #[@elseif(:tenantId==3) and tenant=3]
        """;

    SqlToyResult result = SqlConfigParseUtils.processSql(SqlUtil.clearMark(sql),
        new String[]{"flag", "status", "name", "orderType", "saleType", "operateType", "tenantId"},
        new Object[]{2, 1, "Chen", "SALE", null, 4, 3});
    System.err.println(JSON.toJSONString(result));
}
```

- The logical expressions of @if include:
- Single condition: @if(:paramName>=value)
- Multiple conditions: @if(:paramName1>=value1 && :paramName2<=value2)
- Comparison operators: >, >=, ==, <, <=, !=, <>; logical operators: && and ||, include, in, out
- Time comparison: @if(:paramName>=now()+x) or @if(:paramName>=now()-x)
- Time expressions: now(), nowtime(), ${.now}
- Date expressions: day(), sysdate(), ${.day}
- include containment check: @if(:statusAry include 1) — if statusAry is an array containing 1, it returns true; if statusAry is a string, it returns the result of statusAry.contains("1").
- exclude: @if(:statusAry exclude 1), the opposite of include, meaning "does not contain"
- Get the size: @if(size(:statusAry)>1), i.e. get the length of a collection/array via size(:paramName)
- in usage: @if(:status in '1,2,3') returns true when status equals 1, 2, or 3
- out usage: @if(:status out '1,2,3') returns true when status is none of 1, 2, or 3

4. Usage of @loop(), @secure-loop(), @loop-full(), @secure-loop-full()

- @loop() is used to dynamically loop-concatenate fields and conditions in SQL, generally for arrays or collections, and does not apply to in-clause scenarios.
- @secure-loop(): unlike @loop, parameters are not concatenated directly into the SQL but passed as ? placeholders, preventing SQL injection. Three formats:

```
@secure-loop(:loopParam,loopContent)
@loop(:loopParam,loopContent) not recommended
@secure-loop(:loopParam,loopContent,linkSign)
@loop(:loopParam,loopContent,linkSign) not recommended
@secure-loop(:loopParam,loopContent,linkSign,startIndex,endIndex)
@loop(:loopParam,loopContent,linkSign,startIndex,endIndex) not recommended
```

- Example

```
<sql id="qstart_loop_sql">
    <value><![CDATA[
        select ORDER_ID
        @loop(:fields,:fields[i],",")
        from sqltoy_device_order t
        where #[t.ORDER_ID=:orderId]
        -- @blank(:param) exists to work with sqltoy's #[] null-elimination feature: when the value is not null, @blank() is replaced with an empty
string (you can also use @if instead of @blank)
        #[@if(size(:staffIds)>0) and (@secure-loop(:staffIds," t.STAFF_ID=:staffIds[i] "," or ",1,100))]
        #[@blank(:startDates)
            and ( @loop(:startDates,
                " t.TRANS_DATE between STR_TO_DATE(':startDates[i]','%Y-%m-%d')
                and STR_TO_DATE(':endDates[i]','%Y-%m-%d') ",
            " or "))]
    ]]></value>
</sql>
```

5. Configuration-driven integration in sqltoy, e.g. reports and configurable API services

- sqltoy development is not purely about writing Java code; it also serves the rapid development of page-configured reports and API services. The integration mechanism:
- Use an XML fragment directly in code

```
//sqlXml content can come from database configuration, or from a report configuration model fragment
String sqlXml="""
               <sql>
                 content omitted
               </sql>
              """;
//add parameters: new XMLBinding(sqlXml).id(id).lastUpdateTime(lastUpdateTime)
//lastUpdateTime is the last modified time of the xml; if nothing changed, the parsed model is fetched directly from cache
lightDao.findByQuery(new QueryExecutor(new XMLBinding(sqlXml).id(id).lastUpdateTime(lastUpdateTime))
                    .values(valueDTO).resultType(OrderInfoVO.class));
```

- Parse the XML fragment and register it into the sqltoy context

```
String sqlXML=sqlElt.element("sql").getTextTrim();
//parse the xml with sqltoy's internal utility to build the sqltoy model
SqlToyConfig sqlToyConfig=lightDao.getSqlToyContext().parseSqlSegment(sqlXML);
//a report may contain multiple sql fragments, so ensure uniqueness by combining reportId+"_"+index
sqlToyConfig.setId(reportId+"_"+sqlScriptIndex);
//register the model into the sqltoy context; afterwards you can call lightDao.find(id,paramsMap)
lightDao.getSqlToyContext().putSqlToyConfig(sqlToyConfig);
```

> For the complete closed-loop example of "store configuration in DB → dynamic parse → report / API service" (including the `XMLBinding` cache mechanism and `@joinFields` dynamic columns), see Configurable Service Foundation (Reports / Configurable API Services).

6. Collecting slow-query SQL statistics

- Enable it with the default configuration

```
# Print the sql execution timeout threshold (milliseconds, default is 8000 ms)
#spring.sqltoy.printSqlTimeoutMillis=8000
# Set the default overtime handler, which collects timed-out sql entries
spring.sqltoy.overTimeSqlHandler= org.sagacity.sqltoy.plugins.overtime.DefaultOverTimeHandler
```

- Get slow queries via the API

```
//size     number of records to fetch
//hasSqlId whether it is an sql defined in xml with an id (the other kind is sql written directly in code)
//public List<OverTimeSql> getSlowestSql(int size, boolean hasSqlId);

List<OverTimeSql> slowSqlList=lightDao.getSqlToyContext().getSlowestSql(10,true);
```

7. The database is not in the adaptation list, but it is derived from postgresql/opengaussdb — how to adapt it?

- 7.1 For a single database, specify the dialect directly

```
#solon.sqltoy.dialect=postgresql
spring.sqltoy.dialect=postgresql
```

- 7.2 For multi-database scenarios, map dialects via the database name obtained from conn.getMetaData().getDatabaseProductName()

```
#solon.sqltoy.dialectMap.oscar=opengauss
spring.sqltoy.dialectMap.oscar=opengauss
```

Cache Translate

Cache Translate Introduction

Cache translate exists to simplify SQL logic and avoid excessive joins. Example: when querying the staff table, an employee's post, home town and skill level are data-dictionary codes that must be shown as names. Multi-value translation is supported: a field containing `A,B,C` is translated into `Name A,Name B,Name C`.

Main use cases

1. Reduce join queries — simpler SQL and a significant efficiency gain
2. Fetch caches directly through the API to feed business logic or the UI, avoiding a database hit every time e.g. front-end selects, checkboxes, suggest components and organization trees can read straight from the cache
3. Filter cache data through the API and reverse-match keys, replacing SQL LIKE queries

Three ways to use cache translate

1. The `<translate>` tag in the sql xml
2. `QueryExecutor::translates(Translate... translates)` in findByQuery (and EntityQuery equivalents)
3. The `@Translate` annotation on DTO properties

Create the dictionary and organization tables

SQL sample script: translate_template.sql

Configure cache translate

Enable cache translate in sqltoy

```
spring:
    sqltoy:
        # defaults to classpath:sqltoy-translate.xml;classpath:translates — no config needed with default naming
        translateConfig: classpath:sqltoy-translate.xml
```

The sqltoy-translate.xml file

> For the schema location of `sqltoy-translate.xml` and IDE completion setup, see the XML Schema Guide.

```
<?xml version="1.0" encoding="UTF-8"?>
<sagacity
	xmlns="https://www.sagframe.com/schema/sqltoy-translate"
	xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="https://www.sagframe.com/schema/sqltoy-translate https://sagframe.github.io/schema/sqltoy-translate.xsd">
	<!-- caches expire by default after 1 hour, so only frequently-changing caches need timely checkers -->
	<cache-translates>
		<!-- cache loaded directly by a sql query -->
		<sql-translate cache="dictKeyNameCache">
			<sql>
			<![CDATA[
				select t.DICT_KEY,t.DICT_NAME,t.STATUS
				from SQLTOY_DICT_DETAIL t
			 where t.DICT_TYPE=:dictType
			 order by t.SHOW_INDEX
			]]>
			</sql>
		</sql-translate>
		<!-- organ id to organ name cache -->
		<sql-translate cache="organIdNameCache">
			<sql>
			<![CDATA[
				select ORGAN_ID,ORGAN_NAME from SQLTOY_ORGAN_INFO order by SHOW_INDEX
			]]>
			</sql>
		</sql-translate>
	</cache-translates>

	<!-- cache refresh checkers; multiple sql/service/rest based checkers are allowed -->
	<cache-update-checkers cluster-time-deviation="1">
		<!-- sql-based cache update checker -->
		<sql-increment-checker cache="organIdNameCache"    check-frequency="60">
			<sql><![CDATA[
			--#not_debug#--
			select ORGAN_ID,ORGAN_NAME
			from SQLTOY_ORGAN_INFO
			where UPDATE_TIME >=:lastUpdateTime
			]]></sql>
		</sql-increment-checker>

		<!-- incremental update where the first column of the result is the inside category -->
		<sql-increment-checker cache="dictKeyNameCache"    check-frequency="15" has-inside-group="true">
			<sql><![CDATA[
			--#not_debug#--
			select t.DICT_TYPE,t.DICT_KEY,t.DICT_NAME,t.STATUS
			from SQLTOY_DICT_DETAIL t
		 where t.UPDATE_TIME >=:lastUpdateTime
			]]></sql>
		</sql-increment-checker>
	</cache-update-checkers>
</sagacity>
```

Overview of cache definition and update-checker types

`sqltoy-translate.xml` supports **4 cache definition types** (under `<cache-translates>`) and **6 update checkers** (under `<cache-update-checkers>`), freely combinable by data source:

Type	Use case	Key attributes
`<sql-translate>`	The most common: one sql query loads the full key/name cache	`cache`, `datasource`, `sql` (attribute or child element)
`<service-translate>`	Cache data must be post-processed by a service (multi-source aggregation, decryption before caching, etc.)	`service` (spring bean name), `method`
`<rest-translate>`	Cache data comes from a REST service (with basic/jwt auth, see next section)	`url`, `username`, `password`, `auth-type`, etc.
`<local-translate>`	Local static/dynamic data: a properties file, or combined with the FIFO dynamic cache	`properties`, `sid`, `dynamic-cache` series (see FIFO cache)

All 4 cache definitions share a set of capacity/lifecycle attributes: `cache` (required), `keep-alive` (seconds alive, default 3600), `heap` (max in-heap entries, default 10000), `off-heap` (off-heap MB, default 0), `disk-size` (disk MB, default 0), `i18n` (e.g. `zh_cn:1,en_us:2`), `desc`.

Method-signature contract of `service-translate`: `service` is first resolved as a spring bean name; if not found and the name contains `.`, it is resolved as a fully-qualified class name. The method takes a **single String argument** (the cache category cacheType; empty when no cache-type is

configured) and returns `List<List>` / `List<Object[]>` (or `Map<String,Object[]>`):

```
<!-- Example: dictService.getDictByType(String cacheType) -->
<service-translate cache="dictKeyNameCache" service="dictService" method="getDictByType" keep-alive="30" />
```

Update checkers come in two flavors — **full check** (fetches the complete data and compares) and **incremental check** (queries only changes after `lastUpdateTime`; preferred) — with `check-frequency` being the check interval in seconds:

Checker	Description
`<sql-increment-checker>`	The most common: incremental sql; `has-inside-group="true"` means the first result column is the inside category
`<sql-checker>`	Full check: fetches the complete data for comparison each time
`<service-increment-checker>` / `<service-checker>`	Update detection via a service method (same signature contract as service-translate)
`<rest-checker>` / `<rest-increment-checker>`	Update detection via a REST service (same auth as rest-translate)

> All 6 checkers accept the container attribute `cluster-time-deviation` (default 1 second) to tolerate clock drift across a cluster.

REST authentication (basic / jwt)

REST-mode cache fetching (`rest-translate`) and REST-form update checking (`rest-checker` / `rest-increment-checker`) support **two authentication modes — basic (default) and jwt** (**jwt since 6.0.4**); under jwt the token can be obtained in one of three ways, mutually exclusive:

```
<!-- basic mode (default, backward compatible): username/password basic auth -->
<rest-translate cache="restCache" url="https://xxx/cache/list"
    auth-type="basic" username="admin" password="xxx" />

<!-- jwt mode 1: fetch the token by login (username/password submitted as a form,
     the token is extracted from the json response via token-path and cached) -->
<rest-translate cache="restCache" url="https://xxx/cache/list"
    auth-type="jwt" username="admin" password="xxx"
    token-url="https://xxx/login" token-path="data.token" />

<!-- jwt mode 2: self-signed locally (no login endpoint; the server verifies with the same secret or the public key) -->
<rest-translate cache="restCache" url="https://xxx/cache/list"
    auth-type="jwt" username="admin" token-secure-key="xxxx"
    sign-algorithm="HS256" />

<!-- jwt mode 3: static long-lived token configured directly -->
<rest-translate cache="restCache" url="https://xxx/cache/list"
    auth-type="jwt" authorization="xxxxx" />
```

The three ways of obtaining the jwt token — **token-url / token-secure-key / authorization — are mutually exclusive** (when several are configured, priority follows this order):

Attribute	Description	Default
`auth-type`	Authentication mode: `basic` (default) / `jwt`	basic
`token-url`	Login url to fetch the token (username/password submitted as a form)	-
`token-path`	Token field path in the login response json, dot path supported (e.g. `data.token`)	access_token
`token-prefix`	Header value prefix; set to `none` to send the bare token	Bearer
`token-header`	Header name carrying the token	Authorization
`token-expire`	Token validity in seconds; `expires_in` in the login response takes precedence	1800
`token-secure-key`	Self-signing key (HS family: shared secret; RS family: PKCS#8 private key)	-
`sign-algorithm`	Self-signing algorithm: HS256 / HS384 / HS512 / RS256 / RS384 / RS512	HS256
`authorization`	Static token configured directly (long-lived token scenario)	-

Runtime behavior: the token is cached and refreshed automatically when nearly expired; on a 401 the token is invalidated and re-fetched (or re-signed) with one retry. The self-signing mode generates the jwt locally (claims include sub=username, iat, exp); HS keys are shared secrets, RS keys are PKCS#8 PEM private keys (convertible via `openssl pkcs8 -topk8 -nocrypt`).

> [!TIP] The caches above are all **full-load** mode. If your data reaches millions of rows (e.g. marketplace SKUs, phone-number regions), use the Large-Scale Master-Data Cache (FIFO): the cache is not fully loaded; missing keys are fetched in batches on demand, and the local cache keeps only the hottest data.

Test it

Add 2 properties to OrderInfoVO

```
@Data
@Accessors(chain = true)
public class OrderInfoVO implements Serializable {
	/**
	 *
	 */
	private static final long serialVersionUID = 3853685083880883292L;
/*---begin-auto-generate-don't-update-this-area--*/
// existing properties omitted here
/*---end-auto-generate-don't-update-this-area--*/

	// two new extended properties
	/**
	 * Organ name
	 */
	private String organName;

	/**
	 * Order type name
	 */
	private String orderTypeName;
}
```

Write the test class

1. Java-code mode

```
package com.sqltoy.helloworld.service;

import java.time.LocalDateTime;
import java.util.List;

import org.junit.jupiter.api.Test;
import org.sagacity.sqltoy.config.model.Translate;
import org.sagacity.sqltoy.dao.LightDao;
import org.sagacity.sqltoy.model.MapKit;
import org.sagacity.sqltoy.model.QueryExecutor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import com.sqltoy.helloworld.dto.OrderInfoVO;

@SpringBootTest
public class TranslateTest {
	@Autowired
	LightDao lightDao;

	// use translates(Translate...Translates) on the QueryExecutor/EntityQuery of
	// findByQuery / findPageByQuery / findEntity / findPageEntity
	@Test
	public void testTranslate() {
		String sql = """
				select t.*,
					-- two extra query columns mapped to the name properties of the VO
					t.order_type orderTypeName,
					t.organ_id organName
				from SQLTOY_ORDER_INFO t
				where 1=1
				#[and t.status in (:statusAry)]
				#[and t.create_time>=:beginTime]
				#[and t.create_time<=:endTime]
				""";
		List<OrderInfoVO> result = lightDao.findByQuery(new QueryExecutor(sql)
				.translates(new Translate("dictKeyNameCache").setCacheType("ORDER_TYPE").setColumn("orderTypeName"),
						new Translate("organIdNameCache").setColumn("organName"))
				.values(MapKit.keys("statusAry", "beginTime", "endTime").values(new Integer[] { 1 },
						LocalDateTime.parse("2024-10-17T00:00:01"), null))
				.resultType(OrderInfoVO.class)).getRows();
		// print the translate result: orderTypeName / organName are filled by cache translate
		result.forEach(row -> System.err.println(row.getOrderTypeName() + " | " + row.getOrganName()));
	}
	// the xml equivalent:
	//<translate cache="dictKeyNameCache" cache-type="ORDER_TYPE" columns="orderTypeName"/>
	//<translate cache="organIdNameCache" columns="organName"/>
	// sql see com/sqltoy/helloworld/sqltoy/sqltoy-helloworld.sql.xml; sqlId must be globally unique
	@Test
	public void testTranslateWithSqlId() {
		List<OrderInfoVO> result = lightDao.find("helloworld_search_orderInfo",
				MapKit.keys("statusAry", "beginTime", "endTime").values(new Integer[] { 1 },
						LocalDateTime.parse("2024-10-17T00:00:01"), null),
				OrderInfoVO.class);
		// print the translate result: orderTypeName / organName are filled by cache translate
		result.forEach(row -> System.err.println(row.getOrderTypeName() + " | " + row.getOrganName()));
	}

	/**
	 * Annotation mode: put @Translate on the organName / orderTypeName properties of OrderInfoVO, e.g.:
	 * @Translate(cacheName = "organIdNameCache", keyField = "organId")
	 * private String organName;
	 */
	@Test
	public void testTranslateByAnnotationTrans() {
		List<OrderInfoVO> result = lightDao.findEntity(OrderInfo.class,
				// for select * there is no need to call .select(); shown here for demonstration only
				// EntityQuery.create().unselect(excludedFields)
				EntityQuery.create().select()
						// follows the sqltoy dynamic-condition rules
						.where("#[and status in (:statusAry)]#[and createTime>=:beginTime]#[and createTime<=:endTime]")
						.values(MapKit.keys("statusAry", "beginTime", "endTime").values(new Integer[] { 1 },
								LocalDateTime.parse("2024-10-17T00:00:01"), null)),
				OrderInfoVO.class);
		// print the translate result: orderTypeName / organName are filled by cache translate
		result.forEach(row -> System.err.println(row.getOrderTypeName() + " | " + row.getOrganName()));
	}
```

```
}
```

2. xml mode

```
<?xml version="1.0" encoding="UTF-8"?>
<sagacity
	xmlns="https://www.sagframe.com/schema/sqltoy-translate"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="https://www.sagframe.com/schema/sqltoy-translate https://sagframe.github.io/schema/sqltoy-translate.xsd">
	<!-- caches expire by default after 1 hour, so only frequently-changing caches need timely checkers -->
	<cache-translates>
		<!-- cache loaded directly by a sql query -->
		<sql-translate cache="dictKeyNameCache">
			<sql>
			<![CDATA[
				select t.DICT_KEY,t.DICT_NAME,t.STATUS
				from SQLTOY_DICT_DETAIL t
			 where t.DICT_TYPE=:dictType
			 order by t.SHOW_INDEX
			]]>
			</sql>
		</sql-translate>
		<!-- organ id to organ name cache -->
		<sql-translate cache="organIdNameCache">
			<sql>
			<![CDATA[
				select ORGAN_ID,ORGAN_NAME from SQLTOY_ORGAN_INFO order by SHOW_INDEX
			]]>
			</sql>
		</sql-translate>
	</cache-translates>

	<!-- cache refresh checkers; multiple sql/service/rest based checkers are allowed -->
	<cache-update-checkers cluster-time-deviation="1">
		<!-- sql-based cache update checker -->
		<sql-increment-checker cache="organIdNameCache"    check-frequency="60">
			<sql><![CDATA[
			--#not_debug#--
			select ORGAN_ID,ORGAN_NAME
			from SQLTOY_ORGAN_INFO
			where UPDATE_TIME >=:lastUpdateTime
			]]></sql>
		</sql-increment-checker>

		<!-- incremental update where the first column of the result is the inside category -->
		<sql-increment-checker cache="dictKeyNameCache"    check-frequency="15" has-inside-group="true">
			<sql><![CDATA[
			--#not_debug#--
			select t.DICT_TYPE,t.DICT_KEY,t.DICT_NAME,t.STATUS
			from SQLTOY_DICT_DETAIL t
		 where t.UPDATE_TIME >=:lastUpdateTime
			]]></sql>
		</sql-increment-checker>
	</cache-update-checkers>
</sagacity>
```

Other uses of the cache

- 1. Fetch a cache directly:

```
lightDao.getTranslateCache("dictKeyNameCache","ORDER_TYPE");
lightDao.getTranslateCache("organIdNameCache",null,OrganInfoVO.class);
```

```
/**
 * Get a translate cache, handy for front-end select options, checkboxes, suggest components, etc.
 * @param cacheName
 * @param cacheType for data dictionaries pass the dictionary type, otherwise null
 */
public HashMap<String, Object[]> getTranslateCache(String cacheName, String cacheType);

/**
 * Fetch the cache data as objects
 * @param cacheName
 * @param cacheType  for data dictionaries pass the dictionary type, otherwise null
 * @param reusltType the sql/properties aliases defined in the cache must match the properties of resultType
 */
public <T> List<T> getTranslateCache(String cacheName, String cacheType, Class<T> reusltType);
```

- 2. Translate a collection through the cache

```
/**
 * Translate a specific property of each row via a callback
 * @param dataSet        the data collection
 * @param cacheName      cache name
 * @param cacheType      for categorized caches (data dictionary) pass the category; for staff/organ caches pass null
 * @param cacheNameIndex defaults to 1 — the column holding the name in the cache array (name, alias, short name, full name...)
 * @param handler
 */
public void translate(Collection dataSet, String cacheName, String cacheType, Integer cacheNameIndex,
              TranslateHandler handler);

// usage example
lightDao.translate(staffVOs, "staffIdName", null, 1, new TranslateHandler() {
       // provide the key
       public Object getKey(Object row) {
              return ((StaffInfoVO)row).getStaffId();
       }
       // set the translated name onto the target property
       public void setName(Object row, String name) {
              ((StaffInfoVO)row).setStaffName(name);
       }
});
```

- 3. Filter through the cache

```
/**
 * Fuzzy-match names against a cache to obtain the matching keys
 * @param cacheMatchFilter
 * @param matchRegexes     array
 */
public String[] cacheMatchKeys(CacheMatchFilter cacheMatchFilter, String... matchRegexes);

// example
String[] keys = lightDao.cacheMatchKeys(CacheMatchFilter.create()
              .cacheName("organIdNameCache")
              // match on the first column (organ name); matchIndexs(1,2) would also match aliases
              .matchIndexs(1)
              // return column 0
              .cacheKeyIndex(0)
              // prefer exact matching (e.g. given "Shanghai New Energy" against "Shanghai New Energy..." and "China Shanghai New Energy Development
Co.",
              // the first organ's code is returned; otherwise a LIKE-style match applies)
              .priorMatchEqual(true)
              // max number of matches
              .matchSize(2), "New Energy Research Institute");
```

Advanced cache-translate scenarios

Multiple translates per field / conditional translate

The same field can have several `<translate>` tags against different caches; the `where` attribute enables **conditional translation** (translate only when the condition holds), supporting four logics: `==`, `!=`, `in`, `out`:

```
<!-- translate only when the order type is PO -->
<translate cache="orderTypeCache" columns="orderTypeName" where="order_type==PO" />
<!-- neq / in / out work the same way -->
<!-- <translate ... where="status!=0" /> -->
<!-- <translate ... where="order_type in (PO,SPO)" /> -->
<!-- <translate ... where="order_type out (SALES)" /> -->
```

Split-value translation (split-sign / link-sign)

When a field holds **concatenated codes** such as `F,M`, `split-sign` splits them for individual translation and `link-sign` joins the results back:

```
<!-- "F,M" → translated to "Female,Male" -->
<translate cache="sexCache" columns="sexName" split-sign="," link-sign="、" />
```

Tenant-isolated translation

In multi-tenant scenarios, `cache-type` can dynamically carry the current user's tenant id (e.g. `${user_tenant_id}`) through the common-field extension, giving a **tenant-isolated cache** (reusing the data-dictionary categorization mechanism):

```
<translate cache="tenantConfigCache" cache-type="${user_tenant_id}" columns="configName" />
```

i18n translation

Add an `i18n` attribute to the cache definition in `sqltoy-translate.xml` (format `locale:columnIndex`, comma- or semicolon-separated, e.g. `zh_cn:1,en_us:2`) so translation picks the value from the column matching the current locale:

```
<!-- the cache sql must also select the per-language name columns; i18n declares the locale-to-column mapping -->
<sql-translate cache="dictCache" datasource="dataSource" i18n="zh_cn:1,en_us:2">
    <sql><![CDATA[
        select t.DICT_KEY, t.DICT_NAME, t.DICT_NAME_EN
        from SQLTOY_DICT_DETAIL t
        where t.DICT_TYPE=:dictType
    ]]></sql>
</sql-translate>
```

In code, set the current thread locale via `I18nThreadHolder` (in real projects a filter usually injects the user's locale, calling `remove()` at the end of the request):

```
I18nThreadHolder.put("en_us");
try {
    // cache translation for this query takes the en_us column (DICT_NAME_EN)
    List result = lightDao.find("sqltoy_dict_find", paramsMap, DictInfoVO.class);
} finally {
    I18nThreadHolder.remove();
}
```

Large-Scale Cache (FIFO)

Cache Translate typically **loads the full cache in one go** after startup, which is very efficient for small-to-medium data such as dictionaries and organizations. However, in scenarios like platform e-commerce SKUs, administrative divisions plus house numbers, or phone-number location lookup, the data volume can reach **the millions or even tens of millions**, while the actually high-frequency entries are often only tens of thousands to a little over a hundred thousand — full loading both slows down startup and consumes a large amount of memory.

For such scenarios, sqltoy provides a **dynamic cache** mechanism: locally it keeps only a bounded `FIFOMap` (e.g., at most 100,000 entries); during translation, keys missing from the cache are **loaded on demand in batches** through the `DynamicCacheFetch` interface. Frequently used data is automatically retained, and rarely used data is automatically evicted.

> [!NOTE] The batch fetch interface `getCache(..., String[] keys)` and the default cache manager `FIFODynamicFetchCacheManager` are provided since version **5.6.67** (see the 5.6.67 release notes for details).

1. How It Works

```
flowchart TD
    A["Query result set with keys"] --> B{"Key exists in local FIFOMap?"}
    B -- hit --> C["Return translation directly"]
    B -- miss --> D["Collect & dedupe missing keys"]
    D --> E["One batch call to DynamicCacheFetch.getCache"]
    E --> F[("Massive data source (Redis / DB)")]
    F --> G["Backfill FIFOMap, finish batch translation"]
    G --> H["Evict coldest key when over maxSize"]
```

Key design points:

1. **Locally bounded cache**: `FIFOMap` works in `accessOrder` mode; every `get` moves the entry to the tail of the queue (effectively extending its life), and when capacity exceeds `maxSize`, the least-recently-used entry at the head is evicted — a natural match with the real-world pattern that "frequently used data concentrates";
2. **Batch fetching**: all keys missed in one query result set are **deduplicated and passed to the interface in a single call**, so the whole batch requires only one IO round trip, avoiding the performance disaster of per-row queries;
3. **Multi-level storage**: implementations of `DynamicCacheFetch` usually query a distributed cache such as Redis first and then fall back to the database, forming a multi-level structure of "local FIFOMap → Redis → DB".

2. Usage Steps (Three Steps)

Step 1: Define the dynamic cache (sqltoy-translate.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<sagacity xmlns="https://www.sagframe.com/schema/sqltoy-translate"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://www.sagframe.com/schema/sqltoy-translate
    https://sagframe.github.io/schema/sqltoy-translate.xsd">
    <cache-translates>
        <!-- dynamic cache: no full loading; missed keys are fetched on demand in batches via DynamicCacheFetch -->
        <local-translate cache="skuDict" sid="SKU_CACHE" properties="id,name,brand"
            keep-alive="3600" dynamic-cache="true" dynamic-cache-maxSize="100000" />
    </cache-translates>
</sagacity>
```

Description of the `local-translate` attributes:

Attribute	Description	Default
`cache`	Cache name, referenced via `cache="xxx"` at the usage site	Required
`dynamic-cache`	Enables the dynamic fetch mode	false
`dynamic-cache-initSize`	Initial capacity of the FIFOMap	10000
`dynamic-cache-maxSize`	Maximum capacity of the FIFOMap; the least-recently-used data is evicted beyond it	100000
`dynamic-cache-loadFactor`	Map load factor	0.75
`keep-alive`	Data time-to-live in seconds; **a negative value means valid forever**	default-keep-alive (3600)
`sid`	Identifier, passed as-is to the fetch interface (e.g., sqlId), helping a generic implementation distinguish data sources	none
`properties`	Property info, passed as-is to the fetch interface (e.g., specifying which columns to return)	none

Step 2: Implement the DynamicCacheFetch fetch interface

```
package org.sagacity.sqltoy.translate;

public interface DynamicCacheFetch {

    /**
     * Initialization method; you can get beans from the container (e.g., redisTemplate)
     * via appContext.getBean(...)
     */
    public void initialize(AppContext appContext);

    /**
     * Gets the cached data of a single key (called in non-batch scenarios such as
     * single-object translation)
     * @return Object[]{key, name1, name2, ...}; returns null if not found
     */
    public Object[] getCache(String cacheName, String cacheType, String sid,
            String[] properties, String key);

    /**
     * Batch fetch (implemented in version 5.6.67, using a batch-query mode to reduce
     * IO round trips)
     * @param keys keys missed in the result set, already deduplicated
     * @return Map<key, Object[]> in the structure {key,[id,name,email]}
     */
    public Map<String, Object[]> getCache(String cacheName, String cacheType, String sid,
            String[] properties, String[] keys);
}
```

Parameter description:

Parameter	Description
`cacheName`	Cache name, i.e., the `cache` of `local-translate`
`cacheType`	The group type of a dictionary-like cache (has a value when `cache-type` is passed at the usage site); null in non-grouped scenarios
`sid` / `properties`	The identifier and properties defined in the cache; passed through as-is so generic implementations can branch on them
`key` / `keys`	Cache key(s) to fetch (deduplicated in batch mode)

Complete implementation example (the same one used in the demo project, which demonstrates fetching from a database; in production, large-scale data usually queries Redis first and then falls back to the database):

```
public class BigDictDynamicCacheFetch implements DynamicCacheFetch {

    // initialize is called during SqlToyContext initialization, when beans such as
    // LightDao may not be ready yet;
    // calling getBean directly can easily create a circular dependency, so appContext
    // is recorded and lazy loading is used instead
    private AppContext appContext;
    private volatile LightDao lightDao;

    @Override
    public void initialize(AppContext appContext) {
        this.appContext = appContext;
        // If you need to integrate Redis, get it here: appContext.getBean("redisTemplate")
    }

    // The single-key scenario directly reuses the batch logic, avoiding two sets of fetch code
    @Override
    public Object[] getCache(String cacheName, String cacheType, String sid,
            String[] properties, String key) {
        Map<String, Object[]> datas = getCache(cacheName, cacheType, sid,
                properties, new String[] { key });
        return (datas == null) ? null : datas.get(key);
    }

    // Batch fetch: keys missed in one result set are deduplicated and passed in at once
    @SuppressWarnings({ "unchecked", "rawtypes" })
    @Override
    public Map<String, Object[]> getCache(String cacheName, String cacheType, String sid,
            String[] properties, String[] keys) {
        Map<String, Object[]> result = new HashMap<String, Object[]>();
        if (keys == null || keys.length == 0) {
            return result;
        }
        // resultType=Array.class returns rows as Object[] arrays; column 0 is the key
        List rows = getLightDao().find(
                "select DICT_KEY,DICT_NAME,STATUS from sqltoy_dict_detail where DICT_KEY in (:keys)",
                MapKit.map("keys", Arrays.asList(keys)), Array.class);
        for (Object row : rows) {
            Object[] rowData = (Object[]) row;
            result.put(rowData[0].toString(), rowData);
        }
        return result;
    }

    private LightDao getLightDao() {
        if (lightDao == null) {
            synchronized (this) {
                if (lightDao == null) {
                    lightDao = appContext.getBean(LightDao.class);
                }
            }
        }
        return lightDao;
    }
}
```

Step 3: Register the implementation class

Spring Boot (sqltoy-spring-boot-starter): the value supports a **container bean name** (matched first) or a **fully qualified class name** (instantiated via reflection):

```
spring:
    sqltoy:
        dynamicCacheFetch: com.xxx.cache.BigDictDynamicCacheFetch
```

Traditional Spring (XML-configured) projects: inject the SqlToyContext property directly:

```
<bean id="sqlToyContext" class="org.sagacity.sqltoy.SqlToyContext" init-method="initialize">
    <!-- other properties omitted -->
    <property name="dynamicCacheFetch">
        <bean class="com.xxx.cache.BigDictDynamicCacheFetch" />
    </property>
</bean>
```

3. Runtime Mechanism Details

Batch fetching and two-pass translation: during the first translation pass over the result set, missed keys are only registered in `DynamicCacheHolder` (a deduplicated set) without issuing any query; after the first pass ends, the framework calls the batch interface to fetch them in one go, and after **backfilling the FIFOMap**, it performs a second pass to batch-write the translations into the unmatched rows. Therefore the whole batch of missed keys incurs only one fetch IO.

FIFO eviction policy: `FIFOMap` is implemented on top of `LinkedHashMap(accessOrder=true)`; a `get` hit moves the entry to the tail (extending its life), and a `put` that exceeds `maxSize` evicts the least-recently-used entry at the head. `get/put/remove` are all synchronized, making it safe under concurrent multi-threaded translation.

keep-alive expiration: it takes effect for the whole group of "cache name + cacheType" — timing starts from when the group's data is initialized; after `keep-alive` seconds, a background daemon thread (checking once every 3 minutes) clears the whole group, and it is dynamically reloaded on next use; a negative value means it never expires.

Cooperation with cache-update-checker incremental updates: the dynamic cache can also be configured with a cache update checker, but on update **only keys that already exist locally are overwritten** — keys that were not loaded have no old value locally and naturally get the latest data the next time they are dynamically fetched.

Zero awareness on the usage side: the way translation is referenced is exactly the same as for a normal cache — whether it is `<translate cache="skuDict" ... />` in the sql xml or `@Translate(cache="skuDict")` on a VO — without any extra code.

4. [Optional] Custom DynamicFecthCacheManager

Storage management of the dynamic cache is handled by the `DynamicFecthCacheManager` interface (`Fecth` in the interface name is a historical spelling in the source code). The framework provides a default FIFOMap-based `org.sagacity.sqltoy.translate.cache.impl.FIFODynamicFetchCacheManager`, and in the vast majority of scenarios there is no need to replace it.

```
package org.sagacity.sqltoy.translate.cache;

public interface DynamicFecthCacheManager {
    // Get (or lazily create) the dynamic cache Map for the given cache + group
    public HashMap<String, Object[]> getDynamicCache(TranslateConfigModel cacheModel, String cacheType);
    // Whether a cache has been used (lets the update checker decide whether a check is needed)
    public boolean hasCache(String cacheName);
    // Clear the data of a cache (or of one of its groups)
    public void clear(String cacheName, String cacheType);
    // Initialization (the default implementation starts the keep-alive expiration check thread)
    public void initialize();
    // Destruction (the default implementation stops the check thread)
    public void destroy();
}
```

To replace it with a custom implementation:

```
spring:
    sqltoy:
        # Also supports a bean name or a fully qualified class name (note the source property name is Fecth)
        dynamicFecthCacheManager: com.xxx.cache.MyDynamicFetchCacheManager
```

5. Notes and Cautions

- `dynamic-cache="true"` **must** be accompanied by a registered `DynamicCacheFetch` implementation; if none is registered, the cache degrades to the normal full-loading mode, and since `local-translate` itself has no data source configured, no data can be loaded;
- Column 0 of the `Object[]` returned by the fetch must be the key, and the translation column positions correspond to `cache-indexs` at the usage site (e.g., `Object[]{key, name, status}` corresponds to `cache-indexs="1"`);
- `dynamic-cache-maxSize` should be evaluated against JVM memory: 100,000 entries × about 200 bytes each ≈ on the order of tens of MB; watch the heap limit when multiple large caches are stacked;
- keep-alive expiration clears the whole group, so the first query after a clear triggers a burst of concentrated dynamic fetches; for high-frequency scenarios, it is advisable to increase `keep-alive`, or to combine a negative (permanently valid) setting with the incremental update checker.

6. Complete Runnable Example

> For the complete example, see the demo project `sqltoy-showcase`: `BigDictDynamicCacheFetch.java` (the fetch implementation), `DynamicCacheTest.java` (translation + FIFOMap batch-fetch and eviction verification), the `bigDictCache` definition in `sqltoy-translate.xml`, and

> the `spring.sqltoy.dynamicCacheFetch` registration in `application.yml`.

Related Pages

- Cache Translate usage: basic configuration of Cache Translate and various additional usages
- Dynamic Cache Control: dynamically extend and refresh caches at runtime via `TranslateManager`

Dynamic Cache Control

Since version 5.2.8, sqltoy allows you to flexibly **dynamically extend caches and dynamically extend cache update checkers**, and to update cached data in real time. All of this is done through `TranslateManager` (the cache translate manager).

1. Getting the Cache Manager

```
TranslateManager translateManager = lightDao.getSqlToyContext().getTranslateManager();
```

> **Prerequisite**: to dynamically extend cache translation, a `translate.xml` (i.e., the cache translate configuration file pointed to by `spring.sqltoy.translateConfig`) must be configured; it may contain no concrete cache configuration at all.

2. Common Operations

`TranslateManager` provides comprehensive operation capabilities over caches:

Method	Description
`existCache(String cacheName)`	Whether a cache exists
`getCacheNames()`	Get all cache names
`getCacheData(String cacheName, String cacheType)`	Get the data of a cache (`HashMap<String, Object[]>`)
`putCacheData(String cacheName, String cacheType, HashMap<String,Object[]> cacheValue)`	Write/update cache data
`clear(String cacheName, String cacheType)`	Clear the data of a cache
`getCacheConfig(String cacheName)`	Get the cache configuration model
`putCache(TranslateConfigModel model)`	**Dynamically add a cache**
`removeCache(String cacheName)`	Dynamically remove a cache
`putCacheUpdater(CheckerConfigModel model)`	Dynamically add a cache update checker
`removeCacheUpdater(CheckerConfigModel model)`	Remove a cache update checker
`getAllTranslates()`	Get all cache translate configurations

3. Two Typical Scenarios

1) Only dynamically control cache data (the cache is already defined in translate.xml): directly use `getCacheData` / `putCacheData` / `clear` to read, write, and refresh cache content — for example, proactively refreshing the cached value of a code after a business change.

2) Dynamically extend caches (adding cache definitions at runtime): use `putCache(TranslateConfigModel)` to dynamically register a new cache, and attach an update checker to it with `putCacheUpdater`, extending the cache translation capability on demand at runtime.

> For the basic usage of Cache Translate, see Cache Translate usage; for the FIFO dynamic cache of large-scale master data, see Large-Scale Master-Data Cache (FIFO).

Cross-DB & Dialect

Dialect Adaptation & Functions

1. sqltoy provides automatic database function adaptation, i.e., at runtime the functions in SQL are automatically converted to functions that fit the current database dialect

```
# enable sqltoy's default function auto-adaptation conversion
spring.sqltoy.functionConverts=default
```

- To replace individual function implementations, e.g., for NVL, keeping the class name the same implements your own custom implementation replacing the one in default

```
# use a custom function implementation to replace Nvl
spring.sqltoy.functionConverts=default,com.yourpackage.Nvl
```

- You can also turn off default and fully decide yourself which functions to enable or load custom functions

```
# enable the built-in Nvl, Instr
spring.sqltoy.functionConverts=Nvl,Instr
# enable custom Nvl, Instr
# spring.sqltoy.functionConverts=com.yourpackage.Nvl,com.yourpackage.Instr
```

- functionConverts=default in sqltoy includes the following functions (28 in total since 6.0.5)

```
org.sagacity.sqltoy.plugins.function.impl.SubStr
org.sagacity.sqltoy.plugins.function.impl.Trim
org.sagacity.sqltoy.plugins.function.impl.Instr
org.sagacity.sqltoy.plugins.function.impl.Concat
org.sagacity.sqltoy.plugins.function.impl.ConcatWs
org.sagacity.sqltoy.plugins.function.impl.Nvl
org.sagacity.sqltoy.plugins.function.impl.DateFormat
org.sagacity.sqltoy.plugins.function.impl.Now
org.sagacity.sqltoy.plugins.function.impl.CurrentDate
org.sagacity.sqltoy.plugins.function.impl.Length
org.sagacity.sqltoy.plugins.function.impl.ToChar
org.sagacity.sqltoy.plugins.function.impl.If
org.sagacity.sqltoy.plugins.function.impl.GroupConcat
org.sagacity.sqltoy.plugins.function.impl.ToNumber
org.sagacity.sqltoy.plugins.function.impl.ToDate
org.sagacity.sqltoy.plugins.function.impl.DateDiff
org.sagacity.sqltoy.plugins.function.impl.Decode
org.sagacity.sqltoy.plugins.function.impl.FindInSet
org.sagacity.sqltoy.plugins.function.impl.Uuid
org.sagacity.sqltoy.plugins.function.impl.YearMonthDay
org.sagacity.sqltoy.plugins.function.impl.LastDay
org.sagacity.sqltoy.plugins.function.impl.AddMonths
org.sagacity.sqltoy.plugins.function.impl.DateAdd
org.sagacity.sqltoy.plugins.function.impl.LeftRight
org.sagacity.sqltoy.plugins.function.impl.Pad
org.sagacity.sqltoy.plugins.function.impl.Extract
org.sagacity.sqltoy.plugins.function.impl.DateTrunc
org.sagacity.sqltoy.plugins.function.impl.Mod
```

- Function implementation interface

```
org.sagacity.sqltoy.plugins.function.IFunction
```

- Trim function implementation example

```
public class Trim extends IFunction {
	private static Pattern regex = Pattern.compile("(?i)\\Wtrim\\(");

	/*
	 * (non-Javadoc)
	 *
	 * @see org.sagacity.sqltoy.config.function.IFunction#dialects()
	 */
	@Override
	public String dialects() {
		return ALL;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see org.sagacity.sqltoy.config.function.IFunction#regex()
	 */
	@Override
	public Pattern regex() {
		return regex;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see org.sagacity.sqltoy.config.function.IFunction#wrap(int,
	 * java.lang.String[])
	 */
	@Override
	public String wrap(int dialect, String functionName, boolean hasArgs, String... args) {
		if (args == null || args.length == 0) {
			return super.IGNORE;
		}
		if (dialect == DBType.SQLSERVER) {
			return "rtrim(ltrim(" + args[0] + "))";
		}
		if (dialect == DBType.H2) {
			return "trim(both ' ' from " + args[0] + ")";
		}
		// other databases keep the trim(field) pattern
		return super.IGNORE;
	}
}
```

1.1 Function adaptation improvements (6.0.4)

New source functions / source forms recognized:

- `nvl2(a, b, c)`: an Oracle source function; non-Oracle databases automatically convert it to `case when a is not null then b else c` null-check logic (passed through natively on the Oracle family; argument-count mismatch keeps the loud error);
- `isnull(x)` single-argument null check: MySQL's isnull returns a 0/1 flag; it was previously converted to nvl/coalesce/ifnull (all requiring ≥2 arguments) and failed with an argument-count error on target databases — now handled correctly;
- `sys_timestamp`: the Oracle source form of current time (precision argument supported); passed through natively on the Oracle family, converted to the corresponding current-time function elsewhere (mysql family keeps the fsp precision argument, sqlserver uses `sysdatetime()`);
- `strpos(str, sub)`: native on the PostgreSQL family / ClickHouse and passed through directly (argument order matches instr); converted to the corresponding instr implementation on other databases;
- `substr(s, -n)` negative offset: the two-argument literal form becomes `RIGHT(s, n)` (compatible with db2-style 1-based positions); the three-argument form uses a case guard (`start = length-n+1`, falling back to 1 when the string is shorter — mysql semantics); expression start positions are left as-is for the target database to judge; HANA's negative offset follows pg semantics (silently wrong values) and is normalized the same way;
- `curdate()` / `CURRENT_DATE`: new current-date (midnight) function class — the Oracle family becomes `TRUNC(CURRENT_DATE)` (session timezone preserved), db2 uses the `CURRENT DATE` keyword form, sqlite uses the local clock (its native CURRENT_DATE is a UTC wall clock, inconsistent with business timezones).

Fixes and enhancements to existing conversions:

- `group_concat` / `listagg` / `string_agg` conversion enhancements: supports the `within group (order by …)` trailing clause, embedded `order by` inside arguments (e.g. pg source `string_agg(a, '-' order by b)`), and `DISTINCT` prefix recognition (explicit loud passthrough where ClickHouse's groupArray does not support them); an "order by" text inside a separator literal is no longer misread as an order-by clause;
- `datediff(month, …)` unified month-difference semantics: MONTHS_BETWEEN's Oracle fractional basis (day difference divided by a 1/31-day month) diverges across databases and is no longer registered as a source form — the year/month component difference is used uniformly;

- `to_char` → sqlserver target: carried by FORMAT (2012+), fixing wrong output for uppercase format forms (e.g. `'YYYY-MM-DD HH24:MI:SS'`) caused by .NET token case-sensitivity, plus quoted-format literal protection;
- sqlite targets for now() / CURRENT_DATE / CURRENT_TIMESTAMP: uniformly use the local clock (the native forms are UTC wall clocks, 8 hours off for UTC+8 business logic).

1.2 Function adaptation improvements (6.0.5)

New function classes (11, all registered into `default`):

- `find_in_set(str, strlist)` : native on the MySQL family; elsewhere rewritten as "wrap with commas + exact substring positioning via instr" (so `1` does not match `11`);
- `uuid()` unique-id generation: unifies source spellings `uuid` / `guid` / `newid` / `sys_guid` / `random_uuid` / `gen_random_uuid` / `uuid_generate_v4`, converted per target (mysql `uuid()`, oracle family `sys_guid()`, pg family `gen_random_uuid()`, sqlserver `newid()`, etc.);
- `year` / `month` / `day` (plus `hour` / `minute` / `second`, ...) date-time component extraction;
- `last_day` / `eomonth` last day of the month;
- `add_months` month arithmetic (month-end clamping);
- `date_add` / `date_sub` / `dateadd` / `adddate` / `subdate` date-interval arithmetic family;
- `left(str, n)` / `right(str, n)` prefix/suffix extraction;
- `lpad` / `rpad` fixed-length padding;
- `extract` / `date_part` component extraction (ANSI and pg forms);
- `trunc` / `date_trunc` date-truncation family (oracle `trunc(x[, unit])` and pg `date_trunc(unit, x)` dual source forms);
- `mod(a, b)` modulo (real-DB probing across 18 environments shows only sqlserver lacks mod — converted to the `%` operator).

Source-spelling expansion for existing functions:

- `instr` : added `locate` / `posstr` (mysql and db2 user forms);
- `length` : added `character_length` (ANSI) / `octet_length` (pg byte form);
- `now` : added the `current_timestamp` keyword source form (standard SQL);
- `if` : added `iif` (native on sqlserver 2012+/sqlite 3.32+);
- `substr` : added `mid` (mysql synonym);
- `to_date` : added `str_to_date` (mysql user form).

Function adaptation for the new database (HighGo added in 6.0.5):

- HighGo (PG12 kernel + Oracle compatibility layer): no `uuid()` / `gen_random_uuid()` but has `sys_guid()`; `last_day` / `add_months` / `trunc` are native; no `listagg` but `string_agg` is native — carried by the HIGHGO branch of the function adapters.

> All the above additions/expansions are verified by real-database probing across multiple databases; forms that cannot be converted safely (e.g. aggregate ordering unsupported by the target) are kept "loud" — passed through as-is for the target database to error on, avoiding silent wrong values.

1.3 Extending function adaptation yourself (application side, no framework change needed)

The 28 built-in functions cover the common ones **whose syntax differs across databases**. When your project meets a function the framework does not adapt — a database-private function, a domain-specific one, or something a database added recently — **extend it yourself; there is no need to wait for framework support**. Implement `IFunction` and mount it via `functionConverts`; it then runs through exactly the same mechanism as the built-ins.

The contract (`org.sagacity.sqltoy.plugins.function.IFunction`)

Method	Purpose	Return convention
`dialects()`	declares which dialects this function adapts	`ALL` (the empty string) means every dialect; otherwise a comma- or semicolon-separated **exact enumeration** of dialect names
`regex()`	matches the function call in the SQL	a `Pattern` used to locate the function to rewrite
`wrap(dialect, functionName, hasArgs, args...)`	produces the SQL for the target dialect	return the rewritten fragment; return `IGNORE` (null) to **pass through unchanged**
`wrapArgs(functionName, args...)`	protected convenience helper	builds the result for the most common shape: same function name, arguments re-joined

Minimal example: only SQL Server needs rewriting, every other database passes through (the built-in `Mod`)

```
public class Mod extends IFunction {
	// the leading \W prevents matching trim( inside ltrim(, and (?i) makes it case-insensitive
	private static Pattern regex = Pattern.compile("(?i)\\Wmod\\(");

	@Override
	public String dialects() {
		return ALL;
	}

	@Override
	public Pattern regex() {
		return regex;
	}

	@Override
	public String wrap(int dialect, String functionName, boolean hasArgs, String... args) {
		if (args == null || args.length < 2) {
			return super.IGNORE;
		}
		// sqlserver has no mod function (verified against a real server); the % operator carries it.
		// every other database supports mod natively, so pass through
		if (dialect == DBType.SQLSERVER) {
			return "(" + args[0].trim() + " % " + args[1].trim() + ")";
		}
		return super.IGNORE;
	}
}
```

Points that matter

1. `regex()` should start with `\W` and end with `\(`, e.g. `(?i)\Wmod\(`. The leading `\W` prevents substring false positives; `(?i)` makes it case-insensitive.
2. `dialects()` is matched by **splitting on commas/semicolons and comparing each item exactly**, not by substring containment — containment is biased in both directions (declaring `oracle11` would be hit by the current dialect `oracle`, and declaring `oracle` would fail to match the dialect `oracle11`). A custom function must therefore explicitly enumerate the dialect names it adapts.
3. Matching runs against the SQL **with literals masked**, so the same text appearing inside a string constant is never rewritten by mistake.
4. `args` in `wrap` are the already-split argument expressions as written, possibly with surrounding whitespace — `trim()` them before concatenating.
5. **Only the function-call form can be rewritten.** Operator forms (SQL Server's `a % b`, Oracle's `a || b`) are not function calls, so the regex framework cannot capture them and they are out of scope — write the function form in cross-database SQL (`mod(a,b)`, `concat(a,b)`) for the best portability.
6. In most cases `wrap` should return `IGNORE` for databases that already support the function natively, and rewrite only the dialects that genuinely differ. Rewriting indiscriminately breaks native behavior.

Overriding a built-in implementation

`functionConverts` de-duplicates by function name, and the built-ins expanded from `default` are loaded **after** the custom entries, so a custom implementation with the same name overrides the built-in one:

```
# your own Nvl overrides the built-in Nvl; the rest of the built-ins stay active
spring.sqltoy.functionConverts=default,com.yourpackage.Nvl
```

Short class names (`Nvl`, `Instr`) resolve only inside the framework's own impl package — use fully qualified names for custom implementations.

Advanced: trailing-clause hooks

For a few functions the dialect difference is not inside the parentheses but in the clause immediately following the closing paren — e.g. an aggregate's `within group (order by ...)`, or the boolean convention in `find_in_set(x,y) > 0`. Three optional hooks cover this:

- `suffixPattern()`: the pattern for the trailing clause; it must start with `\s*` to absorb the separating whitespace and end with `\(*`. Returning null (the default) means this function does not consume a trailing clause.
- `suffixNeedsParen()`: true (default) consumes the whole clause by balanced parentheses; false consumes exactly what the pattern matched, for bare operator clauses such as `>0` / `=0`.
- `wrapWithSuffix(dialect, functionName, suffix, args...)`: the entry point when the suffix pattern hits; `suffix` is the clause as written (including leading whitespace). Returning `IGNORE` keeps the clause verbatim.

When not to extend a function, and use a dialect sqlId instead

If the difference is not "the same function written differently" but **a structurally different statement** (pagination syntax, recursive CTE form, upsert semantics, window-function support), function rewriting cannot solve it — use the `sqlId_dialect` / `dialect_sqlId` variants or the `<sql dialect="xxx">` attribute instead; see section 2.

2. The sqlId+dialect pattern

- You can write SQL for specific databases; sqltoy picks the SQL that is actually executed based on the database type, in the order: dialect_sqlId->sqlId_dialect->sqlId. For example, if the database is mysql and sqlId: sqltoy_showcase is called, the SQL actually executed is: sqltoy_showcase_mysql

```
<sql id="sqltoy_showcase">
       <value>
       <![CDATA[
       select * from sqltoy_user_log t
       where t.user_id=:userId
       ]]>
       </value>
</sql>
<!-- sqlId_dialect (lowercase) -->
<sql id="sqltoy_showcase_mysql">
       <value>
       <![CDATA[
       select * from sqltoy_user_log t
       where t.user_id=:userId
       ]]>
       </value>
</sql>
```

2.1 The `dialect` attribute of the `<sql>` element (new in 6.0.3)

A single sql can declare — via the `dialect` attribute — that it is **parsed and solidified in a specified dialect form** (previously this attribute was silently ignored):

```
<!-- Declare that this sql is solidified in the kingbase dialect form: at load time function/reserved-word conversion is done once against kingbase -
->
<sql id="sqltoy_showcase_kingbase" dialect="kingbase">
       <value>
       <![CDATA[
       select ifnull(max(t.amount),0) from sqltoy_user_log t where t.user_id=:userId
       ]]>
       </value>
</sql>
```

How it works:

- **At load time**: function/reserved-word conversion is performed once against the declared dialect and the parse label is solidified (e.g. `ifnull` above is solidified into its `coalesce` form for kingbase); `count-sql` uses the same effective dialect as the main sql;
- **At execution time**: when the current database dialect equals the declaration, the "query dialect == parse label" early-exit path is taken and **function replacement is skipped** (zero overhead); otherwise lazy conversion + caching adapts it as before;
- **Priority**: `mql/eql forced to mongo/es` > `element dialect attribute` > `global dialect`; `mql` / `eql` do not support the attribute;
- The attribute value is validated against the dialect list; an unrecognized dialect triggers a warning and falls back to the global dialect;
- Working with `realDialectFirst`: when `<sql id="x_kingbase" dialect="kingbase">` is matched via variant lookup, the kingbase form is already solidified and the labels are equal — even the lazy conversion is skipped at execution time.

2.2 realDialectFirst: prefer the real dialect for variant lookup (new in 6.0.3)

sqltoy determines the dialect from the JDBC connection's `productName`; in some scenarios (OceanBase configured as mysql, proxied databases, etc.) the **configured dialect differs from the real one**. With this switch enabled, sqlId dialect-variant lookup first follows the real dialect detected from the connection (`DBProfile.realDialect`):

```
# default false; when the real-dialect variant does not exist, the original configured-dialect lookup chain is used as fallback;
# behavior is unchanged when the real dialect equals the configured one
spring.sqltoy.realDialectFirst=true
```

- Example: global `dialect=mysql` while the real database is kingbase — when enabled, calling `sqlId` first looks for `sqlId_kingbase`, then falls back to `sqlId_mysql` → `sqlId`;
- For sqls selected via real-dialect variant matching, function/reserved-word replacement also follows the real dialect (sqls that fall back to the base keep the configured-dialect behavior);
- Only the sql form-conversion layer (functions and reserved words) is affected; execution-strategy dispatch such as pagination is not.

3. Virtual-table macro @virtualTable() (new in 6.0.6)

For "select constant-row union real-query" SQL (dictionary dropdown fallback rows, fixed report total rows, etc.), databases differ in their support for "select of constant rows (without FROM)": oracle/db2/hana require a pseudo-table, while mysql/pg/sqlserver etc. natively support FROM-less select (writing `from dual` actually errors there). Since 6.0.6, write the `@virtualTable()` macro where the pseudo-table belongs — sqltoy expands it per dialect at SQL parse time (processed before function/reserved-word conversion and solidified with SqlToyConfig), so one SQL works across all databases. Taking an xml configuration as an example:

```
<sql id="sys_tenant_options">
	<value>
	<![CDATA[
		select '0' tenant_id,'Global' tenant_name @virtualTable()
		union
		select s.tenant_id,s.tenant_name
		from sys_tenant_info s
		where s.status=1
	]]>
	</value>
</sql>
```

The SQL actually executed on different databases:

```
-- oracle / oracle11 / dm (Dameng) / oceanbase: expanded to from dual
select '0' tenant_id,'Global' tenant_name from dual
union
select s.tenant_id,s.tenant_name from sys_tenant_info s where s.status=1

-- db2: expanded to from sysibm.sysdummy1
select '0' tenant_id,'Global' tenant_name from sysibm.sysdummy1
union
select s.tenant_id,s.tenant_name from sys_tenant_info s where s.status=1

-- hana: expanded to from dummy
select '0' tenant_id,'Global' tenant_name from dummy
union
select s.tenant_id,s.tenant_name from sys_tenant_info s where s.status=1

-- mysql / postgresql / sqlserver etc.: macro stripped (FROM-less select is native)
select '0' tenant_id,'Global' tenant_name
union
select s.tenant_id,s.tenant_name from sys_tenant_info s where s.status=1
```

Dialect expansion reference:

Database	Macro expansion
oracle / oracle11 / dm (Dameng) / oceanbase	`from dual`
db2	`from sysibm.sysdummy1`
hana	`from dummy`
mysql family (mysql/tidb/doris/starrocks), pg family (postgresql/gaussdb/mogdb/openGauss/stardb/vastbase/oscar/kingbase/highgo), sqlserver, sqlite, h2, clickhouse, impala, tdengine	Stripped (FROM-less select is native)

Notes:

- Compatible aliases `@dual()` / `@dummyTable()` — all three names are equivalent, case-insensitive, and tolerant of spaces inside the parentheses;
- May appear multiple times in one SQL, each expanded independently per dialect (one per constant select);
- Works in xml-configured SQL, `@Sql` annotation SQL and code SQL (`findBySql` etc.) alike (unified parse pipeline);
- Expansion happens at parse time and is solidified with SqlToyConfig (same timing as function/reserved-word conversion), zero runtime overhead; SQL without the macro is completely unaffected (plain SQL containing `@` such as `email like 'a@%'` is not touched);
- The pseudo-table mapping shares the same source as the pseudo-tables used by the framework's merge-statement generation (each Dialect's VIRTUAL_TABLE constant);
- With `realDialectFirst`, the macro is expanded and solidified by the "parse dialect" — the runtime lazy-conversion layer does not re-expand it.

4. How to test against multiple database environments at the same time

- sqltoy provides the spring.sqltoy.redoDataSources parameter to set the databases on which queries are repeatedly executed

```
# e.g., while on a mysql setup, test other types of databases at the same time to verify that the SQL adapts to different databases; mainly used for productized software
spring.sqltoy.redoDataSources[0]=pgdb
```

5. FAQ

- 1. Why do my kingbase, polardb, oceanbase and similar databases still produce lots of errors when using the mysql or postgresql database mode? Answer: because sqltoy determines the database type from the productName obtained through the JDBC connection, it cannot always accurately determine the database's dialect mode. Therefore, resolve this with one of the following two approaches

Option 1: In a single-database scenario, set the current database dialect directly

```
spring.sqltoy.dialect=mysql
```

Option 2: In multi-database scenarios, alias incorrectly identified products to the correct database dialect

- spring.sqltoy.dialectMap is of type Map<String,String>; the current dialect can be queried with SELECT version() or a similar statement (ask AI for the exact statement)

```
spring.sqltoy.dialectMap.kingbase=mysql
```

- How the dialectMap mapping works, to help you set correct Map keys: the key is matched via indexOf against the productName

```
// For databases not supported by the framework, match via the dialectMap keys to map to the corresponding dialect
for (Map.Entry<String, String> entry : dialectMap.entrySet()) {
	if (StringUtil.indexOfIgnoreCase(dbDialect, entry.getKey()) != -1) {
		dilectName = entry.getValue().toLowerCase();
		break;
	}
}
```

- 2. After solidifying the global dialect, my dialect-variant sql written for the real database does not take effect? Answer: dialect variants (`sqlId_dialect` / `dialect_sqlId`) are looked up by the **configured dialect** by default. If the configured dialect differs from the connection's real dialect (e.g. OceanBase configured as mysql while the real database is oceanbase), since 6.0.3 you can enable `spring.sqltoy.realDialectFirst=true` — variant lookup and function replacement will then follow the real dialect first; see 2.2 realDialectFirst.

Multi-DB Verification

Since the project was initiated in 2008, sqltoy has advocated **productization**: one software product adapting to multiple databases (e.g., a CRM that simultaneously supports Oracle, MySQL, DB2, SQL Server, PostgreSQL, DM (Dameng), GaussDB, etc.), making it easy to roll out across different customer scenarios. Only productization can build real competitiveness.

1. How to Make SQL Adapt to Multiple Databases

1. **Primary-key strategy**: minimize the use of `identity` and `sequence`, which are strongly database-dependent, and prefer sqltoy's programmatic primary keys (22-digit/26-digit/snowflake/UUID, etc., see Primary Key Strategies).
2. **Inserts/updates/deletes**: use sqltoy's object-based operations (save/update/delete/load) and let the framework handle dialect differences.
3. **Queries**: enable `spring.sqltoy.functionConverts=default` so that functions in SQL are automatically adapted and replaced per database (see Dialect Adaptation & Functions).
4. **Constant-row pseudo-tables**: for "select constant-row union real-query" SQL (dictionary dropdown fallback rows, fixed report total rows, etc.), use the virtual-table macro `@virtualTable()` (new in 6.0.6) — expanded at parse time per dialect into `from dual` / `from sysibm.sysdummy1` / `from dummy`, or simply stripped, so you no longer maintain per-database `from dual` forms; see Dialect Adaptation & Functions.
5. When necessary, use `sqlId_dialect` / `dialect_sqlId` to provide dedicated SQL for specific databases; since 6.0.3 you can also declare a `dialect` attribute on the `<sql>` element to solidify parsing in a specified dialect, and enable `spring.sqltoy.realDialectFirst` so variant lookup prefers the real dialect detected from the connection — see Dialect Adaptation & Functions.

2. Enable Cross-Database Adaptation Testing (redoDataSources)

To **also verify while testing that SQL runs correctly on other databases**, sqltoy provides the `redoDataSources` mechanism: after configuring the other types of databases the product needs to support, executing any query makes the framework **execute it once on the redoDataSources data sources at the same time**, thereby verifying the SQL's cross-database adaptability.

Step 1: Configure multiple data sources with `@Configuration` (focus on the two `@Bean` definitions for data sources), for example the primary database + a PostgreSQL database `pgdb`.

Step 2: Configure the databases to be verified

```
spring.sqltoy.redoDataSources[0]=pgdb
```

Once configured, whenever any query feature is executed, sqltoy replays (redo) that SQL once on `pgdb`. If the SQL contains incompatible dialect constructs, the replay exposes them immediately, so cross-database issues are discovered during development/testing instead of surfacing as errors only when the database is switched at a customer site.

> Tip: `redoDataSources` is for **adaptation verification** only; replayed queries do not affect the primary query result. It can be turned off in production.

3. Supporting Tools and Verification Projects

Tool	Purpose
sagacity-db-diversity	**Schema difference comparison**: connect to a reference database and a target database separately, export tables/columns/primary keys/foreign keys/indexes/comments to xml, then compare offline. It supports one-sided connections (a dev environment reachable only through a bastion) and comparing exports taken at different times. Run it before a database migration and clear the schema differences first
sqltoy-aot-verify	**Native-build verification matrix** (Spring + Solon, PostgreSQL pgvector + postgis) that also covers dialect function conversion and both cache-translate implementations
sqltoy-showcase	Full-feature demo project (two H2 databases simulating a sharded topology + real Mongo/ES via Testcontainers); for **per-database verification** see Lesson 9 of sqltoy-quickstart: the same function-conversion SQL run against dockerized PostgreSQL / SQL Server / Oracle / MySQL

Caveats for `redoDataSources`:

- Only **queries** are replayed; writes are not. Use it in development/testing and switch it off in production;
- Every query hits the database one extra time, so **test-environment latency and connection usage roughly double** — do not enable it during load tests;
- It exposes **dialect incompatibilities**, and is no substitute for business result validation (the same SQL can differ subtly per database: case sensitivity, ordering, null ordering).

4. Related

- Function auto-replacement and dialect sqlId: Dialect Adaptation & Functions
- Adapting unlisted PostgreSQL-family databases (dialect/dialectMap): SQL Showcase
- The full migration assessment checklist: Database Migration Guide

Primary Key & DDL

Primary Key Strategies

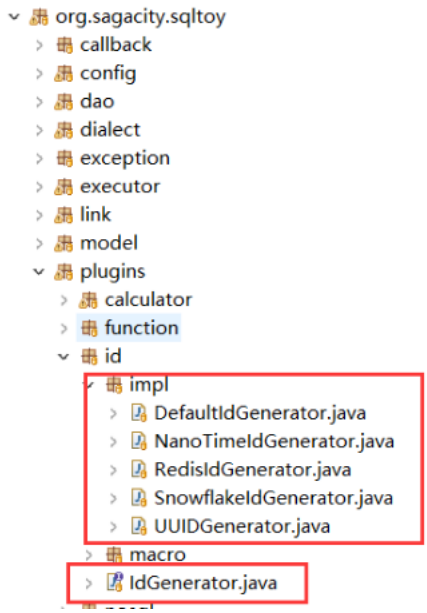
SqlToy primary keys fall into two families: **programmatic keys** (produced by the framework or a custom generator, independent of the database) and **database keys** (`sequence` / `identity` , produced by the database). Programmatic keys behave identically across databases and are the recommended choice for productized software.

1. Strategy overview

Strategy keyword (the `generator` value in quickvo / `@Id.generator`)	Implementation (`org.sagacity.sqltoy.plugins.id.impl`)	Shape
`default`	`DefaultIdGenerator`	22-digit ordered number: 13-digit current millis + 6-digit nanos + 3-digit host id, e.g. `159221418407222190021 9`
`nanotime`	`NanoTimeIdGenerator`	26-digit ordered number: 15-digit `yyMMddHHmmssSSS` + 6-digit nanos + 2 digits (thread id + random) + 3-digit host id
`snowflake`	`SnowflakeIdGenerator`	Snowflake, ordered number
`uuid`	`UUIDGenerator`	32-character UUIDv7 (ordered)
`ulid`	`ULIDGenerator`	26-character ordered string: 10 characters of timestamp + 16 of randomness (requires `ulid-creator`)
`redis`	`RedisIdGenerator`	Business key: rule-based, e.g. an order number of "date + sequence"
`sequence` / `identity` / `assign`	Database or caller	Database sequence / auto-increment column / manual assignment

> Keywords are case-insensitive; spellings such as `defaultIdGenerator` or `snowflakeIdGenerator` also work (the framework tolerates the naming variants).

The built-in generators (`plugins.id.impl` ; the screenshot predates ULID):



`ULID` needs an extra dependency:

```
<dependency>
       <groupId>com.github.f4b6a3</groupId>
       <artifactId>ulid-creator</artifactId>
       <!-- adjust the version as needed -->
       <version>5.2.4</version>
</dependency>
```

2. Specifying the strategy in quickvo.xml

```
<!-- identity columns are detected automatically; everything else is inferred from type/length.
     strategy: sequence (needs a sequence name) | assign (manual) |
               generator (default 22-digit | nanotime 26-digit | snowflake | uuid | ulid | redis | custom class) -->
<primary-key>
    <!-- a regex may match table names in bulk -->
    <table name="SQLTOY_\w+|LESSON_\w+|SYS_\w+" strategy="generator" generator="default" />
    <!-- a database sequence -->
    <table name="XXX_TABLE" strategy="sequence" sequence="SEQ_XXXX" />
    <!-- snowflake -->
    <table name="sys_staff_info" strategy="generator" generator="snowflake" />
</primary-key>
```

Automatic inference (when `primary-key` is not configured): `bigint` → snowflake; a 22-character column → `DefaultIdGenerator`; a 26-character column → `NanoTimeIdGenerator`; an `identity` column → generated by the database, no configuration needed.

A real sequence configuration:

```
<primary-key>
    <table name="SQLTOY_SEQ_SHOWCASE" strategy="sequence" sequence="SEQ_SQLTOY_SEQ_CASE"/>
</primary-key>
```

3. The generated annotation

Based on that configuration quickvo writes `@Id` onto the primary-key field:

```
// strategy="generator" with a generator keyword
@Id(strategy = "generator", generator = "org.sagacity.sqltoy.plugins.id.impl.NanoTimeIdGenerator")
@Column(name = "ID", type = java.sql.Types.VARCHAR, length = 26, nullable = false)
private String id;

// strategy="sequence" points at the sequence name
@Id(strategy = "sequence", sequence = "SEQ_XXXX")
@Column(name = "ID", type = java.sql.Types.BIGINT, nullable = false)
private Long id;
```

> `@Id` only describes the generation strategy while `@Column` describes the column; both belong on the primary-key field. See Annotations Cheat Sheet · @Id.

What quickvo actually generates (screenshot):

```
/**
 * ID
 */
@Id(strategy="sequence",sequence = "SEQ_SQLTOY_SEQ_CASE")
@Column(name="ID",type=java.sql.Types.INTEGER,nullable=false)
protected String id;
```

4. Redis business keys

The redis strategy belongs to **business keys** (order numbers, document numbers —keys with a rule). quickvo declares them with `business-primary-key`:

```
<!-- macros: @substr(${field},startIndex,length) @case(value,case1,value1,...) @df(yyMMdd)
     use @df('') when no date is wanted; related-columns lists the source fields -->
<business-primary-key>
    <table name="LESSON_DEVICE_ORDER" column="ORDER_ID"
           generator="redis" signature="PO@df(yyMMdd)" length="16" />
</business-primary-key>
```

A fuller real configuration (macro composition plus source fields):

```xml
<!-- 基于redis产生有规则的业务主键 -->
<business-primary-key>
    <!-- 1位购销标志_2位设备分类代码_6位日期_3位流水（如果当天超过3位会自动扩展到4位） -->
    <!-- 业务主键可以充当实际的主键，如果字段是主键会覆盖原来的主键策略 -->
    <!-- 包含的三个宏：
          @substr(${field},startIndex,length)
          @case(value,case1,value1,case2,value2,other)
          @df(yyMMdd) 日期格式化，如果不需要加入日期可以用@df('')
    -->
    <table name="SQLTOY_DEVICE_ORDER_INFO" column="ORDER_ID" generator="redis"
        signature="${psType}@case(${deviceType},PC,PC,NET,NT,OFFICE,OF,SOFTWARE,SF,OT)@day(yyMMdd)"
        related-columns="psType,deviceType" length="12" />
</business-primary-key>
```

> When the business-key column is itself the primary key (`@Id`), it **overrides** the original key strategy —the business number becomes the physical primary key; `length` and the sequence digits decide the shape of the number.

Generating keys with redis requires a `RedisTemplate` on the application side.

@BusinessId and generateBizId

At the configuration level **a table supports only one** business key. When a table needs several, use the `@BusinessId` annotation together with `generateBizId` and assign the values yourself:

```
/** order number (business key) */
@BusinessId(signature = "HW@case(orderType,SALE,SC,BUY,PO)@day(yyMMdd)",
        generator = "org.sagacity.sqltoy.plugins.id.impl.RedisIdGenerator",
        length = 20, sequenceSize = 5, relatedColumns = {"orderType"})
private String orderId;
```

`@BusinessId` attributes:

Attribute	Notes
`signature`	Signature expression, supporting macros such as `@case(name,v1,then1,v2,then2)` , `@day(yyMMdd)` and `@substr(name,start,length)`
`generator`	Generator implementation (e.g. `RedisIdGenerator`)
`length` / `sequenceSize`	Total key length / trailing sequence digits (-1 uses the global setting)
`relatedColumns`	Columns whose values feed the generation
`start`	Initial sequence value, default 1

Generating through `LightDao` :

```
// 1) use the @BusinessId strategy configured on the entity
String orderId = lightDao.generateBizId(orderInfo);

// 2) generate with explicit parameters (business keys are managed per table; signature supports @case/@day/@substr)
String bizId = lightDao.generateBizId("sqltoy_order_info",
        "HW@case(orderType,SALE,SC,BUY,PO)@day(yyMMdd)",
        MapKit.map("orderType", "SALE"), null, 20, 5);

// 3) simple form: an identifier plus an increment
long id = lightDao.generateBizId("ORDER", 1);
```

> 🎯 A complete runnable example lives in `BusinessIdAndDataVersionTest.java` of the `sqltoy-showcase` project (it also demonstrates the `@BusinessId` annotation and is annotated `@Disabled` because it needs redis).

5. Using a custom key strategy

Implement the `IdGenerator` interface (package `org.sagacity.sqltoy.plugins.id` ; older material often spells it `IdGenarator` , which is a typo):

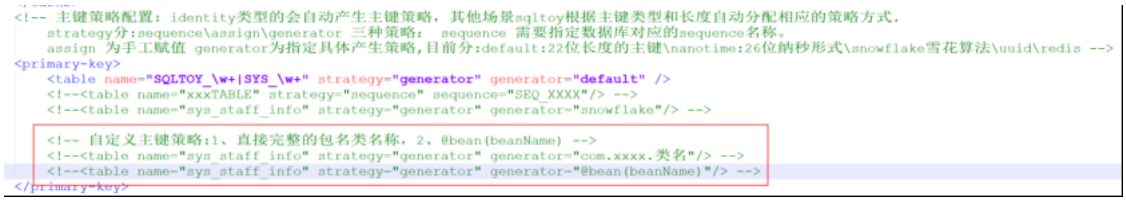
```
public interface IdGenerator {
    /**
     * @param tableName       table name (some strategies isolate keys per table)
     * @param signature       signature symbol
     * @param relatedColumns  related column names
     * @param relatedColValue related column values
     * @param bizDate         current date
     * @param idJavaType      Java type of the key
     * @param length          key length
     * @param sequenceSize    sequence digits (e.g. 20210709[001] has three trailing digits)
     */
    public Object getId(String tableName, String signature, String[] relatedColumns, Object[] relatedColValue,
            Date bizDate, String idJavaType, int length, int sequenceSize);

    /** Optional: initialisation, invoked by the framework at startup */
    public default void initialize(SqlToyContext sqlToyContext) throws Exception { }
}
```



Then point `generator` at the implementation in quickvo.xml, in **either of two forms**: the fully qualified class name, or `@bean(beanName)` (the generator is managed by the Spring container, so dependencies can be injected):

```
<primary-key>
    <table name="SQLTOY_\w+" strategy="generator"
           generator="com.yourproject.MyIdGenerator" />
    <!-- or: let the Spring container provide it -->
    <table name="SQLTOY_\w+" strategy="generator"
           generator="@bean(myIdGenerator)" />
</primary-key>
```



Or annotate the POJO directly, bypassing quickvo:

```
@Id(strategy = "generator", generator = "com.yourproject.MyIdGenerator")
```

> If the custom generator depends on Spring beans (a redis template, for instance), obtain them through `initialize(SqlToyContext)` or inside the implementation. Under native images the class needs reflection registration —see the GraalVM Native Image Guide.

6. Handling duplicate keys

`default`, `nanotime` and `snowflake` rely on a **host id** to separate workers. Several application instances on one IP will therefore collide, so set it explicitly:

```
# default, nanotime: three digits
java -Dsqltoy.server.id=112

# snowflake: two numeric parameters, both below 32
java -Dsqltoy.snowflake.workerId=11
java -Dsqltoy.snowflake.dataCenterId=20
```

> In containers and cloud environments (where pod IPs change), either configure the parameters above explicitly or switch to the `redis` strategy / a custom generator that allocates keys centrally.

7. Related

- Attribute tables for `@Id` / `@BusinessId`: Annotations Cheat Sheet
- Dialect differences when returning inserted keys: Dialect Adaptation & Functions
- Key strategy assessment during a database migration: Database Migration Guide

POJO to DDL

- sqltoy supports generating DDL scripts from POJO entities at application startup and executing them in the database

```
spring:
	sqltoy:
		# specify the POJO package paths to scan; you can specify a top-level package for automatic recursive scanning, or a lower-level
package
		packagesToScan:
			- com.example.modules.commons.entity
			- com.example.modules.system.entity
		autoDDL: true
		#dialectDDLGenerator: com.xxx custom extension implementation class
```

- Generate the database script for all POJOs in the project via code

```
@Test
public void testCreateSqlFile() {
	// specify the package path where the POJOs reside
	String[] scanPackages = new String[] { "org.sagacity.sqltoy.demo.domain" };
	try {
		/**
		 * @param scanPackages
		 * @param saveFile            file where the script is saved
		 * @param upperOrLower        upper|lower whether table names and column names in the script are uniformly converted to upper or lower
case
		 * @param dbType              database type, provided via DBType.xx
		 * @param schema              required for sqlserver (can be null for other databases); at application startup the schema is obtained
from the connection
		 * @param dialectDDLGenerator your own DDL creator; for databases such as mysql, oracle, pg, sqlserver, no extension is needed, just
pass null
		 */
		DDLFactory.createSqlFile(scanPackages, "D://sqltoy.sql", "upper", DBType.MYSQL, null, null);
	} catch (Exception e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
}
```

Entity Annotations and DDL Metadata

DDL generation supports the following entity annotations (`@Partition` / `@PartitionDef` / `@MppTable` are **new in 6.0.3**; quickvo generates them automatically from the database's partition definitions and MPP table definitions, and you can also add them by hand):

Annotation	Description
`@Foreign(table, field, constraintName)`	**Foreign key marker** (new in 6.0/2023.07): declares the table and field a column references; a foreign key constraint is created when the DDL is generated
`@Index(name, columns, ...)`	**Index definition**: declares a single- or multi-column index (optionally unique); the index is created when the DDL is generated — see section 4 below
`@Indexes({ @Index, ... })`	**Index collection**: declare multiple indexes on one entity
`@PartitionKey`	**Partition key marker**: the partition key when creating tables on MPP databases (such as StarRocks)
`@Partition`	**Partition strategy marker** (new in 6.0.3): describes the partitioning method and partition key; renders the `PARTITION BY` clause
`@PartitionDef`	**Partition detail** (new in 6.0.3): used with `@Partition` to describe each partition's name and bound
`@MppTable`	**MPP table-engine metadata** (new in 6.0.3): engine, key model, bucketing and similar definitions when creating tables on ClickHouse/Doris/StarRocks

1. Foreign keys and referential actions

```
@Accessors(chain = true)
@Entity(tableName = "sqltoy_order_info", pk_constraint = "PRIMARY")
public class OrderInfo implements Serializable {

    @Column(name = "ORGAN_ID")
    // Foreign key: references the ORGAN_ID field of the sqltoy_organ_info table
    // deleteRestict/updateRestict values: see the org.sagacity.sqltoy.config.model.ReferentialAction constants:
    // 0:CASCADE, 1:RESTRICT (default), 2:SET_NULL, 3:NO_ACTION, 4:SET_DEFAULT
    @Foreign(table = "sqltoy_organ_info", field = "ORGAN_ID", constraintName = "FK_ORDER_ORGAN", deleteRestict = 0)
    private String organId;
}
```

When generating the foreign key constraint, actions are trimmed automatically per database compatibility: Oracle/DM do not support the `ON UPDATE` clause; Oracle rejects the `NO ACTION` keyword (ORA-02000); Oracle/DM and the MySQL family (InnoDB) do not support `SET DEFAULT` — unsupported actions are skipped automatically, degrading to the default NO ACTION behavior.

2. Table partitioning (@Partition / @PartitionDef / @PartitionKey)

```
@Entity(tableName = "SQLTOY_TRANS_INFO", pk_constraint = "PRIMARY")
// Partition strategy: RANGE|LIST|HASH|RANGE_COLUMNS|LIST_COLUMNS;
// columns: partition key columns (defaults to collecting fields marked with field-level @PartitionKey; the two link both ways);
// expression: the database-native partition expression verbatim (e.g. mysql's TO_DAYS(createTime)); takes precedence when provided
@Partition(strategy = "RANGE", columns = { "BIZ_DATE" }, partitions = {
        @PartitionDef(name = "p202601", value = "2026-02-01"),   // bound value: date-form values are quoted automatically
        @PartitionDef(name = "p202602", value = "2026-03-01"),
        @PartitionDef(name = "pmax", value = "MAXVALUE") })      // catch-all partition
public class TransInfo implements Serializable {

    @Column(name = "BIZ_DATE")
    @PartitionKey   // field-level partition marker, linked both ways with @Partition.columns
    private LocalDate bizDate;
}
```

The generated MySQL create-table statement renders automatically:

```
CREATE TABLE SQLTOY_TRANS_INFO (...)
PARTITION BY RANGE (BIZ_DATE) (
   PARTITION p202601 VALUES LESS THAN ('2026-02-01'),
   PARTITION p202602 VALUES LESS THAN ('2026-03-01'),
   PARTITION pmax VALUES LESS THAN (MAXVALUE)
)
```

Partition detail rendering rules: RANGE partitions are sorted by upper bound ascending (the MAXVALUE catch-all partition goes last); LIST partitions render `VALUES (v)` on Oracle/DM and `VALUES IN (v)` on the MySQL family; HASH/KEY strategies render `PARTITIONS n` by partition count; on strategy-based partitioning databases such as pg/oracle, when no details are present they remain in comment form and partitions are managed by the database itself.

3. MPP analytical-database table engines (@MppTable)

Creating tables on ClickHouse/Doris/StarRocks requires engine, sort key, bucketing and similar metadata, described via `@MppTable`:

```
@Entity(tableName = "SQLTOY_ORDER_OLAP")
@MppTable(
    engine = "OLAP",                          // ClickHouse: MergeTree/ReplacingMergeTree etc.; Doris/StarRocks: OLAP
    keyModel = "DUPLICATE",                   // Doris/StarRocks key model: UNIQUE/PRIMARY/DUPLICATE/AGGREGATE
    orderBy = { "EVENT_TIME", "ORDER_ID" },   // sort key: ClickHouse's ORDER BY columns; Doris/SR's KEY-model key columns
    distributedBy = { "ORDER_ID" },           // bucketing columns: Doris/StarRocks's DISTRIBUTED BY HASH columns
    buckets = 10,                             // bucket count; by default the database decides
    properties = { "replication_num=3" })     // table properties: CK renders SETTINGS k=v; Doris/SR renders PROPERTIES("k"="v")
@Partition(strategy = "RANGE", columns = { "EVENT_TIME" })
public class OrderOlap implements Serializable { ... }
```

The Doris/StarRocks create-table statement looks like:

```
CREATE TABLE SQLTOY_ORDER_OLAP (...)
ENGINE = OLAP
DUPLICATE KEY(EVENT_TIME, ORDER_ID)
COMMENT 'order detail'
PARTITION BY RANGE (EVENT_TIME) (...)
DISTRIBUTED BY HASH(ORDER_ID) BUCKETS 10
PROPERTIES ("replication_num" = "3")
```

The ClickHouse create-table statement looks like (`engine=ReplacingMergeTree` , `engineArgs` carrying engine arguments such as the version column, `properties` rendered as `SETTINGS`):

```
CREATE TABLE SQLTOY_ORDER_OLAP (...)
ENGINE = ReplacingMergeTree(ver)
ORDER BY (EVENT_TIME, ORDER_ID)
PARTITION BY EVENT_TIME
SETTINGS index_granularity = 8192
```

4. Index definitions (@Index / @Indexes)

Declare indexes on the entity class so autoDDL creates them together with the table; quickvo 2.x also generates this annotation automatically from the database's existing index definitions (hand-written whitelist annotations are not overwritten):

```
@Indexes(indexes = {
    // Regular index: name (required), columns (required), isUnique (default false), sortTypes (asc/desc, ascending by default)
    @Index(name = "idx_staff_name", columns = { "staffName" }, sortTypes = { "asc" }),
    // Unique index
    @Index(name = "uniq_staff_code", isUnique = true, columns = { "staffCode" })
})
@Entity(tableName = "SQLTOY_STAFF_INFO")
public class StaffInfo implements Serializable {
    //......
}
```

Rendering rules: on MySQL-family databases the index is merged into the create-table statement as an in-table `[UNIQUE] KEY indexName (col asc/desc,...)` ; on all other databases a standalone `CREATE [UNIQUE] INDEX indexName ON tableName (...);` statement is emitted.

5. DDL generator support per database (6.0.3)

Database	Generator	Notes
MySQL / MySQL 5.7	`MySqlDDLGenerator`	supports the `@Partition` partition clause
Doris / StarRocks	`DorisDDLGenerator` / `StarRocksDDLGenerator`	dedicated generators new in 6.0.3 (previously reused the MySQL generator); support `@MppTable` engine/key model/bucketing/properties and partition clauses; VARCHAR beyond 65533 degrades to STRING automatically
ClickHouse	`ClickHouseDDLGenerator`	new in 6.0.3; supports the MergeTree engine family, `ORDER BY` , `SETTINGS` and `PARTITION BY` expressions
PostgreSQL family (PG/GaussDB/OpenGauss/MogDB/Vastbase/StarDB/Oscar/Kingbase)	`PostgreSqlDDLGenerator`	includes Kingbase coverage added in 6.0.3
Oracle / Oracle11 / DM	`OracleDDLGenerator`	
SQL Server	`SqlServerDDLGenerator`	
DB2	`DB2DDLGenerator`	dedicated generator new in 6.0.3 (incl. CLOB and JSON→CLOB carrier)
SAP HANA	`HanaDDLGenerator`	new in 6.0.3 (incl. NCLOB large text and JSON→NCLOB carrier)
SQLite	`SQLiteDDLGenerator`	new in 6.0.3
H2	`H2DDLGenerator`	

Other typing details: JSON columns degrade gracefully on databases without a native JSON type (HANA→NCLOB, SQL Server→NVARCHAR(MAX), DB2/Oracle11→CLOB); the pgvector vector type renders `(n)` only when the dimension is valid, and an unbounded vector renders as `vector` .

> These annotations only affect DDL generation (autoDDL / DDLFactory) and do not change runtime behavior (`@PartitionKey` also serves as the partition key hint for DML on MPP databases).

Enterprise Features

Multi-Tenancy

Introduction

With the growth of SaaS adoption and some users' special scenario requirements, it often becomes necessary to centrally process the SQL in a project, for example:

- **Tenant filtering**: automatically append the tenant ID condition to prevent data privilege escalation.
- **SQL injection validation**: enhances security.
- **Schema modification**: replace the actual schema as needed (e.g. `select xxx from schema.table`).

Usage

1. Configuration Parameters

Register interceptors by configuring the `spring.sqltoy.sqlInterceptors` array. The SQLToy framework provides `TenantFilterInterceptor` out of the box, which performs tenant filtering specifically for object operations and single-table operations such as `findEntity`.

Configuration file example (application.yml):

```
spring:
  sqltoy:
    # Keep sqltoy at the start, so developers do not overlook the spring.sqltoy prefix and write sqltoy instead, which would prevent loading
    # Separate multiple paths with commas (please read carefully)
    sqlResourcesDir: classpath:/sqltoy/quickstart
    # Defaults to classpath:sqltoy-translate.xml; no need to set it if identical
    translateConfig: classpath:sqltoy-translate.xml
    # Handles special types such as json (optional configuration)
    typeHandler: com.sqltoy.plugins.JsonTypeHandler
    # Can be default, defaultFormatter, defaultSqlFormatter, or a concrete class (with package name); the ones provided by sqltoy require the druid
jar
    sqlFormatter: defaultSqlFormatter

    # SQL interceptors, which can rewrite SQL, e.g. multi-tenancy, SQL injection validation, schema rewriting, etc.
    sqlInterceptors:
      # If a concrete package path and class name is given, sqltoy will instantiate it automatically
      - org.sagacity.sqltoy.plugins.interceptors.TenantFilterInterceptor
      # You can also register a spring bean by putting the bean name here
      # - yourInterceptorBeanName
```

2. Entity Class Annotation

If you use SQLToy's built-in tenant filtering, you need to add the `@Tenant` annotation on the POJO to specify the tenant field.

Entity class example (StaffInfoVO.java):

```
/**
 * @project sqltoy-mssql
 * @author zhongxuchen
 * @version 1.0.0 Table: SQLTOY_STAFF_INFO
 */
// Specify the tenant field as tenantId
@Tenant(field = "tenantId")
@Entity(tableName = "SQLTOY_STAFF_INFO", comment = "", pk_constraint = "PK_SQLTOY_STAFF_INFO")
public class StaffInfoVO implements Serializable {
    // ... field definitions
}
```

3. Interceptor Logic and the Unified Fields Handler

The interceptor must be used together with the **unified fields handler (`IUnifyFieldsHandler`)** to obtain the current user's authorized tenant information.

A. Interceptor Implementation Logic (TenantFilterInterceptor.java)

The framework checks whether the entity class contains a tenant field, then obtains the authorized tenant information to perform filtering.

```
public class TenantFilterInterceptor implements SqlInterceptor {

    @Override
    public SqlToyResult decorate(SqlToyContext sqlToyContext, SqlToyConfig sqlToyConfig,
                                 OperateType operateType, SqlToyResult sqlToyResult, Class entityClass, Integer dbType) {
        // A unified fields handler exists and the operation targets an entity object
        if (sqlToyContext.getUnifyFieldsHandler() == null || entityClass == null) {
            return sqlToyResult;
        }

        EntityMeta entityMeta = sqlToyContext.getEntityMeta(entityClass);

        // No tenant filtering configured
        if (entityMeta.getTenantField() == null) {
            return sqlToyResult;
        }

        // [Optional] You can also use this approach to check for a tenant field, instead of the @Tenant annotation
        // if (entityMeta.getColumnName("tenantId") != null) { ... }

        // Skip filtering when the authorized tenant information is empty
        // Get the current user's authorized tenants via the unified fields handler
        String[] tenants = sqlToyContext.getUnifyFieldsHandler().authTenants(entityClass, OperateType.QUERY); // assuming a query operation

        if (tenants == null || tenants.length == 0) {
            return sqlToyResult;
        }

        // ... then rewrite the SQL, appending and tenant_id in (...)
        return sqlToyResult;
    }
}
```

B. Unified Fields Handler Implementation (SqlToyUnifyFieldsHandler.java)

Implement the `IUnifyFieldsHandler` interface and return the current logged-in user's tenant IDs in the `authTenants` method.

```
/**
 * @project sqltoy-showcase
 * @description unified fields assignment example
 * @author chenrenfei
 * @version id:SqlToyUnifyFieldsHandler.java, Revision:v1.0, Date:2018-01-18
 */
public class SqlToyUnifyFieldsHandler implements IUnifyFieldsHandler {

    private String defaultUserName = "system-auto";

    /**
     * Get the current user's authorized tenant information, used for unified tenant filtering
     * during data operations,
     * e.g.: update xxxx set xx=xx where id=? and tenant_id='S0002'
     */
    @Override
    public String[] authTenants(Class entityClass, OperateType operType) {
        // Note: this is a simulated value; in practice, obtain it dynamically from the current user's information
        // e.g. get the logged-in user's tenant ID from SecurityContext or ThreadLocal
        return new String[] { "S0002" };
    }
}
```

Data Permission Passing and Overreach Check

In enterprise systems, "which organizations / account sets / tenants / business lines may the current user see" is information that **every query must carry but business code should not care about**. SqlToy splits this into two capabilities:

Capability	Problem it solves	Implemented in
Unified parameter filling + overreach check	Front end omitted it → fill in the authorized values; front end passed it → check for overreach and throw immediately	`IUnifyFieldsHandler.dataAuthFilters()`
Automatic filtering by dimension	The parameter should not even come from the front end: inject `tenant_id='x'` style conditions into the SQL	`SqlInterceptor` (built-in `TenantFilterInterceptor`)

1. Unified parameter filling + overreach check (dataAuthFilters)

1.1 How it works

`IUnifyFieldsHandler.dataAuthFilters()` returns a "**SQL parameter name → authorization config**" map. Before every query/execution the framework inspects the **same-named parameters** in the SQL:

```
public default IgnoreKeyCaseMap<String, DataAuthFilterConfig> dataAuthFilters() {
    return null;   // disabled by default
}
```

Behaviour, judged per named parameter:

What the front end passed	What the framework does
Nothing (null/blank), or the `choiceAllValue` ("select all" marker)	**Fills the parameter with the authorized range** `values` (unified passing; business code does nothing)
A value, with `isForcelimit=true`	Compares every passed value against the authorized set; **any value outside it throws** `DataAccessException`
A value, with `isForcelimit=false`	No overreach check; the passed value is used as-is

An overreach exception looks like this:

```
param [organId] value [S004] is beyond the authorized scope
(see the implementation configured by spring.sqltoy.unifyFieldsHandler), please check!
```

1.2 DataAuthFilterConfig attributes

Attribute	Notes
`values`	**The authorized values**: an array, collection or single value taken from the logged-in user's authorized scope (organ ids, account-set ids, tenant ids...)
`choiceAllValue`	The "select all" marker (e.g. a dropdown passing `-1`); when passed, it is treated as unspecified and `values` is used instead
`isForcelimit`	Whether to enforce the constraint (default `true`); `false` disables the overreach check
`ignoreType`	Whether to ignore type differences during comparison (e.g. the front end sends `String` while the authorized value is `BigInteger`)

1.3 Example implementation

```
@Component("unifyFieldsHandler")
public class SqlToyUnifyFieldsHandler implements IUnifyFieldsHandler {

    /** Current-user context (written into a ThreadLocal by a filter/interceptor) */
    @Override
    public IgnoreKeyCaseMap<String, DataAuthFilterConfig> dataAuthFilters() {
        IgnoreKeyCaseMap<String, DataAuthFilterConfig> map = new IgnoreKeyCaseMap<>();
        // (1) organization dimension: the :organId parameter in SQL
        map.put("organId", new DataAuthFilterConfig()
                .setValues(getAuthOrganIds())      // String[]: authorized organizations
                .setChoiceAllValue("-1")           // the "all" option passes -1, so use the authorized range
                .setForcelimit(true));             // overreach throws
        // (2) account-set dimension: authorized values are numeric, the front end sends strings
        map.put("accountSetId", new DataAuthFilterConfig()
                .setValues(getAuthAccountSetIds())
                .setIgnoreType(true));
        return map;
    }
}
```

Configuration (`unifyFieldsHandler` also fills common fields; this mechanism reuses the same implementation):

```
spring.sqltoy.unifyFieldsHandler=com.yourproject.SqlToyUnifyFieldsHandler
```

1.4 What the SQL looks like

Just make the parameter name match a key of `dataAuthFilters()` — no extra tags:

```
<sql id="staff_findlist">
    <value><![CDATA[
        select * from sqltoy_staff_info t
        where t.status=1
        #[and t.organ_id in (:organId)]
        #[and t.tenant_id=:tenantId]
    ]]></value>
</sql>
```

- Front end omits `organId` (or passes `-1`) → the framework fills in the authorized organizations;
- Front end passes `organId=S004` while only A/B/C are authorized → an overreach exception is thrown;
- Multiple values (arrays, or a comma string split by the `<split>` filter) are compared one by one.

1.5 Caveats

- **Parameter names must line up**: this is the classic trap — renaming the parameter in SQL without updating `dataAuthFilters()` silently disables the mechanism (an empty parameter just gets the authorized values, so it still "looks fine" while the overreach check never runs);
- **The authorized values must come from a trusted source**: `values` should be derived server-side from the login state, never from the front end;
- **Only SQL that actually has the named parameter is affected**: without the parameter in the SQL, no condition is invented (for "always filter" use the interceptor in section 2);
- **Paginated count caching**: `values` participates in the pagination count cache key, so a change of authorized scope naturally yields a different key and counts never leak across scopes;
- The single-table entity paths (`findEntity` / `updateByQuery` / `deleteByQuery`) behave exactly like non-entity queries.

2. Automatic filtering by dimension (TenantFilterInterceptor)

When the condition must be on **every** query and the front end never passes it, a SQL interceptor is more thorough: it injects the condition into the final SQL, so neither the business SQL nor its parameters mention that column.

2.1 Configuration

```
# built-in tenant/organization filtering interceptor
spring.sqltoy.sqlInterceptors[0]=org.sagacity.sqltoy.plugins.interceptors.TenantFilterInterceptor
# the interceptor takes the authorized values from the unified field handler
spring.sqltoy.unifyFieldsHandler=com.yourproject.SqlToyUnifyFieldsHandler
```

2.2 Where the authorized values come from

The interceptor asks `IUnifyFieldsHandler` for two values:

```
/** The current user's authorized tenant/organization ids; operType distinguishes query/page/top/random/count etc. */
public default String[] authTenants(Class entityClass, OperateType operType) { ... }

/** Single-tenant simplification: return just the current user's tenant id */
public default String getUserTenantId() { ... }
```

`authTenants` wins; when it returns nothing, `getUserTenantId` is used. If both are empty, **no filtering happens** (fail-open — "not configured means not filtered", not "filter everything").

> For pagination, implement `authTenants`: the paginated count cache needs that value as part of its key, otherwise different tenants share one count.

2.3 Injection rules

- The tenant column comes from `@Tenant(field="...")` on the entity, or from the conventional column name (e.g. `tenantId`);
- It only applies to operations **that have an entity type** (object CRUD, `findEntity`, `findPageEntity`, `updateByQuery`, `deleteByQuery`, etc.); **a plain sqlId query has no entity and is not injected**;
- When the SQL already carries that tenant condition (`tenant_id=` / `tenant_id in` already present in the `where` part) it is **not injected twice**;
- SQL without a `where` gets the condition inserted before `group by` / `order by`;
- For `saveOrUpdate` (merge) it is injected into the `on` clause;
- Authorized values are spliced in as **literals** (single quotes are escaped by the framework), so they must come from a trusted source.

2.4 Relation to multi-tenancy

This mechanism is also one way to implement multi-tenant isolation: `@Tenant` + `TenantFilterInterceptor` filter and fill consistently. For the full multi-tenancy story (including filling the tenant column on write) see Multi-Tenancy.

3. Choosing between the two

Scenario	Choice
The page has an organization/account-set filter the user may or may not set	`dataAuthFilters()` (unified filling + overreach check)
Isolation must always apply (SaaS tenant, hard data isolation)	`TenantFilterInterceptor` (automatic injection)
Isolation plus a **clear error** when the front end overreaches	`dataAuthFilters()` with `isForcelimit=true`
The filtering dimension is not an entity column (cross-table, complex expression)	A custom `SqlInterceptor`, see SQL Interceptors

> The two can be combined: the interceptor provides the baseline isolation, `dataAuthFilters` handles filling the visible page condition and reporting overreach.

4. Related

- Full multi-tenancy usage: Multi-Tenancy
- Custom SQL processing: SQL Interceptors
- The unified field handler interface (`createUnifyFields` / `updateUnifyFields`): Common-Field Auto-Fill
- All configuration properties: Configuration

SQL Interceptors

With the rise of SaaS and various special-scenario requirements, it is often necessary to process the SQL in a project in a **centralized** way, for example: unified tenant filtering, SQL injection validation, schema rewriting (replacing the actual schema in `select xxx from schema.table` on demand), and so on. sqltoy implements this through the **SQL interceptor (SqlInterceptor)** mechanism, i.e. SQL interception & rewriting.

1. Configuration

Interceptors are registered via the configuration parameter `spring.sqltoy.sqlInterceptors` (an array); multiple interceptors can be configured:

```
spring.sqltoy.sqlInterceptors[0]=org.sagacity.sqltoy.plugins.interceptors.TenantFilterInterceptor
spring.sqltoy.sqlInterceptors[1]=com.yourpkg.YourSqlInterceptor
```

2. The SqlInterceptor Interface

A custom interceptor implements the `org.sagacity.sqltoy.plugins.SqlInterceptor` interface, whose core consists of two default methods:

```
public interface SqlInterceptor {
    // Intercepts the final sql and parameters (SqlToyResult) before execution and returns the processed result
    default SqlToyResult decorate(SqlToyContext sqlToyContext, SqlToyConfig sqlToyConfig,
                                  OperateType operateType, SqlToyResult sqlToyResult,
                                  Class entityClass, Integer dbType) {
        return sqlToyResult;
    }
    // Returns the tenant field names of the entity (used for tenant filtering, e.g. saveOrUpdate / merge-into on conditions)
    default String[] tenantFieldNames(EntityMeta entityMeta, OperateType operateType) {
        // ...
    }
}
```

Inside `decorate` you get the SQL and parameters that are about to execute, so you can rewrite the SQL, append filter conditions, inject validation, replace the schema, and more.

3. Built-in Tenant Filter Interceptor

The framework provides `TenantFilterInterceptor` by default, which automatically appends tenant filter conditions for **object operations and single-table operations such as** `findEntity`. Using it requires:

1. **Adding the** `@Tenant(field="xxx")` **annotation to the POJO** (optional, marks the tenant field):

```
@Tenant(field = "tenantId")
@Entity(tableName = "sqltoy_order_info")
public class OrderInfo implements Serializable {
    private String tenantId;
    // ...
}
```

2. **Providing the current user's authorized tenants via the unified fields handler (** `IUnifyFieldsHandler` **)**; based on this the interceptor decides whether the POJO contains a tenant field and appends the filtering.

Reference implementation: `org.sagacity.sqltoy.plugins.interceptors.TenantFilterInterceptor`.

> For the complete multi-tenancy configuration, including `@Tenant`, the unified fields handler and privilege-escalation filtering, see Multi-Tenancy and Data Permission & Overreach Check.

Data Versioning

> Supported since sqltoy 4.19.62.

1. Purpose

Prevents data from being changed by other users while the current user is working on it, so the current user can be effectively alerted (optimistic locking semantics).

2. Logic

1. A version number is automatically assigned when the data is **created**;
2. On **update**, the page holds the current version number of the data;
3. When the update is submitted, the **version is checked for consistency**: if it matches, the data is updated and the version number is incremented by 1; otherwise an exception is thrown, indicating that the data has been changed by someone else.

3. Usage

Add the `@DataVersion` annotation to the POJO class to mark the object as requiring version control:

```
@DataVersion(field = "dataVersion", startDate = false)
@Entity(tableName = "sqltoy_order_info", pk_constraint = "PRIMARY")
public class OrderInfo implements Serializable {
    @Id(...)
    private String orderId;
    // Version number field (field points to this property name)
    private Long dataVersion;
    // ... other fields
}
```

`@DataVersion` attributes:

Attribute	Default	Description
`field`	`""`	Version number field name (the object's property name)
`startDate`	`false`	Whether the version number starts with a date, e.g. `202209231` (with `true` it is easy to distinguish version batches by date)

The current version number must be carried when updating data: when submitting the update, the front end / caller must send back the `dataVersion` obtained at query time; the framework uses it to perform the version check.

```
// Get dataVersion at query time
OrderInfo order = lightDao.load(queryVO);
// Submit after modification; the original dataVersion must be kept for the check
order.setStatus(2);
lightDao.update(order);   // If the version matches, update and increment by 1; otherwise an exception is thrown
```

> Data versioning can be combined with other enterprise-grade features such as the elastic update and data permission.

Data Masking & Encryption

Data Masking

- sqltoy masking covers the fixed types tel\name\id-card\bank-card\address\email\public-account
- For characters of relatively fixed length, you can define your own masking rules via head-size, tail-size and mask-code
- For non-fixed-length characters, set the proportion via the mask-rate masking ratio, e.g.: mask-rate="50"
- You can also define your own masking handler

```
# Implement org.sagacity.sqltoy.plugins.secure.DesensitizeProvider
# Default implementation class: org.sagacity.sqltoy.plugins.secure.impl.DesensitizeDefaultProvider
spring.sqltoy.desensitizeProvider=your.DesensitizeProvider
```

- XML usage

```
<sql id="sys_findStaff_info">
       <!-- Security mask types: tel\name\id-card\bank-card\address\email\public-account\other -->
       <!--Simplest usage: <secure-mask columns="mobile_tel" type="tel"/> -->
       <secure-mask type="tel" columns="mobile_tel" head-size="3"
              tail-size="4" mask-code="*****" />
    <value>
        <![CDATA[
        select staff_code,staff_name,address,mobile_tel,id_card,blank_type,blank_code
        from sys_staff_info
        where staff_code=:staffCode
        ]]>
    </value>
</sql>
```

- Usage in Java code

```
String sql = """
                select staff_code,staff_name,address,mobile_tel,id_card,blank_type,blank_code
                from sys_staff_info
                where staff_code=:staffCode
                            """;
List result = lightDao
              .findByQuery(
                            new QueryExecutor(sql).secureMask(MaskType.TEL, "mobile_tel").secureMask(MaskType.ID, "id_card")
                                          .values(MapKit.keys("staffCode").values("S0001")).resultType(StaffInfoVO.class))
              .getRows();
```

Field Encryption/Decryption

- sqltoy supports encrypting fields for storage and decrypting them dynamically at query time
- sqltoy encrypts/decrypts fields with an RSA key pair (public key encrypts, private key decrypts). The key format is **a single line of Base64 without the PEM headers and line breaks** (private key PKCS#8, public key X.509); one openssl command pair generates it (openssl ships with Git Bash / Linux / macOS, so Windows users have it once Git is installed):

```
# generate a 1024-bit private key (PKCS#8) and export the matching public key (X.509),
# then strip headers and squeeze to single-line Base64
openssl genrsa 1024 2>/dev/null | openssl pkcs8 -topk8 -nocrypt > private.pem
openssl rsa -in private.pem -pubout > public.pem
grep -v '^-' private.pem | tr -d '\n' > rsa_private.key && echo >> rsa_private.key
grep -v '^-' public.pem  | tr -d '\n' > rsa_public.key  && echo >> rsa_public.key
```

> Or generate with plain Java (JDK built-in JCE, zero dependencies) — the `org.sagacity.sqltoy.utils.SecureUtils` class under `src/test` of the framework source wraps exactly this logic:

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Base64;

public class GenerateRsaKeys {
    public static void main(String[] args) throws Exception {
        KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
        generator.initialize(1024); // 2048 for production; widen the encrypted column accordingly
        KeyPair keyPair = generator.generateKeyPair();
        // single-line Base64 without PEM headers — the format sqltoy expects
        // (private key PKCS#8, public key X.509); print two lines and save each to its own file
        System.out.println(Base64.getEncoder().encodeToString(keyPair.getPrivate().getEncoded()));
        System.out.println(Base64.getEncoder().encodeToString(keyPair.getPublic().getEncoded()));
    }
}
```

A 1024-bit ciphertext is about 172 characters per field, so encrypted columns should be `varchar(500)`; longer keys are fine for production as long as the column width grows accordingly.

```
# You can provide a custom encryption/decryption algorithm implementation; the framework ships a default one
# spring.sqltoy.fieldsSecureProvider=org.sagacity.sqltoy.plugins.secure.impl.FieldsRSASecureProvider
spring.sqltoy.securePrivateKey=classpath:mock/rsa_private.key
spring.sqltoy.securePublicKey=classpath:mock/rsa_public.key
```

In traditional Spring XML mode, configure the properties of the same names on the `SqlToyContext` bean:

```
<property name="securePrivateKey" value="classpath:mock/rsa_private.key" />
<property name="securePublicKey" value="classpath:mock/rsa_public.key" />
```

> 🎬 Fully runnable examples: `SecureEncryptTest.java` (encrypted storage / automatic decrypt / search mask) and `SecureMaskTest.java` (decrypt-then-mask) in the demo project `sqltoy-showcase`; the test resources already bundle the demo keys `mock/rsa_private.key` and `rsa_public.key`.

- Add the @SecureConfig annotation on the POJO class; to facilitate searching, you can additionally store a masked value — pay attention to what sourceField points to

```
@Data
@Accessors(chain = true)
@Entity(tableName="sqltoy_secure_case",comment="Security encryption/decryption demo",pk_constraint="PRIMARY")
@SecureConfig(secures = { @Secure(field = "telNoMask", secureType = SecureType.TEL),
               @Secure(field = "telNo", secureType = SecureType.ENCRYPT),
               @Secure(field = "telNoMask", secureType = SecureType.TEL,sourceField="telNo"),
               @Secure(field = "homeAddressMask", secureType = SecureType.DISCRETE_RATE, maskRate = 50, sourceField = "homeAddress"),
               @Secure(field = "homeAddress", secureType = SecureType.ENCRYPT) })
public class SecureCaseVO implements Serializable {
       @Schema(name="staffId",description="Staff ID",nullable=false)
       @Id(strategy="generator",generator="org.sagacity.sqltoy.plugins.id.impl.DefaultIdGenerator")
       @Column(name="STAFF_ID",comment="Staff ID",length=22L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
       private String staffId;

       @Schema(name="staffName",description="Name",nullable=false)
       @Column(name="STAFF_NAME",comment="Name",length=30L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
       private String staffName;

       @Schema(name="telNo",description="Mobile phone",nullable=false)
       @Column(name="TEL_NO",comment="Mobile phone",length=500L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
       private String telNo;

       @Schema(name="telNoMask",description="Phone search",nullable=false)
       @Column(name="TEL_NO_MASK",comment="Phone search",length=30L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
       private String telNoMask;

       @Schema(name="homeAddress",description="Home address",nullable=false)
       @Column(name="HOME_ADDRESS",comment="Home address",length=500L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
       private String homeAddress;

       @Schema(name="homeAddressMask",description="Home address search",nullable=false)
       @Column(name="HOME_ADDRESS_MASK",comment="Home address search",length=100L,type=java.sql.Types.VARCHAR,nativeType="VARCHAR",nullable=false)
       private String homeAddressMask;

       ......
}
```

- When the object is saved, the database columns store the encrypted and masked values

- Loading via the object automatically decrypts the values
- Decryption for SQL-based queries

```
List result = lightDao.findByQuery(new QueryExecutor(sql).secureDecrypt("address")
                            .values(MapKit.keys("staffCode").values("S0001")).resultType(StaffInfoVO.class)).getRows();
```

- Decryption in XML queries: secure-decrypt

```
<sql id="qstart_secure_decrypt">
       <!-- Decryption configuration -->
       <secure-decrypt columns="tel_no,home_address" />
       <value>
              <![CDATA[
                     select * from sqltoy_secure_case
              ]]>
       </value>
</sql>
```

Sharding

> SqlToy supports database and table sharding, though with the rise of distributed databases such as Doris and StarRocks, horizontal sharding matters less than it used to. **The usage that keeps paying off is splitting "live" tables from "history" tables by date** (recent data stays hot, older data is archived), plus hashing across several databases. This chapter covers the full strategy configuration, the three declaration styles (annotation / XML / code) and the known boundaries.

1. Two sharding dimensions

Dimension	What it decides	Where it is declared
Database (datasource)	Which datasource executes the SQL	`<sharding-datasource>` / `QueryExecutor.dbSharding(...)` / `@Sharding(db=...)`
Table	Which physical table replaces the table name in the SQL	`<sharding-table tables="..." />` / `QueryExecutor.tableSharding(...)` / `@Sharding(table=...)`

Either can be used alone, or both together (one SQL sharded across databases and several tables, each with its own strategy). **Queries** declare sharding in XML or code; **inserts/updates/deletes** declare it on the entity with `@Sharding`.

2. Sharding strategies

SqlToy ships two ready-made implementations; you can also write your own.

2.1 The strategy interface

```
public interface ShardingStrategy {
    /** Decides the table for the current SQL; returning null means "use the original table" */
    String getShardingTable(SqlToyContext sqlToyContext, Class entityClass, String baseTableName,
            String decisionType, IgnoreCaseLinkedMap<String, Object> paramsMap);

    /** Decides the database to execute on; an empty model means "use the current default datasource" */
    ShardingDBModel getShardingDB(SqlToyContext sqlToyContext, Class entityClass, String tableOrSql,
            String decisionType, IgnoreCaseLinkedMap<String, Object> paramsMap);

    /** Bean initialisation (initMethod): validates the configuration and starts the availability monitor */
    void initialize();
}
```

`paramsMap` holds the values of the fields declared by the strategy's `fields`; `decisionType` lets a strategy distinguish scenarios (query/update, etc.).

2.2 HashShardingStrategy (hash modulo, in core)

Setter	Notes
`setDataSourceMap(Map<String,String>)`	`modulo result → datasource name`, keys `"0"`, `"1"` ...
`setTableMap(Map<String,String>)`	`modulo result → physical table name`, same keys

The modulo is `Math.floorMod(hashCode, N)` (safe for negative hashes, so values never silently fall back to the base table), where `N` is the number of map entries.

```
@Bean(name = "hashDataSource", initMethod = "initialize")
public HashShardingStrategy hashDataSource() {
    HashShardingStrategy strategy = new HashShardingStrategy();
    HashMap<String, String> dataSourceMap = new HashMap<>();
    dataSourceMap.put("0", "ds0");
    dataSourceMap.put("1", "ds1");
    strategy.setDataSourceMap(dataSourceMap);
    return strategy;
}
```

> `initialize()` derives the modulus from the map size; **a null sharding value now throws and names the field**, typically because the query parameter alias does not match `@Strategy.fields`, or the sharding column is itself empty.

2.3 DefaultShardingStrategy (date-based table sharding + weighted database sharding)

This class lives in the **spring / spring-starter module** (package unchanged: `org.sagacity.sqltoy.plugins.sharding.impl`); core only keeps `HashShardingStrategy`, so a plain-Java setup must add the spring module or implement the strategy itself.

Setter	Notes
`setDays("14")` / `setDays("1,15,90")`	**Day thresholds** (ascending), matched one-to-one with the table-name array of `setTableNamesMap`
`setTableNamesMap(Map)`	`base table name → "table1,table2,table3"` (comma separated, in the same order as `days`)
`setDataSourceWeight(Map<String,Integer>)`	`datasource name → weight`, routed probabilistically (only when there is more than one entry)
`setCheckSeconds(int)`	Availability check interval (seconds)
`setDateParams("bizDate")`	Name(s) of the business-date parameter(s) used to compute the interval; the first non-null one wins

How date-based table matching works (the part that is easiest to get wrong — align it carefully):

- Compute `intervalDays = |today - businessDate|`;
- Walking `days` in the order written, take the **first** `i` where `intervalDays >= days[i]`;
- On a hit use `tables[i]` (the table array must be aligned with the same order); **when nothing matches** (the interval is smaller than the smallest threshold) it returns null and the **original table** in the SQL is used;
- The original table therefore covers the "most recent tier" — **do not put the base table itself into `tables`**, or every tier shifts and only `tables[0]` can ever be selected.

```
// Correct: base (live) table covers a 15-day window; anything older goes to the history table
@Bean(name = "realHisTable", initMethod = "initialize")
public DefaultShardingStrategy realHisTable() {
    DefaultShardingStrategy strategy = new DefaultShardingStrategy();
    strategy.setDays("14");
    HashMap<String, String> tableMap = new HashMap<>();
    tableMap.put("SQLTOY_TRANS_LOG_15D", "SQLTOY_TRANS_LOG_HIS");
    strategy.setTableNamesMap(tableMap);
    return strategy;
}

// Three tiers: base = today's table; 1-14 days -> _15; 15-89 days -> _90; 90+ -> _HIS
@Bean(name = "threeTierTable", initMethod = "initialize")
public DefaultShardingStrategy threeTierTable() {
    DefaultShardingStrategy strategy = new DefaultShardingStrategy();
    strategy.setDays("1,15,90");                      // ascending thresholds
    HashMap<String, String> tableMap = new HashMap<>();
    tableMap.put("SQLTOY_TRANS_INFO",                 // base (today)
            "SQLTOY_TRANS_INFO_15,SQLTOY_TRANS_INFO_90,SQLTOY_TRANS_INFO_HIS");
    strategy.setTableNamesMap(tableMap);
    return strategy;
}
```

Weighted routing plus automatic probing: once `setDataSourceWeight` is configured, `initialize()` starts an `IdleConnectionMonitor` daemon thread:

- It starts 60 seconds after initialisation and checks every datasource's connectivity each `checkSeconds`;
- When a database is unreachable its **weight is temporarily set to 0 with an error log**, so subsequent requests do not route there probabilistically; the weight is restored once it recovers;
- With a single datasource no probabilistic choice happens — it is used directly.

```
@Bean(name = "weightDataSource", initMethod = "initialize")
public DefaultShardingStrategy weightDataSource() {
    DefaultShardingStrategy strategy = new DefaultShardingStrategy();
    HashMap<String, Integer> weight = new HashMap<>();
    weight.put("ds1", 70);   // 70% of requests land on ds1
    weight.put("ds2", 30);
    strategy.setDataSourceWeight(weight);
    strategy.setCheckSeconds(120);   // probe every 2 minutes
    return strategy;
}
```

> Weighting is per-request probabilistic routing and gives no cross-database transactional consistency; for write-heavy, strongly consistent workloads, question whether weighted sharding is really needed.

3. Declaration style 1: entity annotation (insert/update/delete)

Annotate the POJO with `@Sharding`, after which `lightDao.save/update/delete/load` route automatically:

```
@Data
@Accessors(chain = true)
@Entity(tableName = "SQLTOY_TRANS_INFO", pk_constraint = "PRIMARY")
@Sharding(db = @Strategy(name = "hashDataSource", fields = { "tenantId" }),
          table = @Strategy(name = "realHisTable", fields = { "bizDate" }))
public class SqlToyTransInfo implements Serializable {
    @Id
    @Column(name = "ID", type = java.sql.Types.VARCHAR, length = 50, nullable = false)
    private String id;

    @Column(name = "TENANT_ID", type = java.sql.Types.VARCHAR, length = 50, nullable = false)
    private String tenantId;

    @Column(name = "BIZ_DATE", type = java.sql.Types.DATE, nullable = false)
    private LocalDate bizDate;
}
```

For table sharding only, drop `db`:

```
@Sharding(table = @Strategy(name = "realHisTable", fields = { "bizDate" }))
```

> Full attribute reference for `@Sharding` (including `maxConcurrents`, `maxWaitSeconds`, `is_global_rollback`) is in the Annotations Cheat Sheet · @Sharding.

4. Declaration style 2: XML (queries)

```
<sql id="sys_queryTransInfo">
    <!-- modulo on tenant id decides the datasource -->
    <sharding-datasource strategy="hashDataSource" params="tenantId" />
    <!-- the business date decides which table to query -->
    <sharding-table tables="SQLTOY_TRANS_INFO" strategy="realHisTable" params="bizDate" />
    <value><![CDATA[
        select * from SQLTOY_TRANS_INFO
        where tenant_id=:tenantId and biz_date<=:bizDate
    ]]></value>
</sql>
```

- `params` names the parameter(s) supplying the values, matching the strategy's `@Strategy.fields`;
- `tables` lists the physical tables this strategy may substitute for (matching the table in the original SQL).

5. Declaration style 3: code (QueryExecutor / EntityQuery)

```
lightDao.findByQuery(new QueryExecutor("""
        select * from SQLTOY_TRANS_INFO where tenant_id=:tenantId and biz_date<=:bizDate
        """)
        .values(MapKit.keys("tenantId", "bizDate").values("S001", LocalDate.now().minusDays(2)))
        .dbSharding("hashDataSource", "tenantId")
        .tableSharding("realHisTable", new String[] { "SQLTOY_TRANS_INFO" }, "bizDate"));
```

`EntityQuery` offers `dbSharding` / `tableSharding` too, for the single-table paths (`findEntity` / `updateByQuery` / `deleteByQuery`, etc.).

6. Parallel execution, global rollback and exceptions

- `@Sharding.maxConcurrents` (default 10) controls how many shards run in parallel: **0 means unlimited, 1 means serial** (6.0.3 fixed the case where 1 had no effect);
- `maxWaitSeconds` bounds the parallel wait;
- With `is_global_rollback` (default true) any failing shard rolls everything back;
- Since 6.0.3 `ShardingResult` carries a `cause` field preserving the failing shard's original exception, and the `DataAccessException` thrown by a global rollback includes the root-cause stack trace.

7. Known boundaries

- **Cross-database joins and pagination are not supported**: merging paginated results across databases and tables is disproportionately expensive and points at a design problem; use an MPP database such as ClickHouse/StarRocks instead (both supported by sqltoy) — see the FAQ;

- **Transactions after database sharding**: cross-database writes need JTA or a distributed-transaction framework; see Multi-Datasource & Transactions;
- `@DataVersion` **/cache translate on sharded tables**: table substitution happens at the SQL layer, so entity-level enhancements still apply; but after **database** sharding a single SQL only runs on one database, so aggregate statistics must be assembled by hand;
- **Strategy beans must implement** `initialize()` **and declare** `initMethod="initialize"` : the weighted probing of `DefaultShardingStrategy` and the modulus of `HashShardingStrategy` are set up there. Under native images, declare the **concrete return type** on the `@Bean` (an interface declaration leaves the init method unresolvable) — see the GraalVM Native Image Guide.

8. Related

- Datasource configuration and transactions: Multi-Datasource & Transactions
- Annotation attributes: Annotations Cheat Sheet
- Demo project: sqltoy-showcase (`DataSourceConfig` contains complete hash and date sharding setups)

TypeHandler Extension

Configure a handler class for special types such as json by extending sqltoy's abstract class: org.sagacity.sqltoy.plugins.TypeHandler

1. Configuration

```
# Spring Boot (spring.sqltoy prefix)
spring:
  sqltoy:
    typeHandler: com.example.sqltoy_solon_demo.handler.MyTypeHandler

# Solon (solon.sqltoy prefix)
solon:
  sqltoy:
    typeHandler: com.example.sqltoy_solon_demo.handler.MyTypeHandler
```

2. Implementation Class

Create a MyTypeHandler class that extends org.sagacity.sqltoy.plugins.TypeHandler

```
public class MyTypeHandler extends TypeHandler {
    @Override
    public boolean setValue(Integer integer, PreparedStatement ps, int parameterIndex, int i1, Object o) throws SQLException {
        if (parameterIndex == 1) {
            ps.setString(parameterIndex, "localhost");
            System.out.printf("setValue: %s ==> %s%n", o, "localhost");
            return true;
        }
        return false;
    }
}
```

3. JSONTypeHandler Example

- Note: since version 5.6.82 the framework provides built-in json/jsonb type support; upgrade quickvo to version 1.0.22, with jdbc-type=2016(json), 2017(jsonb)

3.1 Type Mapping for POJOs Generated by quickvo

```
<type-mapping>
              <!-- Keep one example; normally no configuration is needed -->
              <!-- Demo of adding the snowflake algorithm -->
              <sql-type native-types="BIGINT" jdbc-type="BIGINT" java-type="java.math.BigInteger" />

              <!-- Mind the xml escape characters for generics; table-field specifies the concrete table and field; jdbc-type can also be a plain
number, here java-type="List<StaffInfoVO>" -->
              <sql-type table-field="sqltoy_jsontype_showcae.staff_set" native-types="json" jdbc-type="2016" java-type="List&lt;StaffInfoVO&gt;"
              import-types="com.sqltoy.quickstart.vo.StaffInfoVO" />
</type-mapping>
```

3.2 Define JsonTypeHandler

```
public class JsonTypeHandler extends TypeHandler {
	// Handle null as appropriate; this step is mandatory on postgresql
	@Override
	public boolean setNull(Integer dbType, PreparedStatement pst, int paramIndex, int jdbcType) throws SQLException {
		return false;
	}

	/**
	 * <li>Returning true means the type matched and the setValue assignment has been done</li>
	 * <li>Returning false means a regular type, handed back to the framework for default processing</li>
	 */
	@Override
	public boolean setValue(Integer dbType, PreparedStatement pst, int paramIndex, int jdbcType, Object value)
			throws SQLException {
		// Distinguish specific types by setting the int number directly in jdbc-type in quickvo.xml
		if (jdbcType == 1021) {
			pst.setString(paramIndex, JSONObject.toJSONString(value));
			return true;
		}
		return false;
	}

	/*
	 * <li>1. Returning null means it is a regular type, handed back to the framework for processing</li>
	 * <li>2. Returning non-null means a special type; the type conversion is done and the result can be mapped directly to the VO property</li>
	 */
	@Override
	public Object toJavaType(String javaTypeName, Class genericType, Object jdbcValue) throws Exception {
		if (javaTypeName.equalsIgnoreCase("java.lang.string")) {
			return jdbcValue.toString();
		}
		// A VO object
		if (javaTypeName.contains("com.sqltoy")) {
			return JSON.parseObject(jdbcValue.toString(), Class.forName(javaTypeName));
		}
		if (javaTypeName.toLowerCase().contains("jsonobject")) {
			return JSON.parse(jdbcValue.toString());
		} else if (javaTypeName.toLowerCase().contains("jsonarray")) {
			return JSONArray.parseArray(jdbcValue.toString());
		}
		// List<VO> form
		if (javaTypeName.equalsIgnoreCase("java.util.List") && genericType != null) {
			return JSONArray.parseArray(jdbcValue.toString(), genericType);
		}
		// Any other scenario means it is not json; return null and let the framework handle it
		return null;
	}
}
```

Slow SQL Handling

sqltoy has a built-in mechanism for collecting and handling slow SQL (over-time SQL), making it easy to locate performance bottlenecks.

1. Default Configuration

The framework provides `DefaultOverTimeHandler` by default:

```
spring.sqltoy.overTimeSqlHandler=org.sagacity.sqltoy.plugins.overtime.DefaultOverTimeHandler
```

Together with the configuration parameter `printSqlTimeoutMillis` (8000 milliseconds by default), a warning log is printed when SQL execution exceeds that threshold.

2. Getting Slow SQL in Code

Get the slowest SQL statements through `SqlToyContext`:

```
// Get the 10 slowest SQL statements (the second argument, hasSqlId: whether to include only sql defined with an id in the xml)
List slowest = lightDao.getSqlToyContext().getSlowestSql(10, true);
```

3. Extending Slow SQL Collection

If you need custom collection logic (e.g. reporting to a monitoring system, writing to the database, integrating with an APM), implement the `OverTimeSqlHandler` interface yourself and configure:

```
spring.sqltoy.overTimeSqlHandler=com.yourpkg.YourOverTimeHandler
```

In the implementation class you get the over-time SQL statement, its parameters, its elapsed time and more, so you can persist it to the database, raise alerts or report it as needed.

> Related: for SQL execution statistics also refer to `org.sagacity.sqltoy.SqlExecuteStat`, `SqlExecuteTrace` and `SqlExecuteLog`. For common SQL optimization patterns see SQL Showcase.

NoSQL

MongoDB

Note: sqltoy's MongoDB integration **supports queries only**; for pure CRUD (insert/delete/update), use spring-data's `MongoTemplate` directly. sqltoy's motivation for integrating MongoDB is:

1. To fully leverage sqltoy's strength in **organizing dynamic conditions**;
2. Combined with sqltoy's **Cache Translate**, it eliminates the complexity of joining configuration collections in MongoDB (or the complexity of ETL cleansing);
3. It makes use of sqltoy's **result-set algorithms** (data masking, formatting, pivot, group summary, etc.) to simplify writing queries/aggregations directly;
4. It facilitates integration with external low-code frameworks (such as reporting platforms), enabling configuration-driven data query and processing.

> Demo project: https://github.com/sagframe/sqltoy-showcase

1. Configuration

sqltoy reuses spring-data-mongodb's `MongoTemplate`; just configure it in the standard Spring Boot way:

```
spring:
  data:
    mongodb:
      host: 192.168.56.101
      port: 27017
      database: sagframe
      username: sagframe
      password: sagframe
```

> **Spring Boot 4 migration note**: since Boot 4 the connection properties moved from `spring.data.mongodb.*` to `spring.mongodb.*` (e.g. `spring.mongodb.uri`); projects crossing versions should set both.

> Fully runnable example: `NoSqlTest` in `sqltoy-showcase` (a real MongoDB container via Testcontainers; the `<mql>` dynamic json query, `$in` array condition and `findPage` are verified end to end).

Direct object operations (`@Document(collection="")`, `@Id`, `@Field` annotations + `MongoTemplate`) belong to the spring-data domain and are not covered here; sqltoy mainly takes care of **querying + Cache Translate + result-set algorithms**.

2. Writing mql Queries

MongoDB queries are defined with the `<mql>` tag; the MongoDB JSON query/aggregation statements go into `value`, dynamic conditions are wrapped with `<#> ... </#>`, and parameters are referenced with `@(:paramName)`.

2.1 Basic Query (Field Alias + Cache Translate + Data Masking)

```
<!-- In fields, transType:transTypeName means the value of transType is also stored under the alias
     transTypeName, which facilitates the subsequent cache translate mapping -->
<mql id="sqltoy_mongo_find"
     fields="transId,merchantCode,transCode,transCode:transCodeName,transDate,agentOrg,transType,transType:transTypeName,transAmt"
     collection="fact_trans_details">
    <!-- Translate the alias columns; original-columns points to the columns that serve as the code source -->
    <!-- MongoDB cannot do "select trans_type as transTypeName" like SQL, so this is achieved via
         the fields alias + original-columns -->
    <translate cache="dictKeyName" cache-type="TRANS_CODE"
               columns="transCodeName" original-columns="transCode" />
    <translate cache="dictKeyName" cache-type="TRANS_TYPE"
               columns="transTypeName" original-columns="transType" />
    <!-- Secure data masking -->
    <secure-mask columns="merchantCode" type="bank-card" />
    <value><![CDATA[
        {
        <#>transType:@(:transType)</#>
        <#>@if(@(:flag)==1),transAmt:{$gt:@(:transAmt)}</#>
        <#>@else ,transAmt:{$gte:@(:transAmt)}</#>
        }
    ]]></value>
</mql>
```

Key points:

- **Field alias**: MongoDB cannot do `select trans_type as transTypeName` as SQL does; sqltoy uses the `source:alias` syntax in `fields` to copy the same field value into an alias column, then applies Cache Translate to the alias column via `<translate original-columns="sourceField" columns="aliasName">`, keeping the original code column unchanged.
- **Dynamic conditions**: when a parameter inside `<#>...</#>` is null, the whole segment is removed; it can be combined with `@if()/@else` for branching (see Tags & Expressions reference).
- **Data masking**: `<secure-mask>` is used the same way as in SQL queries.

2.2 Aggregation Query

```
<!-- In aggregation, fields can extract nested fields via _id.xxx:alias, e.g. when $group uses
     multiple fields: fields="_id[transCode,colName:aliasName],count,totalAmt" -->
<mql id="sqltoy_mongo_agg"
     fields="_id.transCode:transCode,count,totalAmt"
     collection="fact_trans_details">
    <translate cache="dictKeyName" cache-type="TRANS_CODE" columns="transCode" />
    <value><![CDATA[
        [
         { $group: {
              _id: {transCode:"$transCode"},
              count: { $sum: 1 },
              totalAmt:{$sum:"$transAmt"}
            }
         },
         { $match: {
              <#>transType:@(:transType)</#>
              <#>,transAmt:{$gt:@(:transAmt)}</#>
            }
         }
        ]
    ]]></value>
</mql>
```

The `value` is an aggregation pipeline array (`$group` / `$match` / `$sort`, etc.); `fields` uses `_id.transCode:transCode` to extract the nested `_id.transCode` from the aggregation result into a `transCode` column, on which Cache Translate is then applied.

2.3 mql Tag Attributes

Attribute	Required	Description
`id`	Yes	Unique identifier of the query
`collection`	Yes	MongoDB collection name
`fields`	Yes	Fields to extract/map; supports `source:alias`, `_id.xxx:alias`, `_id[a,b:alias]`, etc.
`aggregate`	No	Whether it is an aggregation query; defaults to `false`
`blank-to-null`	No	Convert blank parameters to null

Inside `<mql>`, sub-tags such as `filters`, `translate`, `secure-mask`, `pivot` / `unpivot` / `summary`, and `page-optimize` are also supported.

3. Cache Translate Definition

The `dictKeyName` cache in the example above is defined in `sqltoy-translate.xml` (with an inside-category `cache-type`, i.e. `DICT_TYPE`):

```
<sagacity xmlns="http://www.sagframe.com/schema/sqltoy-translate">
    <cache-translates>
        <!-- Build the cache from a SQL query; the first column of the query result is the category (DICT_TYPE) -->
        <sql-translate cache="dictKeyName" datasource="dataSource">
            <sql><![CDATA[
                select t.DICT_KEY,t.DICT_NAME,t.STATUS
                from SQLTOY_DICT_DETAIL t
                where t.DICT_TYPE=:dictType
                order by t.SHOW_INDEX
            ]]></sql>
        </sql-translate>
    </cache-translates>
    <cache-update-checkers>
        <!-- Incremental refresh check; has-inside-group=true means the first result column is the category -->
        <sql-increment-checker cache="dictKeyName" check-frequency="15"
                               has-inside-group="true" datasource="dataSource">
            <sql><![CDATA[
                select t.DICT_TYPE,t.DICT_KEY,t.DICT_NAME,t.STATUS
                from SQLTOY_DICT_DETAIL t
                where t.UPDATE_TIME >=:lastUpdateTime
            ]]></sql>
        </sql-increment-checker>
    </cache-update-checkers>
</sagacity>
```

For the full usage of Cache Translate, see Cache Translate usage.

4. Java Invocation

Invoke through the `lightDao.mongo()` chain (pass the mql id to `sql`):

```
// Query with parameters passed via an entity object
PospTransDetailVO queryVO = new PospTransDetailVO();
queryVO.setTransType("N");
queryVO.setTransAmt(0d);
List<PospTransDetailVO> result = (List<PospTransDetailVO>) lightDao.mongo()
        .sql("sqltoy_mongo_find")
        .resultType(PospTransDetailVO.class)
        .entity(queryVO)
        .find();

// Pass parameters via names/values + pagination
Page page = lightDao.mongo()
        .sql("sqltoy_mongo_find")
        .resultType(PospTransDetailVO.class)
        .names("transType", "transAmt", "flag").values("N", 13800, 2)
        .findPage(new Page<>());

// Aggregation query
List aggResult = lightDao.mongo()
        .sql("sqltoy_mongo_agg")
        .names("transType", "transAmt").values(null, null)
        .find();
```

The `mongo()` chain supports: `sql`, `names`, `values`, `entity`, `resultType`, `humpMapLabel`; terminal methods: `find()`, `getOne()`, `findTop(Float)` (a value greater than 1 takes a fixed row count, **less than or equal to 1 takes a proportion of the total records**), `findPage(Page)`.

5. Notes

- sqltoy **only performs queries** on MongoDB; use `MongoTemplate` for insert/delete/update (with the object annotations `@Document` / `@Id` / `@Field`).
- Combined with the `fields` alias + `original-columns`, Cache Translate resolves code-value translation without writing any join lookups.
- For aggregation results, extract nested fields via `fields` using `_id.xxx:alias` before translating/processing them.

Elasticsearch

sqltoy's motivation for integrating Elasticsearch (equally applicable to MongoDB):

1. To fully leverage sqltoy's strength in **organizing dynamic conditions**;
2. Combined with sqltoy's **Cache Translate**, it eliminates the complexity of joining configuration collections in ES (or the complexity of ETL cleansing);
3. It makes use of sqltoy's **result-set algorithms** (data masking, formatting, pivot, group summary, etc.) to simplify writing queries/aggregations directly;
4. It facilitates integration with external low-code frameworks (such as reporting platforms), enabling configuration-driven data query and processing.

> Demo project: https://github.com/sagframe/sqltoy-showcase

1. Configuring ES Nodes

Based on the spring-boot-starter, configure `spring.sqltoy.elastic` in `application.yml` (multiple nodes/clusters are supported):

```
spring:
  sqltoy:
    elastic:
      # Default endpoint id (the first one is used if not set)
      defaultId: default
      endpoints[0]:
        id: default
        # Multiple cluster nodes are comma-separated
        url: http://192.168.56.101:9200
        username: elastic
        password: skyline
        # SQL query path, see the notes below
        sqlPath: _nlpcn/sql
        # ===== The following are optional settings for https/certificate scenarios =====
        #keyStore: path to the certificate file
        #keyStorePass: certificate password
        #keyStoreType: jks          # defaults to jks; can be omitted
        #keyStoreSelfSign: true     # whether the certificate is self-signed, defaults to true
        #authCaching: true          # whether to disable preemptive authentication, defaults to true
```

Available values for `sqlPath` (the query endpoint in SQL mode):

sqlPath	Description
`_sql` (i.e. `_xpack/sql`)	ES native SQL, **now supports cursor-based pagination**
`_nlpcn/sql`	The elasticsearch-sql plugin (recommended, supports pagination)
`_opendistro/_sql`	The AWS OpenDistro SQL plugin

> ES native SQL did not support pagination in early versions but **now supports cursor-based pagination**, so `_sql` can be used directly. The elasticsearch-sql (`_nlpcn/sql`) and OpenDistro (`_opendistro/_sql`) plugins remain viable alternatives; note that elasticsearch-sql is no longer maintained after 7.9.3, in which case the AWS OpenDistro plugin can be used instead.

2. Writing eql Queries

ES queries are defined with the `<eql>` tag; two modes are supported: `mode="sql"` (SQL mode) and the native JSON REST mode (no `mode` attribute).

2.1 SQL Mode

```
<eql id="es_find_company" fields="company_id,company_name,company_type" mode="sql">
    <value><![CDATA[
        select * from cc_company_info
        where 1=1
           #[@if(:flag==1) and company_name=:companyName]
           #[@else  and company_type<>:companyType ]
    ]]></value>
</eql>

<!-- Pagination query (ES native SQL already supports cursor-based pagination;
     the elasticsearch-sql plugin also works) -->
<eql id="es_find_company_page" fields="company_id,company_name,company_type" mode="sql">
    <value><![CDATA[
        select * from cc_company_info
        where company_name like :companyName and company_type<>:companyType
    ]]></value>
</eql>
```

2.2 Native JSON REST Mode

Omitting `mode` gives the native JSON mode. Dynamic conditions are then wrapped with `<#> ... </#>` (instead of the `#[]` used in SQL mode), and parameters are referenced with `@(:paramName)`:

```
<!-- fields and the _source inside the query body are INTERCHANGEABLE:
     when neither is present, result mapping fails with
     "collection or property name array is empty" -->
<eql id="sys_elastic_test_json" fields="" index="cc_company_info">
    <!-- translate (Cache Translate) still works; column refers to the fields defined in _source -->
    <value><![CDATA[
        {
            "_source": ["company_id","company_name","company_type"],
            "query": {
                "bool": {
                    "filter": [
                        <#>{"terms":{"company_type":@(:companyTypes)}}</#>
                    ]
                }
            }
        }
    ]]></value>
</eql>
```

2.3 eql Tag Attributes

Attribute	Required	Description
`id`	Yes	Unique identifier of the query
`fields`	Conditionally	Property names to extract from the JSON finally returned by ES. **Either this or `_source` inside the query body must be present** (verified): when both are absent, mapping throws `collection or property name array is empty`
`mode`	No	`sql` (SQL mode) or `original` (native JSON); defaults to native JSON
`index`	No	Index name (commonly used in JSON mode)
`end-point`	No	The endpoint id defined in the configuration; defaults to the first one
`aggregate`	No	Whether it is an aggregation query; defaults to `false`
`blank-to-null`	No	Convert blank parameters to null; defaults to `true`
`value-root` / `type`	No	Root path / type of the JSON result

Inside `<eql>`, sub-tags such as `filters`, `translate` (Cache Translate), `secure-mask` (data masking), `pivot` / `unpivot` / `summary` (result-set algorithms), and `page-optimize` (pagination optimization) can also be used, exactly as in SQL queries.

3. Java Invocation

Invoke through the `lightDao.elastic()` chain (pass the eql id to `sql`):

```
// Normal query (SQL mode)
List<CompanyInfoVO> result = (List<CompanyInfoVO>) lightDao.elastic()
        .sql("es_find_company")
        .values(MapKit.keys("companyName", "companyType", "flag").values("Zhejiang Huaxu", "3", 2))
        .resultType(CompanyInfoVO.class)
        .find();

// Pagination query (ES native SQL already supports cursor-based pagination)
Page page = (Page) lightDao.elastic()
        .sql("es_find_company_page")
        .values(MapKit.keys("companyName", "companyType", "flag").values("Zhejiang Huaxu", "3", 2))
        .resultType(CompanyInfoVO.class)
        .findPage(new Page());

// Get Top N
List top = lightDao.elastic().sql("es_find_company_page")
        .values(...).resultType(CompanyInfoVO.class).findTop(10);

// Native JSON mode query (the parameter is an array)
List<CompanyInfoVO> jsonResult = (List<CompanyInfoVO>) lightDao.elastic()
        .sql("sys_elastic_test_json")
        .names("companyTypes").values(new Object[]{ new Object[]{ "1", "2" } })
        .resultType(CompanyInfoVO.class)
        .find();
```

The `elastic()` chain supports: `sql`, `names`, `values`, `entity`, `endPoint`, `resultType`, `humpMapLabel`; terminal methods: `find()`, `getOne()`, `findTop(int)` (fixed row count only, no proportional extraction), `findPage(Page)`.

4. Notes

- **Pagination**: ES native SQL already supports cursor-based pagination; the elasticsearch-sql / OpenDistro plugins support it as well — choose `sqlPath` as needed.
- Multiple clusters/nodes are configured via multiple `endpoints[n]` entries; at query time, specify one with the `end-point` attribute or `.endPoint(id)`.
- For pure CRUD, use the official ES client directly; sqltoy's value lies in the combination of **dynamic conditions + Cache Translate + result-set algorithms**.

5. Integration notes (verified)

1. **The endpoint `id` is mandatory**: omitting `endpoints[0].id` fails `SqlToyContext` initialization with an NPE on `getId()` — set an explicit id for every endpoint (or configure `default-id`).
2. **Dependency**: sqltoy's ES client (Rest5Client) is built on Apache HttpClient 5 — add `org.apache.httpcomponents.client5:httpclient5` explicitly (sqltoy does not carry it transitively).
3. **Provide either `fields` or `_source`**: see the attribute table in 2.3 — when both are absent, result mapping throws `collection or property name array is empty`.
4. **Pluggable JSON library**: the ES stack and JSON columns share the same JsonProvider abstraction — auto-detected from the classpath (fastjson2 -> jackson3 -> jackson2). A detected library must also pass a smoke parse before it is accepted; an unusable one (e.g. jackson3 meeting an older jackson-annotations) is logged as a warning and the next candidate is tried. You can also pin one via `spring.sqltoy.jsonProvider`, using a built-in alias (`fastjson2` / `jackson3` / `jackson2`) or the fully qualified class name of a custom implementation. When none is usable, startup only warns and the error appears only when a JSON/ES feature is actually used.

> Fully runnable example: `NoSqlTest` in `sqltoy-showcase` (a real ES container via Testcontainers; the `<eql>` json-mode dynamic query is verified end to end).

6. Secure connections (authentication and TLS)

Production ES clusters usually run with security enabled (auth + TLS). sqltoy endpoints support the full connection-security set:

```yaml
spring:
  sqltoy:
    elastic:
      endpoints[0]:
        id: prod
        url: https://es.prod.internal:9200
        # basic auth
        username: elastic
        password: ${ES_PASSWORD}
        # TLS: truststore for self-signed certificates
        key-store: classpath:certs/es-truststore.p12
        key-store-pass: ${ES_TRUSTSTORE_PASS}
        key-store-type: PKCS12
        key-store-self-sign: true
        # auth caching (on by default, avoids re-authenticating per request)
        auth-caching: true
```

> Full property list on `ElasticEndpoint` (id/url/username/password/keyStore*/requestTimeout, ...). The `NoSqlTest` demo intentionally runs a security-off container (`xpack.security.enabled=false`) for zero-friction testing; use the above for production.

Tools & Extensions

Utility Classes: DTO / POJO Conversion

In architectures where DTOs (VOs) and POJOs (entities) are strictly layered, sqltoy provides convenient, fast object/collection conversion and property copying, saving you from hand-writing large amounts of `get/set` copy code.

1. convertType (LightDao / SqlToyCRUDService)

`LightDao`, `SqlToyCRUDService` and `SqlToyDaoSupport` all provide `convertType` shortcut methods:

```
// Convert a single object
OrderInfo entity = lightDao.convertType(orderInfoVO, OrderInfo.class);

// Ignore some properties
OrderInfo entity2 = lightDao.convertType(orderInfoVO, OrderInfo.class, "createBy", "createTime");

// Convert a collection
List<OrderInfoVO> voList = lightDao.convertType(entityList, OrderInfoVO.class);

// Convert a Page object (pagination info is converted along with the data)
Page<OrderInfoVO> voPage = lightDao.convertType(entityPage, OrderInfoVO.class);
```

Signatures:

```
<T extends Serializable> T convertType(Serializable source, Class<T> resultType, String... ignoreProperties);
<T extends Serializable> List<T> convertType(List sourceList, Class<T> resultType, String... ignoreProperties);
<T extends Serializable> Page<T> convertType(Page sourcePage, Class<T> resultType, String... ignoreProperties);
```

Typical usage (VO→POJO when saving):

```
@Transactional
public void createOrderInfo(OrderInfoVO orderInfoVO) {
    OrderInfo entity = lightDao.convertType(orderInfoVO, OrderInfo.class);
    lightDao.save(entity);
}
```

> [!TIP] **Thousands-separator stripping** (since 6.0.3): when numeric strings with grouping commas produced by `number-format` (e.g. `"8,888.0000"`) are mapped to numeric properties such as `BigDecimal` / `Integer` / `Long` / `Double`, convertType strips the grouping commas automatically. Stripping happens only when the whole string strictly matches the grouping format (`"1,23"` and other invalid forms are left untouched and keep raising the original parse error).

2. MapperUtils (since 5.6.x)

From version 5.6.x onward you can also use the `MapperUtils` utility class for mapping, which suits object mapping outside of a DAO context.

3. Mismatched Field Names: @SqlToyFieldAlias

When the VO and the POJO use different field names, annotate the mapping with `@SqlToyFieldAlias`:

```
// postType in the VO corresponds to post in the POJO
@SqlToyFieldAlias("post")
private String postType;
```

This way `convertType` maps fields with different names correctly.

> For a complete example of strict layering, see the project https://github.com/sagframe/sqltoy-strict, where POJOs/VOs are generated into different packages by quickvo.

4. Common utility classes

Under `org.sagacity.sqltoy.utils` there is a set of helpers you would otherwise rewrite in every project (plus `org.sagacity.sqltoy.plugins.utils.CalculateUtils` for precision/rounding-aware arithmetic):

Class	What it gives you
`IdUtil`	Identifier generation: `getUUID()` (UUIDv7), `getNanoTimeId(workerId)` / `getShortNanoTimeId(...)` (26-digit ordered numbers, returning `BigDecimal`), `getId(distributeIdGenerator, tableName, signature, ...)` (business keys from a distributed generator), `getDebugId()`
`SnowflakeIdWorker` / `StandardUUIDv7Generator`	Standalone snowflake / UUIDv7 implementations you can hold as instances
`DBTransUtils`	Manual transactions: `doTrans(dataSource, transactionHandler)` wraps a block in a transaction, `getCurrentConnection()` returns the thread-bound connection
`StringUtil`	Blank checks, camelCase↔underscore conversion, symbol-aware splitting (brackets excluded), placeholder replacement, `splitExcludeSymMark`
`DateUtil`	Formatting/parsing, month-end/quarter-end, date stepping, `getIntervalDays` (used by sharding strategies)
`NumberUtil`	Number formatting (thousands separators, percentages), `toCapitalMoney` (Chinese uppercase amounts), parsing, `getProbabilityIndex(int[])` (weighted random index, used by weighted sharding), `getRandomNum`
`CalculateUtils`	Add/subtract/multiply/divide with explicit precision and `RoundingMode`
`CollectionUtil`	Collection splitting, grouping, empty checks, type conversion
`BeanUtil`	Property read/write, object↔Map conversion, enum value extraction, type conversion (the engine behind `convertType`)
`SqlUtil`	SQL fragment inspection and rewriting (strip order by, detect union/group, etc.) — handy in custom interceptors
`ResultUtils`	Result-set to Map/VO mapping and extended-type normalization (consult when taking over result processing)
`MapperUtils`	See section 2: VO↔POJO/Map mapping
`ReservedWordsUtil`	Quotes reserved words per dialect (used internally by `@joinFields`)

```
// Generate identifiers without a dao
String uuid = IdUtil.getUUID();                                 // UUIDv7
BigDecimal nanoId = IdUtil.getNanoTimeId("0");                  // 26-digit ordered number (workerId separates instances)

// Manual transaction (plain Java): run a block atomically
Object result = DBTransUtils.doTrans(lightDao.getDataSource(), () -> {
    lightDao.updateByQuery(StaffInfo.class,
            EntityUpdate.create().set("sexType", "F").where("staffId=?").values("S0001"));
    return lightDao.find("select * from staff_info where status=:status",
            MapKit.map("status", "1"), StaffInfo.class);
});
```

> These utilities are almost all **stateless static methods** (except instance-based ones such as `SnowflakeIdWorker`) and are safe to call from multiple threads; `DBTransUtils` transaction state is bound to the current thread. For more entry points see the API Cheat Sheet.

Table Metadata Operations (TableApi)

`lightDao.tableApi()` provides database table metadata queries and table-level operations (truncate/drop). It works with either table names or entity classes, and allows temporarily switching the data source.

1. API

```
TableApi tableApi = lightDao.tableApi();

// Get table info (catalog/schema/tableName support wildcards and may be null)
List<TableMeta> tables = tableApi.getTables(null, null, "sqltoy%");

// Get a table's column info
List<ColumnMeta> columns = tableApi.getTableColumns(null, null, "sqltoy_staff_info");

// Truncate a table (by table name or entity class)
tableApi.truncate("sqltoy_staff_info");
tableApi.truncate(StaffInfo.class);

// Drop a table (by table name or entity class)
tableApi.drop("sqltoy_staff_info");
tableApi.drop(StaffInfo.class);

// Temporarily switch the data source
tableApi.dataSource(otherDataSource).truncate("sqltoy_staff_info");
```

Method	Description
`getTables(catalog, schema, tableName)`	Returns `List<TableMeta>`; table names support wildcard matching
`getTableColumns(catalog, schema, tableName)`	Returns `List<ColumnMeta>`, including column name, type, length, comment, etc.
`truncate(Class entityClass)` / `truncate(String tableName)`	Truncates the table data
`drop(Class entityClass)` / `drop(String tableName)`	Drops the table
`dataSource(DataSource)`	Specifies the data source

2. Notes

- `LightDao` also offers the shortcut `truncate(Class entityClass)` method directly.
- `TableMeta` / `ColumnMeta` wrap database metadata and can be used for dynamic forms, code generation, data dictionaries, and similar scenarios.
- Different databases expose table information differently; some require a SQL query (e.g. Oracle's `select * from user_tab_comments`). In that case simply use sqltoy's SQL query directly; if you need to process the `ResultSet` yourself, write a custom Dao extending `SpringDaoSupport` and use the `DataSourceCallbackHandler` callback pattern.

> To generate table DDL from POJOs, see POJO to DDL.

Configurable Service Foundation (Reports / Configurable API Services)

> `XMLBinding` and `parseSqlSegment` registration are available since sqltoy 6.0.x.

sqltoy's **sqlId + named-parameter** model is a natural fit for configuration-driven queries: SQL fragments are stored as `<sql>` **in a database (or a configuration center / report model)** and parsed and executed by the framework at runtime. This capability is exactly the foundation for page-configured reports, configurable API services, and low-code platforms — the development side only maintains "configuration", and services consume it by id without repeated releases.

1. Overall Idea

Configuration side: every SQL configuration carries a **unique id** and a **last-modified time** (used for cache invalidation).

Execution side: two styles, chosen by how often the configuration changes:

Style	Use case	Notes
`XMLBinding` dynamic execution	Queries whose configuration changes frequently	Every request passes id + lastUpdateTime; the framework decides whether the parse cache is hit by comparing timestamps, re-parsing only when something changed
`parseSqlSegment` + `putSqlToyConfig` registration	Relatively stable queries	Parse and register once at startup / on change; afterwards `lightDao.find(id, params)` behaves exactly like a static sqlId

Dynamic columns: use `@joinFields(:fields, default)` to let configuration decide the `select` / `group by` / `order by` columns (field-name whitelist + dialect-aware reserved-word quoting), see Tags & Expressions.

2. Sample Configuration Storage (Database)

```
create table report_sql_config (
    report_id   varchar(64)  primary key,
    sql_xml     text         not null,   -- complete <sql>...</sql> fragment
    update_time datetime     not null
);
```

3. Style 1: Dynamic Execution via XMLBinding

```
@Service
public class ReportService {

    @Autowired
    private LightDao lightDao;

    /** Load the SQL configuration fragment from DB / config center */
    private String loadSqlXml(String reportId) {
        // e.g. select sql_xml from report_sql_config where report_id = ?
        // or read from a config center / report model
        return lightDao.getValue("select sql_xml from report_sql_config where report_id = :reportId",
                MapKit.keys("reportId").values(reportId), String.class);
    }

    public List<Map> query(String reportId, LocalDateTime updateTime,
                           Map<String, Object> params) {
        String sqlXml = loadSqlXml(reportId);
        return lightDao.findByQuery(
                new QueryExecutor(new XMLBinding(sqlXml)
                        .id(reportId)
                        .lastUpdateTime(updateTime))  // hits the parse cache when unchanged
                .values(params)
                .resultType(Map.class))
            .getRows();
    }
}
```

Cache rule (framework semantics):

- Without `lastUpdateTime`: the first parse is cached and reused afterwards;
- With `lastUpdateTime`: re-parse when the cache is empty, or the time is newer than the cached one, or the cached model has no timestamp; otherwise hit the cache directly with no re-parse.

4. Style 2: Parse and Register, Reuse by sqlId

```
public void registerReport(String reportId, String sqlXml) throws Exception {
    SqlToyConfig config = lightDao.getSqlToyContext().parseSqlSegment(sqlXml);
    // one report may contain several SQL fragments; use reportId + "_" + index to keep uniqueness
    config.setId(reportId);
    lightDao.getSqlToyContext().putSqlToyConfig(config);
}

public List<Map> queryById(String reportId, Map<String, Object> params) {
    // after registration this is identical to a static sqlId
    return lightDao.find(reportId, params);
}
```

> To unregister a no-longer-used id: `lightDao.getSqlToyContext().removeSql(reportId)` .

5. Dynamic Columns: Configuration Decides select / group by / order by

```
<sql id="report_dynamic_columns">
    <value><![CDATA[
        select @joinFields(:fields, t.order_id)
        from sqltoy_order_info t
        #[and t.trans_date >= :startDate]
        #[and t.trans_date < :endDate]
        order by @joinFields(:orderFields, t.trans_date)
    ]]></value>
</sql>
```

```
List rows = lightDao.findByQuery(
        new QueryExecutor("report_dynamic_columns")
            .names("fields", "orderFields", "startDate", "endDate")
            .values(
                "t.order_id,t.customer_name,t.total_amt",  // plain identifiers, comma separated
                "t.total_amt,t.trans_date",
                "2026-01-01", "2026-01-31")
            .resultType(Map.class))
        .getRows();
```

Notes:

- `@joinFields` accepts **plain identifiers only** (field whitelist: letters including Chinese, digits, underscore, at most three dotted segments); expressions such as `sum(t.total_amt)` or `t.*` must keep using `@value` (security is on you).
- When a field name may hit a reserved word (`order` / `desc` / `level` / `group` , etc.), configure `spring.sqltoy.reservedWords` so the framework quotes it per dialect (backticks for mysql, double quotes for oracle/pg, square brackets for sqlserver).
- Sort direction is best fixed in the SQL text; if configuration must decide the direction, concatenate with `@value` and take responsibility for safety.

6. Integration with Report / Low-Code Platforms

- **One report, several SQL fragments**: parse and register each, uniquify with `reportId + "_" + index` , then fetch in order;
- **Stackable capabilities**: the configured fragment may use `<translate>` cache translation, `<summary>` / `<pivot>` / `<link>` in-memory analytics tags, etc. — no extra code;
- **Pagination**: `findPage` / `Page` behave exactly like a static sqlId.

Related

- Dynamic SQL Guide (`#[]` dynamic segments, `@include` , `@joinFields` family)
- Tags & Expressions (`@joinFields` , `@value` , `@loop` / `@secure-loop` , etc.)
- Platform Integration (Reports / Low-Code)

Platform Integration (Reporting / Low-Code Platforms)

sqltoy is built not only for regular application development (admin backends, microservice APIs, etc.) but also for **reporting platforms** and today's popular **low-code platforms** (a new development model where traditional feature pages and API services are built and deployed through page-based configuration).

Examples built on sqltoy:

- **sagacity-rainbow**: a rapid API development framework;
- **sagacity-nebula**: a reporting framework.

1. Integration Approach

The core idea is to hand the SQL xml fragments over to `SqlToyContext` for management. Reporting/low-code platforms dynamically produce or load SQL configurations at runtime, while sqltoy takes care of parsing, parameter binding, dialect adaptation and execution.

2. Parameter Binding: getFullParamNames

A common integration question is: **how do I bind the (report/page) condition parameters to the condition parameters inside the SQL?**

For reports, condition parameters are usually organized as a Map. `SqlToyConfig` provides the `getFullParamNames` method, which lets you obtain **all parameter names** in a SQL statement so that an external condition Map can be aligned with the SQL parameters:

```
// Get all parameter names of the configuration for a given sqlId
SqlToyConfig config = lightDao.getSqlToyConfig("report_sql_id", SqlType.SELECT);
String[] paramNames = config.getFullParamNames();
// Use them to pick/assemble the actual parameters from the report's condition Map, then hand execution to sqltoy
```

3. Design Philosophy

> Sometimes a bit of flexibility goes a long way — just add an intermediate conversion step. As a foundational framework, sqltoy deliberately avoids flooding users with interfaces that would distract regular developers (offering "either this way or that way" only confuses them). Leave the dirty work — SQL hosting, parameter alignment, dialect adaptation — to the framework, and let the platform layer focus on business orchestration.

4. See Also

- For hands-on SQL-configuration-driven patterns (reports/API services), see Common SQL Examples.
- For an overview of configuration parameters, see sqltoy Configuration.

GraalVM Native Image Guide

Both the sqltoy spring-starter and solon-plugin ship with built-in AOT/native support: ORM reflection and resource metadata are registered automatically at build time, and resource loading switches to in-image resources at runtime. This page covers the full picture — what is supported, how to build, what the application must contribute, and the known limitations.

> **Version note**: AOT support was introduced in 5.6; **since 6.0.8** it is substantially enhanced (resources and extension classes are registered dynamically from the actual configuration, and entity scanning no longer depends on a build-time bean snapshot). This page describes 6.0.8; differences for 6.0.7 and earlier are called out per section.

1. What Is Built In

Module	AOT component	Registration	Purpose
`sagacity-sqltoy-spring-starter`	`SqltoyRuntimeHintsRegistrar`	auto-registered via `META-INF/spring/aot.factories`	fixed resource patterns (`sqltoy-translate.xml` , `translates/**` , `sqltoy/**`) + reflection for the config-property classes and the 6 built-in id generators (Default/UUID/ULID/NanoTime/Redis/Snowflake)
`sagacity-sqltoy-spring-starter`	`SqltoyBeanFactoryInitializationAotProcessor`	same	entity reflection plus configuration-driven resource patterns and extension-class reflection (since 6.0.8, see section 3)
`sagacity-sqltoy-solon-plugin`	`SqltoyRuntimeNativeRegistrar`	mounted automatically by `SqlToyPluginImpl` in AOT mode	symmetric with the Spring side (solon's AOT build runs the real startup flow, so configuration beans are genuinely available)

Native runtime adaptation (all versions): when sqltoy detects the `org.graalvm.nativeimage.imagecode` system property, it converts the `sqlResourcesDir` directory configuration into an explicit list of in-image `.sql.xml` resources (`SqlScriptLoader` works in resource-list mode); the solon side does the same for `translateConfig` .

> [!NOTE] One behavioral difference follows from this: sql files are baked into the image, so **debug-mode sql hot reload is automatically disabled under native** (file-change detection only works for File resources on a filesystem).

2. Build Steps (Spring Boot)

Prerequisites (identical to the official Spring Boot requirements — see the Spring Boot Native Image documentation):

- **GraalVM JDK 25+**: older versions are incompatible with the current GraalVM reachability metadata repository (verified matrix: **17 and 23** do not understand the new metadata schema — native-maven-plugin rejects them outright; **21** does not support `typeReached` conditional metadata — conditions count as always-true, spring-aop's AspectJ serialization entries get registered by mistake, and the build fails without aspectjweaver); 25.0.4 is verified with the full matrix on both stacks; on GitHub Actions `actions/setup-java` requires `distribution: graalvm` (`graalvm-community` was removed from setup-java v4's distribution list — using it fails the step outright), which still serves the official GraalVM Community builds;
- Visual Studio Build Tools (MSVC) on Windows, a gcc toolchain on Linux, Xcode Command Line Tools on macOS.

> [!NOTE] If the project uses lombok: JDK 23+ javac no longer runs classpath annotation processors implicitly — declare lombok explicitly in the maven-compiler-plugin's `annotationProcessorPaths` , otherwise entity getters/setters are not generated when compiling with GraalVM 25.

With `spring-boot-starter-parent` as the project parent, the `native` profile is built in:

```
# Build the native image (sqltoy's AOT metadata takes effect automatically)
mvn -Pnative native:compile -DskipTests

# Run it
./target/your-app
```

3. What the Application Must Contribute

1. packagesToScan is required (all versions)

sqltoy entities are **parsed lazily on first use**, while a native build can only register reflection metadata at build time. For native scenarios you must therefore configure `packagesToScan` (or `annotatedClasses`) so entities are registered at startup:

```
spring:
    sqltoy:
        # required for native: the build-time scan registers @Entity reflection metadata from this package
        packagesToScan: com.yourcorp.project.entity
        sqlResourcesDir: classpath:sqltoy
```

> On a regular JVM this property is optional (entities are parsed on demand); setting it only means eager parsing at startup, with no behavioral difference.

> `packagesToScan` only covers `@Entity`; plain DTO/VOs returned by queries are registered with `native-reflection-packages` / `native-reflection-classes` in section 3.1.

2. Resource and extension-class registration (automatic since 6.0.8)

Since 6.0.8, `SqltoyBeanFactoryInitializationAotProcessor` binds the `spring.sqltoy` configuration straight from the Environment (during AOT build processing the container does not instantiate/bind beans, so the old bean-snapshot implementation saw an empty configuration) and registers automatically:

Registered	Configuration source
Entity reflection	`packagesToScan` / `annotatedClasses` scan
Id-generator reflection	the 6 built-in generators are always registered (Default/UUID/ULID/NanoTime/Redis/Snowflake); custom generator classes referenced by `@Id(generator/strategy)` or `@BusinessId(generator)` on scanned entities are registered automatically
Nested config-property classes	`Elastic`, `ElasticConfig` (needed by solon's runtime `cfg().toBean(...)` binding for nested endpoints)
Sql file resource patterns	`sqlResourcesDir` (multi-directory splitting), `sqlResources`
Translate config resources	`translateConfig` (`.xml` as files, directories as `/**`)
Key resources	`securePublicKey` / `securePrivateKey`
Extension-class reflection	`unifyFieldsHandler`, `typeHandler`, `sqlInterceptors`, `sqlFormater` (including `default` alias expansion), `translateCacheManager`, `dataSourceSelector`, `connectionFactory`, `fieldsSecureProvider`, `desensitizeProvider`, `customFilterHandler`, `overTimeSqlHandler`, `dialectDDLGenerator`, `firstBizCodeTrace`, `dynamicCacheFetch`, `dynamicFecthCacheManager` (same convention as runtime: a value containing `.` is a class name, otherwise it is a bean name covered by Spring AOT itself), and `functionConverts` (including the `default`-expanded built-in function set, resolved via `FunctionUtils.resolveFunctionClassNames` with exactly the runtime semantics)
Integration-class reflection (conditional)	registration follows the classpath, not the build-time configuration: caffeine library present → the translate cache manager (the ehcache manager is instantiated directly at runtime and needs no registration); mongo driver present → `SpringMongoQuery`; spring-data-redis present → `SpringRedisIdGenerator`
Extension-type driver classes (conditional)	registered when the corresponding driver is on the classpath: the 5 isomorphic `PGobject` classes of the PG family (postgresql plus the stardb-style fallback, opengauss/mogdb, vastbase, gaussdb, and kingbase's `KBobject` — json/jsonb/vector/geometry are constructed and read reflectively), mssql-jdbc's `Geometry` / `Vector`, and oracle's `OracleTypes`
JsonProvider reflection	**all** built-in implementations whose JSON library is on the classpath (fastjson2/jackson3/jackson2, by their markers) plus an explicitly configured custom implementation class name
Lambda serialization base	java-serialization registration for `SerializedLambda` and sqltoy's mirror class (fixed; the app side must additionally register lambda defining classes, see next section)

In other words: **with entities, resources and extensions configured by convention, a native build needs no extra AOT code since 6.0.8**; non-entity DTO/VOs are registered with `native-reflection-classes` / `native-reflection-packages` in the next section.

> [!WARNING] The built-in implementation behind `sqlFormater` (`DefaultSqlFormater`) hard-depends on druid (optional in core): applications configuring `sqlFormater` must add druid themselves, otherwise the debug-statistics path (`SqlExecuteStat`) throws `NoClassDefFoundError` when it formats SQL (not triggered while debug is off).

> [!NOTE] **Runtime switching**: a native image is frozen at build time, so integration classes and JsonProviders are registered by **classpath** (not by the build-time values of `cacheType` / `jsonProvider`) — as long as the corresponding library is on the classpath, switching `cacheType` (ehcache↔caffeine) or `jsonProvider` (fastjson2/jackson3/jackson2/custom) via startup arguments works at runtime; sqltoy-aot-verify runs dedicated switch rounds on both stacks.

3. What remains application-side (any version)

Five categories the framework cannot infer — complete runnable examples: sqltoy-aot-verify (`AotVerifyRuntimeHints` / solon `Config#aotVerifyNativeRegistrar`) and the showcase's `ShowcaseRuntimeHints`:

1. **classpath resources loaded by application code itself**: e.g. startup schema scripts (the showcase's `mock/schema-*.sql`, read by `DataSourceConfig`);
2. **classes referenced BY NAME from SQL/third-party mechanisms**: e.g. H2's `CREATE ALIAS ... FOR 'com.x.Procedures.listStaff'` (loaded via `Class.forName` by H2 at runtime);
3. `@Bean` **methods declared with an interface return type that carry** `initMethod` **/transactional proxies**: Spring AOT pre-determines bean types from the declaration, and an interface declaration hides the concrete class (the native runtime then fails with a missing init method or "CGLIB runtime enhancement not supported") — **declare the** `@Bean` **with the concrete return type** and no hints are needed (the showcase's sharding-strategy beans and the 6.0.8 starter's DAO beans are declared this way);
4. **DTO/VO classes used as query resultType**: not `@Entity` (so `packagesToScan` does not see them) yet instantiated reflectively by sqltoy at runtime — since 6.0.8 use `native-reflection-classes` / `native-reflection-packages` (see 3.1) instead of writing `RuntimeHints` by hand;
5. **classes that use lambda conditions (lambdaQuery/lambdaUpdate/lambdaDelete)**: property extraction goes through writeReplace/serialization mirroring, which in a native image requires registration by "lambda defining class" — on spring declare `lambdaCapturingTypes` in `META-INF/native-image/**/serialization-config.json` (or `hints.reflection().registerLambda(definingClass, b -> b.withInterfaces(LambdaFn.class))`), on solon `metadata.registerLambdaSerialization(definingClass)`; without it the whole lambda system is unavailable under native (JVM mode is unaffected).

```
public class ShowcaseRuntimeHints implements RuntimeHintsRegistrar {
    @Override
    public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
        // startup schema scripts loaded by app code (the RSA keys are registered
        // by the starter from the secure* properties)
        hints.resources().registerPattern("mock/**");
        // referenced by name from H2's CREATE ALIAS, loaded via Class.forName at runtime
        hints.reflection().registerType(Procedures.class, MemberCategory.values());
    }
}

// mounted on the application class
@SpringBootApplication
@ImportRuntimeHints(ShowcaseRuntimeHints.class)
public class ShowcaseApplication { ... }
```

> [!TIP] Keeping sql files under `classpath:sqltoy/` (the solon default convention) hits the built-in fixed patterns — no resource registration code needed on any version.

3.1 Registering DTO/VO reflection (since 6.0.8)

`packagesToScan` only recognizes `@Entity`; plain resultType DTO/VOs are not covered. Since 6.0.8 both starters expose two build-time properties, and the AOT processor registers the reflection metadata automatically — no hand-written `RuntimeHints` needed:

Property	Value	Use case
`native-reflection-classes`	list of fully-qualified class names	a few special classes, classes referenced by name, individual DTO/VOs
`native-reflection-packages`	list of package paths	many DTO/VOs, register them by package

```
# Spring Boot
spring:
  sqltoy:
    # recursively register every class under the package(s)
    native-reflection-packages:
      - com.yourcorp.project.dto
      - com.yourcorp.project.vo
    # add special classes individually
    native-reflection-classes:
      - com.yourcorp.project.report.TreeModel
```

```
# Solon
solon:
  sqltoy:
    native-reflection-packages:
      - com.yourcorp.project.dto
      - com.yourcorp.project.vo
    native-reflection-classes:
      - com.yourcorp.project.report.TreeModel
```

Rules:

- `native-reflection-packages` recursively scans all `.class` files under each package and registers `MemberCategory.values()`; plain DTO/VOs, records, enums and entity classes are all registered.
- Interfaces, annotations, anonymous/local/synthetic classes and `package-info` are skipped; a class that fails to load is logged and skipped without failing the AOT build.
- Both properties may be used together; duplicate registrations are harmless.
- Package scanning registers every class under the package, which slightly increases image size — keep DTO/VO packages free of unrelated classes rather than adding broad application packages.
- These properties are build-time only and do not affect JVM mode; they do not replace `packagesToScan` — `@Entity` classes still need `packagesToScan` / `annotatedClasses`.

> [!NOTE] Verified in real native builds of `sqltoy-aot-verify` on both Spring and Solon: with the application-side DTO reflection code removed and only `native-reflection-packages` configured, the B1 DTO translate query still passes; that round did not configure `native-entity-proxies`, so both native smoke runs report **16 PASS + 0 FAIL + 1 KNOWN-LIMITATION** (with proxies configured, see 3.2: 17 PASS + 0 KNOWN). The generated `reachability-metadata.json` / `reflect-config.json` contains the DTO types.

3.2 updateSaveFetch entity proxies (`native-entity-proxies`)

The "row exists" branch of `updateSaveFetch(entity, EntityUpdateCallback, ...)` needs a proxy that intercepts entity getters/setters: getters read the current `ResultSet` row and setters write to it. On the JVM this subclass is generated at runtime by ByteBuddy; a native image is closed-world and **cannot define new classes at runtime**, so that branch is unavailable by default.

Since 6.0.8, `native-entity-proxies` makes the AOT build pre-generate proxy subclasses (including ByteBuddy auxiliary classes) only for the listed entities, writes them into the image and registers their reflection metadata. At runtime `EntityResultSetProxy` first loads `<entity class name>$$SqltoyEntityProxy`; only if it is missing does it fall back to JVM dynamic generation:

```
# Spring Boot: list only entities that actually use EntityUpdateCallback
spring:
  sqltoy:
    native-entity-proxies:
      - com.yourcorp.project.entity.StockVO
      - com.yourcorp.project.entity.AccountVO
```

```
# Solon
solon:
  sqltoy:
    native-entity-proxies:
      - com.yourcorp.project.entity.StockVO
      - com.yourcorp.project.entity.AccountVO
```

Constraints:

- The entity must be subclassable: not `final` and with an accessible no-arg constructor; otherwise AOT logs a warning and the entity still fails at runtime with "not pre-generated".
- `byte-buddy` is optional in the starter/plugin and is not transitive; applications using `native-entity-proxies` must add `net.bytebuddy:byte-buddy` explicitly. Without it, the AOT build logs a clear warning and skips proxy generation; the entity remains unavailable at runtime.
- Only the listed entities are covered; unlisted entities keep the "unavailable under native" limitation and never silently fall back to runtime generation.
- Both the proxy and its auxiliary classes are added to the image; more entities slightly increase image size, so do not list every entity indiscriminately.

> [!NOTE] Verified in real native builds of `sqltoy-aot-verify` on both stacks: with `native-entity-proxies: com.sqltoy.aotverify.entity.AotUsfVO`, C1 (`updateSaveFetch` with `EntityUpdateCallback`) changes from KNOWN-LIMITATION to **PASS**, and both smoke runs report **17 PASS + 0 FAIL + 0 KNOWN-LIMITATION**.

> **Historical note (6.0.7 and earlier)**: the starter's AOT entity registration did not work under Spring AOT processing (configuration beans are not bound at build time, so the entity snapshot came out empty), and the built-in fixed patterns only span the `sqltoy/**`, `translates/**`, `sqltoy-translate.xml` conventions. Applications had to cover five things themselves: ① resource patterns for custom sql directories; ② reflection for class-name configured extensions; ③ entity reflection (scanning the same package as `packagesToScan` for `@Entity`); ④ reflection for sqltoy's 8 XML parsing model classes (without them the attributes of `<unpivot>` / `<tree-sort>` /chain-relative tags are **silently dropped** and parsing dies with an NPE); ⑤ re-declaring the DAO beans with concrete return types. The full 6.0.7-era recipe is in the showcase repository history.

4. Known Limitations

Limitation	Cause	Scope
`updateSaveFetch` (`EntityUpdateCallback`) is unavailable by default	`EntityResultSetProxy` generates entity subclasses at runtime via byte-buddy; native image does not support runtime class generation (observed failure: `IllegalArgumentException: Could not create type`)	that feature only; **configuring `native-entity-proxies` for the entity pre-generates the proxy at build time and makes C1 PASS (verified)**; unlisted entities remain unavailable; regular CRUD/queries and the insert branch are unaffected
Sql hot reload is disabled	sql files are baked into the image and loaded as a resource list — there is no File to watch	debug hot reload only; use JVM mode during development
The startup warning "no entity classes ... matched in the sqltoy pojo scan path" is expected	`EntityScanner` enumerates directories via the file/jar protocol, which is unavailable inside the image, so the `packagesToScan` startup scan comes out empty under native	no functional impact: entity reflection metadata is registered at build time and entities are parsed lazily on first use at runtime (CRUD + translate cache verified by the native smoke run)
Third-party components rely on their own metadata	native support for mongo/ES/redis/ehcache clients comes from their official reachability metadata	check the respective library's documentation when using them. Known example: **fastjson2**'s bundled native-image.properties misses nested classes (`JSONFactory$CacheItem` etc.), which fails GraalVM 25's image-heap validation — add the build argument `--initialize-at-build-time=com.alibaba.fastjson2` (already wired in sqltoy-aot-verify); even then its `JDKUtils` Unsafe field-offset recomputation fails at build time, so runtime detection **silently skips fastjson2 and degrades to jackson3/jackson2** by design (functionality unaffected) — prefer jackson2/jackson3 for JSON columns under native (both verified)
Business types serialized to JSON	the JsonProvider implementation is registered automatically, but reachability of serialized business types depends on the JSON library's own native support	JSON columns / ES scenarios; Spring Boot has built-in Jackson support

5. Solon Integration

The solon-plugin ships `SqltoyRuntimeNativeRegistrar` (implementing solon's `RuntimeNativeRegistrar` interface, mounted automatically by `SqlToyPluginImpl` in AOT mode). Its registrations are symmetric with the Spring side: fixed resource patterns, reflection for config-property classes and the 6 built-in id generators, entity reflection — and since 6.0.8, configuration-driven resource patterns, extension-class reflection, and the DTO/VO and special classes listed by `native-reflection-classes` / `native-reflection-packages`, plus `native-entity-proxies` for pre-generated updateSaveFetch proxies. Solon's default `sqlResourcesDir=classpath:sqltoy` hits the built-in patterns, so it works with zero configuration. For the build flow see the official solon native documentation and the Solon Integration Reference.

> [!NOTE] When solon binds a yml **scalar** to a `String[]`, the elements come out quoted (`packagesToScan: com.x.entity` binds as `"com.x.entity"`), and a quoted package name makes the entity scan come out silently empty. Since 6.0.8 the plugin strips surrounding quotes, so both the scalar and the yaml-list forms work; on 6.0.7 and earlier use the yaml-list form.

6. Demo & Verification

Dedicated verification project sqltoy-aot-verify (spring + solon stacks sharing one verification matrix, PostgreSQL pgvector+postgis): the matrix covers CRUD × json/vector/geometry in three shapes (String / float[] / JTS Geometry), cache translate on both ehcache and caffeine, the updateSaveFetch proxy path, and lambda single-table/join/setExpr/matchAll-guard/delete. JVM full matrix: 17/17 PASS on both stacks; with `native-entity-proxies` configured, native smoke: **17 PASS + 0 KNOWN-LIMITATION** on both stacks (C1 resolved by the pre-generated proxy).

The demo project sqltoy-showcase provides two verification layers:

1. **JVM level (runs as part of `mvn test`, no GraalVM needed)**: `NativeAotTest` runs the exact same Spring Boot AOT processing that `native:compile` consumes (`SpringApplicationAotProcessor`), parses the generated `reachability-metadata.json`, and asserts that entity reflection, the sql/mock resource patterns and the extension-class reflection are all in place;
2. **Real compilation (CI)**: the native-build workflow runs `mvn -Pnative native:compile` on ubuntu + GraalVM and smoke-runs the binary with `--smoke` (NativeSmokeRunner: entity load, SQL count, translate cache; manual dispatch + weekly, kept out of the regular PR CI).

> 🎬 Full example source: `NativeAotTest.java` and `ShowcaseRuntimeHints.java` in the sqltoy-showcase demo project

Related

- Other Features
- Solon Integration Reference
- Configuration Reference

Other Features

This page summarizes a few less frequently used capabilities that are quite valuable in specific scenarios.

1. GraalVM Native Image (AOT Support)

`sagacity-sqltoy-spring-starter` ships with built-in Spring AOT support (`SqltoyRuntimeHintsRegistrar` and `SqltoyBeanFactoryInitializationAotProcessor` , auto-registered via `META-INF/spring/aot.factories`), mirrored by `SqltoyRuntimeNativeRegistrar` in `sagacity-sqltoy-solon-plugin` . Build the native image the standard Spring Boot way; since 6.0.8 resource patterns and extension-class reflection are registered dynamically from the actual configuration. In native scenarios entities still need `packagesToScan` , while non-entity DTO/VOs can be registered with `native-reflection-packages` (by package) or `native-reflection-classes` (by class name):

```
# Build the native image the standard Spring Boot way; sqltoy's hints take effect automatically
mvn -Pnative native:compile
```

> For the full guide (build prerequisites, what the application must contribute, known limitations, demo project), see the GraalVM Native Image Guide.

2. geometry Spatial Type Support (JTS)

sqltoy has built-in automatic adaptation for database **geometry spatial columns** (GIS scenarios); the core logic lives in `GeometryTypeUtil` :

- Automatically recognizes geometry column types and handles conversions between **MySQL internal bytes ↔ WKT** and **Oracle SDO ↔ WKT**;
- When **JTS** (`org.locationtech.jts` , auto-detected by `hasJts()`) is on the classpath, query results are automatically parsed into **JTS** `Geometry` **objects**, and writing `Geometry` objects is supported as well;
- Without JTS, interaction happens through **WKT strings** (e.g. `POINT(121.4 31.2)`).

```
<!-- Add the JTS dependency when interacting with Geometry objects -->
<dependency>
    <groupId>org.locationtech.jts</groupId>
    <artifactId>jts-core</artifactId>
    <version>1.20.0</version>
</dependency>
```

> Simply declare the POJO's geometry property as a JTS `Geometry` (or a WKT string); reading and writing is encoded/decoded automatically by the framework (`JtsGeometryCodec` / `GeometryTypeUtil`).

3. Common-Field Auto-Fill Switch (UnifyUpdateFieldsController)

Once a unified fields handler (unifyFieldsHandler) is configured, save/update automatically fills in common fields such as `updateBy` / `updateTime` . In certain scenarios (e.g. system-level data repair scripts where you don't want to alter "last updated by"), you can temporarily **pause** common-field auto-fill for the current thread:

```
try {
    // Pause automatic common-field filling for the current thread (a ThreadLocal switch; only affects the current thread)
    UnifyUpdateFieldsController.stop();
    lightDao.update(fixEntity);   // this update will not auto-fill updateBy/updateTime
} finally {
    // Must resume, otherwise no later operations on this thread will auto-fill
    UnifyUpdateFieldsController.resume();
}
```

Method	Description
`stop()`	Pauses common-field auto-fill for the current thread
`resume()`	Resumes common-field auto-fill for the current thread
`useUnifyFields()`	Checks whether the current thread has it enabled

4. Custom Connection Operations (DataSourceCallbackHandler)

When you need raw JDBC capabilities not wrapped by the framework, write a custom Dao extending `SpringDaoSupport` (or `SqlToyDaoSupport`) and use `DataSourceUtils.processDataSource` together with an anonymous `DataSourceCallbackHandler` to obtain the raw `Connection` and handle it yourself (the framework takes care of acquiring and releasing the connection):

```
public class MySpecialDao extends SpringDaoSupport {

    // Follow the framework's internal usage pattern
    public Object doSpecial(final DataSource dataSource) {
        DataSourceUtils.processDataSource(getSqlToyContext(),
                getDataSource(dataSource),
                new DataSourceCallbackHandler() {
                    @Override
                    public void doConnection(Connection conn, DBProfile profile) throws Exception {
                        // profile is the database execution profile: getDbType()/getDialect()/getProductName()/getMajorVersion()
                        // plus convenient checks such as isMysqlFamily()/isOracleFamily()/isOceanBase()
                        try (PreparedStatement pst = conn.prepareStatement("select ...")) {
                            ResultSet rs = pst.executeQuery();
                            // ... process rs yourself
                            setResult(result);   // put the result back so it can be fetched via getResult()
                        }
                    }
                });
        return getResult();
    }
}
```

Member	Description
`doConnection(conn, profile)`	Abstract method; you receive the raw `Connection` plus the `DBProfile` database execution profile and handle it yourself
`DBProfile`	Provides `getDbType()` / `getDialect()` / `getUrl()` / `getProductName()` / `getMajorVersion()`, plus convenient checks such as `isMysqlFamily()` / `isOracleFamily()` / `isOceanBase()` / `isBackslashEscape()`
`setResult(...)` / `getResult()`	Stash the result inside the callback / fetch it from outside

> **Version note**: starting with 6.0 the callback signature changed from `doConnection(Connection conn, Integer dbType, String dialect)` to `doConnection(Connection conn, DBProfile profile)` (`org.sagacity.sqltoy.model.DBProfile`); on 5.6.x please use the old signature.

Reference

Tags & Expressions Reference

In addition to `#[]` dynamic condition fragments, sqltoy reserves a set of `@` macro directives inside SQL for highly complex dynamic SQL scenarios. Combining `@if`, `@value` and `@blank` flexibly solves the vast majority of problems; for more complex cases, the recommended approach is to assemble the SQL fragment in code and embed it dynamically with `@include(:sqlScript)` (the fragment may keep using named parameters); for plain-text injection, `@value(:sqlScript)` also works.

1. @if() / @elseif() / @else

Conditional logic reserved for special business scenarios; it can be written directly in the SQL.

Two judging modes:

```
Single condition: @if(:paramName>=value)
Dual condition: @if(:paramName1>=value1 && :paramName2<=value2)
```

- Comparison operators: `>`, `>=`, `==`, `<`, `<=`, `!=`, `<>`
- Logical operators: `&&`, `||`, `include`, `in`, `out`
- Combine with `@elseif()` and `@else` to form complete branches.

Time/date comparison:

```
@if(:paramName>=now()+x)   or   @if(:paramName>=now()-x)
```

- Time expressions: `now()`, `nowtime()`, `${.now}`
- Date expressions: `day()`, `sysdate()`, `${.day}`

Collection/containment checks:

Syntax	Meaning
`@if(:statusAry include 1)`	When statusAry is an array, checks whether it contains 1; when it is a string, equivalent to `statusAry.contains("1")`
`@if(:statusAry exclude 1)`	The opposite of include — does not contain
`@if(size(:statusAry)>1)`	Gets the length of a collection/array via `size(:paramName)`
`@if(:status in '1,2,3')`	Returns true when status is one of 1/2/3
`@if(:status out '1,2,3')`	Returns true when status is none of 1/2/3

2. @fast / @fastPage

`@fast()` is written inside the SQL for fast pagination (paginate first, then join). **For non-paged queries the framework automatically strips `@fast`**, so it does not affect find/findTop or other queries. `@fast` and `@fastPage` are equivalent.

3. @blank()

Invents a parameter for statements that are not conditions themselves, so they fit sqltoy's `#[]` organization rules.

```
#[@blank(:sexType) t.SEX_TYPE]
```

- When `sexType` is null, the whole fragment is removed following the `#[]` logic;
- When `sexType` is not null, `@blank(:sexType)` automatically becomes blank, leaving `t.SEX_TYPE`, without affecting SQL execution.

`@blank` is not mandatory; it is commonly used to absorb other parts of the statement such as `and` and parentheses. For complex scenarios, combine it with `@if` and `@value`.

4. @value()

`@value(:paramName)` is similar to `@blank(:paramName)`, with the only difference that it **directly renders the value of paramName**.

Typical usage: `@value(:sqlScript)` embeds a SQL fragment assembled in code, handling highly complex dynamic SQL; if the fragment itself needs named parameters, use `@include(:scriptParam)` (section 7) instead.

> **Note**: `@value` injects text without validation. When all you need is to join a **list of field names** (page-configurable reports/API services where the configuration decides the select, group by and order by columns), use `@joinFields()` in section 8 instead — it validates field names against a whitelist and quotes reserved words per dialect.

5. @loop() / @secure-loop() / @loop-full() / @secure-loop-full()

Used to dynamically loop-concatenate fields or conditions in SQL, generally over arrays or collections; **not intended for in scenarios** (just use `in (:param)` directly for those).

- `@loop()` : parameters are concatenated directly into the SQL (**not recommended** — injection risk).
- `@secure-loop()` : parameters are passed as `?` placeholders, preventing SQL injection (**recommended**).
- The `-full` variants: by default `@secure-loop` skips null and empty values in the loop; if you don't want them skipped, use `@secure-loop-full`.

Three formats:

```
@secure-loop(:loopParam, loopContent)
@secure-loop(:loopParam, loopContent, linkSign)
@secure-loop(:loopParam, loopContent, linkSign, startIndex, endIndex)
```

Parameters: `startIndex` / `endIndex` are an optional loop index range (0-based; `endIndex` beyond the array length is clamped to the end), for looping over only a slice of the collection — e.g. skipping the first element and looping from the second to the end. **Write the indexes as bare numbers, without quotes:**

```
@secure-loop(:staffIds, { t.STAFF_ID=:staffIds[i] }, { or }, 1, 100)
```

Notes:

- Inner parameters are comma-separated; if `loopContent` contains commas, wrap the content with `{}`, `''` or `""` to ensure it is split correctly.
- Sub-object property patterns such as `staffInfos[i].xxxx` are supported for extreme scenarios.
- When `loopParam` is empty or absent the macro returns `@blank(:loopParam)`, which makes the enclosing `#[]` fragment drop out entirely — no extra null-check needed.
- When an in-condition parameter array exceeds 1000 entries (e.g. the Oracle limit), the framework (since 5.1.32 / 4.19.25) **splits it automatically**.
- `@secure-loop` is supported since version 5.1.44.

Examples:

```
-- 1) OR condition chain (VALUE conditions use @secure-loop, bound as ? placeholders):
--    with staffIds = ["S0001","S0003"] it expands to:  and ( t.STAFF_ID=?  or  t.STAFF_ID=? )
#[and (@secure-loop(:staffIds, { t.STAFF_ID=:staffIds[i] }, { or }))]

-- 2) Dynamic columns (column names are IDENTIFIERS — the placeholder mechanism does not
--    apply, so @loop concatenation is the only option here):
--    with fields = ["STAFF_NAME","EMAIL"] it expands to:  select t.STAFF_ID ,t.STAFF_NAME ,t.EMAIL from ...
select t.STAFF_ID @loop(:fields, { ,t.:fields[i] }) from sqltoy_staff_info t

-- 3) Index slice (bare numbers, no quotes): loop from the second element (skipping the first),
--    endIndex beyond the array is clamped
@secure-loop(:staffIds, { t.STAFF_ID=:staffIds[i] }, { or }, 1, 100)
```

> Prefer `@secure-loop` whenever the loop involves **values** (injection-safe); reserve `@loop` for identifier concatenation such as column names.

6. @include(sqlId)

`@include(sqlId)` concatenates SQL: the SQL of the corresponding `sqlId` is substituted at the `@include` position. Multiple includes are allowed and can be embedded anywhere.

> The feature is provided but **not recommended**: it hurts readability and creates coupled side effects.

7. @include(:scriptParam)

A variant of `@include(sqlId)` that passes the SQL fragment in as a **dynamic parameter**:

```
String sql = "select * from sqltoy_fruit_order where status=:status @include(:sqlScript)";
List result = lightDao.find(sql,
        MapKit.keys("status", "saleCount", "sqlScript")
              .values(1, 12, "and sale_count>:saleCount"));
```

8. @joinFields() (new in 6.0.7)

Joins a runtime-supplied **list of field names** into a SQL identifier list — for page-configurable reports and API services where the configuration decides the `select` / `group by` / `order by` columns.

Formats:

```
@joinFields(:fields)
@joinFields(:fields, default)
@joinFields(?, default)
```

- Processed at the same stage as `@value` (after named parameters have become `?`, resolved positionally), so `@joinFields(:fields,...)` and writing `@joinFields(?,...)` directly in the SQL are equivalent; the consumed parameter is removed from the parameter set, so the remaining `?` placeholders stay aligned.
- When `fields` is null, empty or all-blank, `default` is emitted; without a `default` the result is an empty string.
- `default` comes from the SQL text (author-written, trusted) and is emitted verbatim without identifier validation, so it may be an expression such as `t.*` or `sum(t.amount)`; wrap it with `{}`, `''` or `""` when it contains commas or parentheses, e.g. `@joinFields(:fields,{a.id,a.code})`.

Input normalization:

Input shape	Handling
a single string `"a,b,c"`	split on commas into 3 fields
a 1-element array `["a,b,c"]`	same (page multi-selects often serialize this way)
an array with more than 1 element `["a","b"]`	elements are the fields, **no further splitting**
more than 1 element where one contains a comma `["a,b","c"]`	rejected, never silently split
blank fragments (e.g. a trailing comma `"a,b,"`)	skipped

Security: field names go through a character whitelist — letters (including Chinese), digits, underscore, and at most two dots for qualification (`schema.table.column`). `$` and `#` are explicitly excluded (they collide with the `#[]` dynamic segments and the `${}` / `#{}` parameter placeholders), as are whitespace, quotes, brackets, parentheses, semicolons, commas, comment markers and backslashes. An illegal value **throws immediately and names the offending value** — it is never silently sanitized. Legal fields are then quoted per dialect **only when they are reserved words** (backticks for the mysql family, double quotes for oracle/pg, square brackets for sqlserver); non-reserved identifiers stay bare, which avoids the case-sensitivity that quoting would introduce on oracle/pg.

Plain identifiers only: for expressions such as `sum(col)` or `col as alias` keep using `@value` (safety is then on you).

Prerequisite: reserved-word quoting depends on the `spring.sqltoy.reservedWords` configuration, which has no default. When it is not set the reserved-word set is empty, so `ReservedWordsUtil.convertWord` returns the name unchanged without any quoting; a field list containing a reserved word (`order`, `desc`, `level`, `group`, ...) then produces a bare identifier that the database rejects — verified to fail on three databases: PostgreSQL `ERROR: syntax error at or near ","` (the `order` in `select order,staff_code` is parsed as the `ORDER BY` keyword), MySQL `You have an error in your SQL syntax`, and SQL Server `Incorrect syntax near the keyword 'order'`. So **configure the reserved-word list whenever field names may hit one**:

```
spring.sqltoy.reservedWords=order,desc,level,group,key,comment
```

With it configured the same query renders as `select "order",staff_code` on PostgreSQL, `select `order`,staff_code` on MySQL and `select [order],staff_code` on SQL Server — all three were verified to execute and return the column values.

Also note that the `IllegalArgumentException` thrown on validation failure is wrapped into a `DataAccessException` by the DAO layer, so look at the root cause (the end of the `getCause()` chain) when catching it; the message carries the offending value.

Constraint: as with `@loop` and the other macros, when the SQL has no parameters at all it returns early and the macro is not expanded; real report queries always have `:fields`, so this does not bite in practice.

Example:

```
<sql id="report_columns">
       select @joinFields(:cols,{t.*}),count(1) total
       from sqltoy_staff_info t
       #[where t.status=:status]
       group by @joinFields(:cols,{t.organ_id})
</sql>
```

```
// the page configuration decides which columns to select and group by
List result = lightDao.find("report_columns",
        MapKit.keys("cols", "status").values("organ_id,dept_id", 1));
```

9. @df() / @day() (date formatting)

Formats a date into a string and splices it into the SQL — **no knowledge of the target database's date functions required** (this is **string concatenation**, not database function replacement).

```
@df(format)              → current time rendered with the format
@df(param, format)       → the parameter value rendered with the format
@day(...)                → exactly equivalent to @df (historical alias)
```

- The default format is `yyMMdd`;
- An empty or `null` format **emits an empty string** (`@df('')` is a way to omit a whole condition);
- The parameter may be a named parameter (`@df(beginDate, yyyy-MM-dd)`) or the `${beginDate}` form, or a literal; **do not write** `:beginDate` — macro arguments are looked up in the parameter map, whose keys carry no colon.

```
<sql id="query_by_month">
       select * from sqltoy_order_info t
       where t.trans_month = '@df(yyyyMM)'
       #[and t.trans_date >= '@df(beginDate, yyyy-MM-dd)']
</sql>
```

> Difference from the `<to-date>` filter: `@df` formats the **value into SQL text** (producing a literal), while `<to-date>` converts the **parameter to a date type and still binds it as** `?` . Prefer `<to-date>` for external input; `@df` suits report concatenation and partition-table names.

10. @case() (decode-style enumeration)

A `decode` -like value mapping that turns a parameter value into the matching string or SQL fragment:

```
@case(param, value1, result1, value2, result2, ..., defaultResult)
```

- The parameter may be a named parameter (`@case(orderType, ...)`) or the `${orderType}` form;
- Fewer than three parameters emits an empty string;
- **The last parameter is the default result** (emitted when nothing matches).

```
<!-- splice a different ORDER BY per order type -->
select * from sqltoy_order_info t
where t.status=1
order by @case(orderType,'SALE','t.sale_date desc','BUY','t.buy_date desc','t.create_time desc')
```

> The same macro is used for business keys — see Primary Key Strategies · @BusinessId.

11. @substr() / @substring() (substring)

```
@substr(param, startIndex, length)
```

- `startIndex` is 0-based; **a negative value means taking from the tail**;
- A start index at or beyond the string length emits an empty string;
- A blank parameter emits an empty string; `@substring` and `@substr` are equivalent.

```
<!-- use the first two characters of the order id as the business line -->
select * from sqltoy_order_info t
where t.biz_line = '@substr(orderId,0,2)'
```

12. @virtualTable() (cross-database dummy table)

"select constant rows union real query" SQL (dictionary dropdown fallback rows, fixed report total rows, etc.) needs a pseudo-table on oracle/db2/hana, while mysql/pg/sqlserver error out on `from dual`. This macro expands per dialect at parse time:

```
select '0' id,'Global' name @virtualTable()
union
select id,name from sqltoy_dict_detail where status=1
```

- oracle family (oracle/oracle11/dm/oceanbase) → `from dual`; db2 → `from sysibm.sysdummy1`; hana → `from dummy`; stripped for every other dialect;
- Compatible aliases `@dual()` / `@dummyTable()` — all three equivalent, case-insensitive, tolerant of spaces inside the parentheses; may appear several times in one SQL;
- SQL without the macro is completely unaffected; see Dialect Adaptation & Functions.

---

13. @split() (split a parameter into an array)

Splits **one string parameter** into an array by a separator, then feeds it to collection-expansion mechanisms such as `in` — the typical case is the frontend sending a single `"A;B;C"` string while the SQL writes `field in (...)` directly, saving a manual split on the Java side. Handled by `SqlConfigParseUtils` at SQL parse time.

Three forms:

```
-- 1. default comma separator (split by the regex \s*,\s*, tolerant of blanks)
select * from sqltoy_order_info t where t.status in (@split(:statusStr))

-- 2. explicit separator (single or double quotes allowed)
select * from sqltoy_order_info t where t.status in (@split(:statusStr,';'))

-- 3. explicit separator plus numeric conversion (each segment to Integer, blank segments skipped)
select * from sqltoy_order_info t where t.status in (@split(:statusStr,int,','))
```

Notes:

- `@split(...)` is replaced back to `?` at execution time, the split array is put back into the corresponding parameter slot, and then automatically expanded into multiple `?` placeholders by the collection-expansion mechanism;
- Splitting happens only when the parameter value is a `String`; every segment is trimmed automatically, and leading/trailing quotes on the parameter are stripped first;
- Without a second argument the default separator is the comma (`\s*,\s*`).

Java call (the frontend just sends one string):

```
List rows = lightDao.findByQuery(
        new QueryExecutor("sqltoy_order_info_find")
            .names("statusStr").values("1,2,3")   // a single string; no need to pass an array
            .resultType(OrderVO.class))
        .getRows();
```

> Related: `#[]` and filters, Common SQL Examples, Pagination Optimize, Annotations Cheat Sheet.

Annotations Cheat Sheet

Every sqltoy annotation lives in `org.sagacity.sqltoy.config.annotation` (23 in total) and falls into six groups: entity mapping, primary keys and business ids, associations, enterprise features, cache translate and DDL. This page lists each annotation's **target, attributes, defaults and purpose**; follow the links for detailed usage.

> Most POJOs are generated by quickvo straight from database tables and arrive with the annotations filled in. Hand-written annotations are mainly used to adjust mappings, add DDL information such as `@Index` / `@Partition`, or attach enterprise features to existing entities.

1. Overview

Annotation	Target	Purpose	Details
`@SqlToyEntity`	type	Marks a sqltoy entity (semantic alias of `@Entity`; one of the scan markers)	this page
`@Entity`	type	Table mapping: table name / schema / comment / PK constraint name	Object CRUD
`@Column`	field	Column mapping: name, JDBC type, length, precision, default value, etc.	Object CRUD
`@Id`	field	Primary key: generation strategy, sequence, generator, assist key	Primary Key Strategies
`@BusinessId`	field	Business key (order numbers etc.), used with `generateBizId`	Primary Key Strategies
`@SqlToyFieldAlias`	field	Alias mapping when a VO/POJO field name differs	Utilities
`@OneToOne`	field	One-to-one / one-to-many cascade load and save	Object CRUD
`@OneToMany`	field	One-to-many cascade (filtering and sorting supported)	Object CRUD
`@Foreign`	field	Foreign key constraint (DDL generation and referential actions)	POJO to DDL
`@Sharding`	type	Sharding strategy (db, table, concurrency, global rollback)	Sharding
`@Strategy`	type	Sharding strategy reference (nested in `@Sharding`)	Sharding
`@Tenant`	type / field	Multi-tenant column marker	Multi-Tenancy
`@DataVersion`	type / field	Optimistic-locking version column	Data Versioning
`@Secure`	type	Field encryption or masking rule	Masking & Encryption
`@SecureConfig`	type	Container for several `@Secure`	Masking & Encryption
`@Translate`	field	Cache translate for query results (single)	Cache Translate
`@Translates`	field	Container for several `@Translate`	Cache Translate
`@Index`	type	Index definition (goes into DDL)	POJO to DDL
`@Indexes`	type	Container for several `@Index`	POJO to DDL
`@Partition`	type	Table partitioning strategy (goes into DDL)	POJO to DDL
`@PartitionDef`	type	One partition definition (nested in `@Partition`)	POJO to DDL
`@PartitionKey`	field	Partition column marker (MPP table creation and DML hints)	POJO to DDL
`@MppTable`	type	MPP engine metadata (ClickHouse/Doris/StarRocks)	POJO to DDL

2. Entity and column mapping

@SqlToyEntity / @Entity

`@SqlToyEntity` has no attributes — it is a semantic alias of `@Entity`; both mark a class for entity scanning (`packagesToScan`).

`@Entity` attribute	Type	Default	Notes
`tableName`	String	required	Database table name (without schema)
`schema`	String	`""`	Schema name, for multi-schema setups
`comment`	String	`""`	Table comment, written at table creation
`pk_constraint`	String	`""`	Primary-key constraint name, e.g. `PRIMARY`

@Column

Attribute	Type	Default	Notes
`name`	String	required	Database column name
`type`	int	`java.sql.Types.OTHER`	JDBC type; with `OTHER` it is inferred from the Java field type
`nativeType`	String	`""`	Native type name (e.g. `varchar2(50)`, `jsonb`); takes precedence over `type` for DDL
`nullable`	boolean	`true`	Whether the column is nullable
`length`	long	`0`	Length (characters/bytes depending on the type)
`precision` / `scale`	int	`0`	Numeric precision and scale
`generatedType`	`GeneratedType`	`DEFAULT`	Generated column: `DEFAULT` (plain) / `VIRTUAL` / `STORED`
`keyword`	boolean	`false`	Whether the column name is a database reserved word (quoted by object operations)
`comment`	String	`""`	Column comment
`defaultValue`	String	`SqlToyConstants.DEFAULT_NULL`	Default value used in DDL; unset means no default
`autoIncrement`	boolean	`false`	Whether the column auto-increments (DDL)

> `keyword=true` and the global `spring.sqltoy.reservedWords` are two different paths: the former handles quoting of that column in **object operations**, the latter makes **custom SQL** adapt reserved words per dialect. See FAQ.

@SqlToyFieldAlias

Attribute	Type	Notes
`value`	String	Target POJO property name; makes `convertType` map correctly when a VO and a POJO use different field names

---

3. Primary keys and business ids

@Id

Attribute	Type	Default	Notes
`strategy`	String	`assign`	`assign` (manual), `sequence`, `identity`, `default` (22 digits), `nanotime` (26 digits), `snowflake`, `UUID`, `ULID`, `redis`, etc. — same values as Primary Key Strategies
`sequence`	String	`""`	Sequence name for the `sequence` strategy
`generator`	String	`""`	Custom generator class (implementing `IdGenerator`); when set it wins over the built-in `strategy`
`isAssist`	boolean	`false`	Assist key: on partitioned tables the partition column must be part of the primary key, so mark it here (mysql, pg, etc.)

@BusinessId

Attribute	Type	Default	Notes
`signature`	String	required	Signature expression, supporting macros such as `@case(name,v1,then1,...)`, `@day(yyMMdd)`, `@substr(name,start,length)`
`generator`	String	required	Generator class, e.g. `org.sagacity.sqltoy.plugins.id.impl.RedisIdGenerator`
`length`	int	`-1`	Total business-key length; -1 uses the global setting
`sequenceSize`	int	`-1`	Trailing sequence digits; -1 uses the global setting
`relatedColumns`	String[]	`{}`	Associated columns whose values feed the generation
`start`	long	`1`	Initial sequence value

> Only a **single** business key per table can be declared; for several, use `@BusinessId` plus `lightDao.generateBizId(...)` to assign them yourself. See Primary Key Strategies.

---

4. Associations and cascades

@OneToOne / @OneToMany

Both share the same attributes; `@OneToMany` adds `orderBy`:

Attribute	Type	Default	Notes
`fields`	String[]	`{}`	Main-table columns taking part in the association (empty means the primary key)
`mappedFields`	String[]	required	Corresponding columns in the child table
`delete`	boolean	`false`	Whether deleting the parent cascades to children
`load`	String	`""`	Filter applied when cascade-loading (e.g. `status=1`)
`update`	String	`""`	Cascade update control
`notNullField`	String	`""`	Field used for a not-null check
`orderBy`	String	`""`	`@OneToMany` only: ordering of the child collection

> sqltoy supports **one cascade level only**; batch methods such as `loadAllCascade` / `updateCascade` are optimized with IN queries underneath, avoiding N+1.

@Foreign

Attribute	Type	Default	Notes
`constraintName`	String	`""`	Foreign-key constraint name
`table`	String	required	Referenced table
`field`	String	required	Referenced column
`deleteRestict`	int	`1`	Delete referential action (`ReferentialAction`: CASCADE/RESTRICT/SET_NULL/NO_ACTION/SET_DEFAULT)
`updateRestict`	int	`1`	Update referential action, same values

> Since 6.0.3 unsupported referential actions are trimmed per database compatibility. The `Restict` spelling is historical — do not "fix" it.

---

5. Enterprise features

@Sharding and @Strategy

`@Sharding` attribute	Type	Default	Notes
`db`	`@Strategy`	empty	Database sharding strategy
`table`	`@Strategy`	empty	Table sharding strategy
`maxConcurrents`	int	`10`	Shard parallelism: 0 unlimited, 1 serial
`maxWaitSeconds`	int	`-1`	Parallel wait limit (seconds)
`is_global_rollback`	boolean	`true`	Whether a single shard failure rolls everything back

`@Strategy` attribute	Type	Default	Notes
`name`	String	`""`	Strategy bean name
`fields`	String[]	`{}`	Columns feeding the decision
`aliasNames`	String[]	`{}`	Field aliases (parallel to `fields`, for parameter alias scenarios)
`decisionType`	String	`""`	Decision type passed to the strategy (query/update/etc.)

```
@Sharding(db = @Strategy(name = "hashDataSource", fields = { "tenantId" }),
          table = @Strategy(name = "realHisTable", fields = { "bizDate" }))
```

@Tenant

Attribute	Type	Default	Notes
`field`	String	`""`	Tenant field name (entity property name; the framework maps it to the column automatically); when empty the convention/interceptor decides

@DataVersion

Attribute	Type	Default	Notes
`field`	String	`""`	Version property name
`startDate`	boolean	`false`	Whether the version starts with a date (e.g. `202209231`)

@Secure / @SecureConfig

`@Secure` attribute	Type	Default	Notes
`field`	String	required	Column to encrypt or mask
`secureType`	`SecureType`	`ENCRYPT`	`encrypt` (RSA) or a masking type: `tel`, `name`, `email`, `address`, `bank-card`, `id-card`, `public-account`, `discrete-rate`, `other`
`sourceField`	String	`""`	Source column for masking (e.g. taking characters from `staffName`)
`maskCode`	String	`""`	Replacement character, `*` by default
`headSize` / `tailSize`	int	`0`	Leading/trailing characters kept
`maskRate`	int	`0`	Discrete masking ratio (for `discrete-rate`)

`@SecureConfig` has a single attribute `secures` (`Secure[]`) that attaches several `@Secure` rules to one entity.

---

6. Cache translate

@Translate / @Translates

`@Translate` attribute	Type	Default	Notes
`cacheName`	String	required	Cache name (the `cache` defined in `sqltoy-translate.xml`)
`cacheType`	String	`""`	Type inside the cache (e.g. dictionary category)
`cacheIndex`	int	`1`	Which result-set column matches the cache key
`keyField`	String	required	Property that receives the translated value
`split`	String	`""`	Separator for several keys in one column
`join`	String	`""`	Joiner for several translated values
`where`	String	`""`	Conditional translate: translate only when the condition holds
`uncached`	String	`""`	Placeholder emitted when the key is not found

`@Translates` has a single attribute `value` (`Translate[]`).

---

7. DDL-related annotations

Annotation	Required attributes	Notes
`@Index`	`name`, `columns`	`isUnique` (default false), `sortTypes` (ASC/DESC per column)
`@Indexes`	`indexes`	Several `@Index`
`@Partition`	`strategy`	Partition strategy (RANGE/LIST/HASH/RANGE COLUMNS/LIST COLUMNS); `columns`, `expression`, `partitions`
`@PartitionDef`	`name`	One partition: `value`, `tablespace`, `endValue`, `endInclusive` (default true)
`@PartitionKey`	none	Marks the partition column; the partition-key hint for MPP table creation and DML
`@MppTable`	none (as needed)	`engine`, `engineArgs`, `keyModel`, `orderBy`, `distributedBy`, `buckets`, `properties` — the engine metadata of ClickHouse/Doris/StarRocks

```
@SqlToyEntity
@Entity(tableName = "sqltoy_order_info", pk_constraint = "PRIMARY", comment = "orders")
@Indexes({ @Index(name = "idx_order_staff", columns = { "STAFF_CODE" }),
           @Index(name = "uk_order_no", isUnique = true, columns = { "ORDER_NO" }) })
@Partition(strategy = "RANGE", columns = { "TRANS_DATE" }, partitions = {
        @PartitionDef(name = "p202601", endValue = "2026-02-01"),
        @PartitionDef(name = "p202602", endValue = "2026-03-01", endInclusive = false) })
public class OrderInfo implements Serializable {
    @Id(strategy = "default")
    @Column(name = "ID", type = java.sql.Types.VARCHAR, length = 22, nullable = false)
    private String id;
    // ...
}
```

---

8. Common combinations and caveats

- **Entity scanning**: either `@Entity` or `@SqlToyEntity` is enough. Without `packagesToScan` entities are loaded on first use; under **native** it is mandatory — see GraalVM Native Image Guide;
- `@Id` **and** `@Column` **must both be present on a primary-key field**: `@Id` describes the generation strategy, `@Column` describes the column;
- **Leaving** `@Column.type` **as** `OTHER` (inferred from the Java type) is the least error-prone option; use `nativeType` when cross-database DDL needs precise control;
- **Composite keys**: annotate several fields with `@Id`; a lambda join update/delete then correlates through `LPK1…LPKn`;
- **DDL annotations (** `@Index` **/** `@Partition` **/** `@MppTable` **/** `@Foreign` **) only take effect when** `autoDDL=true` **builds tables or when** `lightDao.tableApi()` **generates DDL** — they do not affect runtime CRUD;
- **Enterprise annotations are declarations, not configuration**: `@Tenant` / `@Sharding` / `@Secure` all need the matching interceptor, strategy bean or provider; an annotation alone does nothing.

API Cheat Sheet

`LightDao` is sqltoy's unified entry point: **72 method names, 109 declarations including overloads**, covering object operations, querying, execution, cache translate, extension metadata and chained entry points. This page groups the methods by purpose so you can go from "what I need to do" to "which method" to the detailed page.

> - Authoritative source is the `LightDao` interface (`org.sagacity.sqltoy.dao.LightDao`); `SqlToyDao` / `SqlToyLazyDao` are legacy interfaces — use `LightDao` in new projects.
> - Every method has an underlying implementation built on `QueryExecutor` / `EntityQuery` ; understanding those two gives you flexible combinations — see Query API.
> - Full signatures and Javadoc: javadoc.io.

1. Chained entry points (13)

The chain API is for scenarios needing "many parameters + explicit datasource + batch size + lock mode"; each entry returns a builder that ends with `one()/many()/find()/update()` and friends.

Method	Returns	Purpose	Details
`query()`	`Query`	Chained query	Link Chain API
`save()`	`Save`	Chained save (`saveMode` , `batchSize`)	Link Chain API
`update()`	`Update`	Chained update (`forceUpdateProps`)	Link Chain API
`delete()`	`Delete`	Chained delete	Link Chain API
`load()`	`Load`	Chained load (`lock` , cascades)	Link Chain API
`unique()`	`Unique`	Uniqueness check returning the record	Link Chain API
`execute()`	`Execute`	Execute arbitrary SQL/DML	Link Chain API
`batch()`	`Batch`	Batch execution	Link Chain API
`store()`	`Store`	Stored procedures (incl. multiple result sets)	Stored Procedures
`treeTable()`	`TreeTable`	Tree-table operations	Tree-Table Route
`tableApi()`	`TableApi`	Table/column metadata, truncate/drop	TableApi
`elastic()`	`Elastic`	Elasticsearch (SQL/JSON modes)	Elasticsearch
`mongo()`	`Mongo`	MongoDB query/aggregation	MongoDB

2. Lambda conditions (3)

Method	Purpose	Details
`lambdaQuery(Class)`	Returns a `LambdaQuery` bound to this dao; terminals `find/getOne/findPage/findTop/findRandom/count/exists`	Lambda Conditions
`lambdaUpdate(Class)`	Returns a `LambdaUpdate` ; `set/setExpr` + conditions then `update()`	Lambda Conditions
`lambdaDelete(Class)`	Returns a `LambdaDelete` ; conditions then `delete()`	Lambda Conditions

3. Save and insert (5)

Method	Notes
`save(entity)`	Saves one row, returning the primary key (backfilled by the key strategy)
`saveAll(List)`	Batch save, returns the affected row count
`saveAllIgnoreExist(List)`	Batch save, skipping existing rows (`insert ignore` / `on conflict do nothing` semantics)
`saveOrUpdate(entity, forceUpdateProps...)`	Update when present, insert when absent (merge semantics)
`saveOrUpdateAll(List, forceUpdateProps...)`	Batch merge

4. Update (8)

Method	Notes	Details
`update(entity, forceUpdateProps...)`	Elastic update: non-null fields participate; `forceUpdateProps` forces specific fields	Object CRUD
`updateDeeply(entity)`	Deep update: every field participates (including nulls)	Object CRUD
`updateAll(List, forceUpdateProps...)` / `updateAllDeeply(List)`	Batch elastic/deep update	Object CRUD
`updateByQuery(Class, EntityUpdate)`	Single-table conditional update, supporting `set col = col + ?`	Object CRUD
`updateSaveFetch(...)`	Lock-query, insert when absent, update when present, returning the result (overloads for several unique keys; first choice for high-concurrency ledger scenarios)	Object CRUD
`updateFetch(QueryExecutor, UpdateRowHandler)`	Lock-query → validate → modify → return in one interaction (for scenarios where the record must exist and conditions are freely authored)	Object CRUD
`updateCascade(entity, ...)`	Cascade update (with a per-child-table forced-update-field map)	Object CRUD

5. Delete (5)

Method	Notes
`delete(entity)` / `deleteAll(List)`	Delete by entity primary key
`deleteByIds(Class, Object... ids)`	Delete by a collection of ids
`deleteByQuery(Class, EntityQuery)`	Single-table conditional delete
`truncate(Class)`	Truncate the table

6. Load (8)

Method	Notes	Details
`load(entity)` / `load(entity, LockMode)`	Load by primary key, optionally locked	Object CRUD
`loadById(Class, id)` / `loadByIds(Class, Object... ids)`	Load by id / ids	Object CRUD
`loadAll(List)` / `loadAll(List, LockMode)`	Batch load (optimized with IN)	Object CRUD
`loadCascade(entity, LockMode, Class... cascadeTypes)` / `loadAllCascade(List, Class...)`	Cascade load (choose which child tables)	Object CRUD
`loadEntity(Class, EntityQuery[, resultType])`	Single-table conditional load	Query API
`loadByQuery(QueryExecutor)`	Load through a `QueryExecutor`	Query API

7. Query (15)

Method	Notes	Details
`findOne(sqlOrSqlId, params, resultType)`	Single row	Query API
`find(sqlOrSqlId, params[, resultType])`	List (several parameter-shape overloads)	Query API
`findByQuery(QueryExecutor)`	Unified `QueryExecutor` entry	Query API
`findPage(page, sqlOrSqlId, params[, resultType])`	Pagination	Pagination
`findPageByQuery(page, QueryExecutor)`	Unified pagination entry	Pagination
`findTop(sqlOrSqlId, params, topSize[, resultType])`	First N rows / N%	Query API
`findTopByQuery(QueryExecutor, topSize)`	Same, `QueryExecutor` form	Query API
`findRandom(sqlOrSqlId, params, randomCount[, resultType])`	Random N rows / ratio	Query API
`findRandomByQuery(QueryExecutor, randomCount)`	Same, `QueryExecutor` form	Query API
`findEntity(Class, EntityQuery[, resultType])`	Single-table entity query	Query API
`findPageEntity(page, Class, EntityQuery[, resultType])`	Single-table entity pagination	Query API
`getValue(sqlOrSqlId, params[, resultType])`	Single value	Query API
`getCount(sqlOrSqlId, Map)` / `getCount(Class, EntityQuery)`	Row count	Query API
`isUnique(entity, paramsNamed...)`	Uniqueness check	Query API
`fetchStream(QueryExecutor, StreamResultHandler)`	Streaming callbacks for huge result sets	Query API

8. Execution and everything else

Method	Notes	Details
`executeSql(sqlOrSqlId[, params])`	Execute modifying SQL (four parameter shapes)	Link Chain API
`batchUpdate(sqlOrSqlId, dataSet[, autoCommit])`	Batch execution	Link Chain API
`insertReturnPrimaryKey(sqlOrSqlId, entity, pkField)`	Insert and return the primary key	Link Chain API
`executeStore(storeSqlOrKey, inParamValues)` / `executeMoreResultStore(...)`	Stored procedures (incl. multiple result sets)	Stored Procedures
`parallelQuery(List<ParallelQuery>, paramsMap[, ParallelConfig])`	Parallel query	Parallel Query
`wrapTreeTableRoute(TreeTableModel)`	Build tree node route/level/leaf flags	Tree-Table Route
`generateBizId(signature, increment)` / `generateBizId(entity)` / `generateBizId(table, signature, params, ...)`	Generate business keys	Primary Key Strategies
`convertType(source, resultType, ignoreProperties...)`	DTO↔POJO, List and Page conversion	Utilities
`getTranslateCache(cacheName, cacheType[, resultType])` / `translate(...)` / `cacheMatchKeys(...)` / `existCache(...)` / `getCacheNames()`	Cache-translate APIs (dropdown values, callback translation, reverse key matching)	Cache Translate
`getSqlToyConfig(sqlKey, SqlType)` / `getEntityMeta(Class)`	SQL config / entity metadata (common in report and low-code integration)	Platform Integration
`getSqlToyContext()` / `getDataSource()` / `flush()`	Context, datasource, flush	Technical Architecture

9. Extension points (SPI)

Beyond DAO methods, sqltoy opens up the following interfaces; implement one and attach it through the matching property in Configuration.

9.1 Configured extension points (15)

Property	Interface	Purpose
`unifyFieldsHandler`	`IUnifyFieldsHandler`	Unified common-field filling (created/updated by and at, tenant, etc.)
`typeHandler`	`TypeHandler`	Custom column type mapping (JSON/vector/geometry are built in)
`sqlInterceptors`	`SqlInterceptor`	Central SQL processing (tenant filtering, schema rewriting)
`sqlFormater`	`SqlFormater`	SQL log formatting (depends on druid by default)
`translateCacheManager`	`TranslateCacheManager`	Cache container for translate (beyond ehcache/caffeine)
`dynamicCacheFetch`	`DynamicCacheFetch`	Batch fetch for the FIFO dynamic cache (usually redis-backed)
`dynamicFecthCacheManager`	`DynamicFetchCacheManager`	Dynamic cache manager
`dataSourceSelector`	`DataSourceSelector`	Dynamic datasource routing
`connectionFactory`	`ConnectionFactory`	Connection acquire/release (plain Java scenarios)
`fieldsSecureProvider`	`FieldsSecureProvider`	Field encryption algorithm
`desensitizeProvider`	`DesensitizeProvider`	Masking algorithm
`customFilterHandler`	`FilterHandler`	Custom parameter filter (`<custom-handler>`)
`overTimeSqlHandler`	`OverTimeSqlHandler`	Slow-SQL collection and reporting
`dialectDDLGenerator`	`DialectDDLGenerator`	Custom DDL generation
`firstBizCodeTrace`	`FirstBizCodeTrace`	Record the business call site of a SQL

9.2 Callback and integration interfaces

Package	Interface	Purpose
`callback`	`StreamResultHandler`	Row-by-row streaming callback (`fetchStream`)
`callback`	`UpdateRowHandler` / `UpdateRowCallback`	How `updateFetch` handles the locked rows
`callback`	`EntityUpdateCallback`	Entity update callback of `updateSaveFetch`
`callback`	`DataSourceCallbackHandler`	Work directly on a `Connection` (`doConnection(conn, DBProfile)`)
`callback`	`TransactionHandler`	Custom logic inside a transaction
`callback`	`LockSqlHandler` / `UniqueSqlHandler` / `GenerateSqlHandler`	Dialect-level SQL generation takeover (locking, uniqueness top-N, custom SQL)
`callback`	`GenerateSavePKStrategy`	Custom primary-key backfill strategy
`callback`	`TreeIdAndPidGet`	id/pid extraction for tree sorting
`callback`	`DbAdapterHandler`	Cross-database adaptation callback
`callback`	`DecryptHandler` / `CacheFilter`	Query-time decryption; cache-key filtering (tenant/status dimensions)
`callback`	`RowCallbackHandler` / `PreparedStatementResultHandler` / `CallableStatementResultHandler` / `InsertRowCallbackHandler` / `XMLCallbackHandler`	Row/statement/stored-procedure/batch-insert/XML callbacks (rarely used, reserved for extreme cases)
`integration`	`AppContext` / `ConnectionFactory` / `DistributeIdGenerator` / `MongoQuery`	Framework-agnostic SPIs implemented per Spring/Solon/plain Java

> Custom implementation classes need reflection registration under **native images**; since 6.0.8 the 15 configured extension points above are registered automatically from configuration — see GraalVM Native Image Guide.

Release Notes (6.0 line)

The 6.0 line requires JDK 17+ (JDK 8 projects should stay on `5.6.96.jre8`). This page is an **index**: the full change list, caveats and upgrade actions for each release live in the matching chapter of the 6.0 Upgrade Guide.

Release	Highlights	Details
6.0.8	**Lambda condition API moved into core** (`org.sagacity.sqltoy.lambda`: one builder for single-table and joins; the former sqltoy-plus capability is built in); **GraalVM AOT rewritten** (dynamic registration of resources, extension classes, JSON and driver classes; new `native-reflection-classes` / `native-reflection-packages` / `native-entity-proxies`; fixes the entity hints that had been empty since 5.6); dangling and/or inside brackets cleaned up after `#[...]` removal	10. 6.0.8 · Lambda Conditions · GraalVM
6.0.7	Pluggable JSON library (`JsonProvider`: fastjson2/jackson3/jackson2); new `@joinFields()` field-list joining macro; dangling and/or cleanup inside brackets	9. 6.0.7 · Tags & Expressions
6.0.6	New virtual-table macro `@virtualTable()` (aliases `@dual()` / `@dummyTable()`) solving cross-database "constant rows union query" SQL; quickvo-maven-plugin 2.0.3	8. 6.0.6 · Dialect & Functions
6.0.5	New HighGo dialect (24 → 25 dialects); 11 new cross-database function classes; inserted-key column names lowercased across the PG-folding family (OpenGauss/GaussDB/MogDB/Vastbase/StarDB/Oscar/Kingbase/HighGo)	7. 6.0.5
6.0.4	Cache-translate REST authentication (basic/jwt with token-url login, local self-signing or a static token); function adaptation improvements (`nvl2`, negative-start `substr`, `curdate`, etc.); fixes for silently wrong values in negative-start SubStr and independent HH time tokens	6. 6.0.4
6.0.3	`<sql dialect="xxx">` element attribute; `spring.sqltoy.realDialectFirst`; POJO-to-DDL enhancements (`@Partition` / `@PartitionDef` / `@MppTable`, dedicated Doris/StarRocks/ClickHouse/DB2/HANA/SQLite generators, `@Foreign` referential actions); sharding parallel-execution fixes	5. 6.0.3 · POJO to DDL
6.0.0	JDK 17+; API renames `parallQuery` → `parallelQuery` (`ParallelQuery`) and `doConnection(conn, DBProfile)`; package reshuffle (`PageOptimizeUtils` / `ParallelUtils` / `TranslateUtils`, etc.); new SAP HANA dialect	2. API changes · 3. Package changes
5.6.96.jre8	The final JDK 8 release (still supports quickvo-maven-plugin)	Environment requirements

Finding what changed in a release

1. Locate the release here and follow "Details" into the matching chapter of the upgrade guide;
2. For **line-by-line detail** (root cause of each bug, databases verified, test cases) see `trunk/sqltoy-orm-core/changelog.md` in the source repository;
3. For the "do I need to change code?" answer: sections 2 (API changes) and 3 (package changes) of the upgrade guide — everything else is incremental.

Choosing a version

Scenario	Recommendation
New project	Use the latest 6.0.x
Existing 5.6.x on JDK 8	Stay on `5.6.96.jre8`; do not upgrade your JDK just for the sake of it
Existing 5.6.x on JDK 17	Walk through the upgrade guide section by section, paying attention to the 6.0.0 API/package changes
Using sqltoy-plus	Upgrade to 6.0.8 and migrate to the built-in lambda API per section 10.2
Building native images	6.0.8+ and GraalVM 25+

> For older releases (5.x and earlier) consult the changelog in the source repository; this page only tracks the 6.0 line.

Best Picks

Best Picks

For CRUD, cascading, cross-database access, sharding and the like, sqltoy differs little from hibernate/mybatis. What sqltoy really wants to share is a **different way of thinking** — the "killer features" below are high-value capabilities distilled from extensive project practice.

1. A clean SQL style

Requirement: query the order table with conditions covering a transaction date range, order status, organization, employee and order number.

The mybatis way (`<if>` tags interleaved with the SQL — verbose and error-prone):

```
<select id="findOrder" resultType="OrderInfoVO">
    select * from sqltoy_order_info t
    <where>
        <if test="beginDate!=null"> and t.trans_date &gt;= #{beginDate} </if>
        <if test="endDate!=null"> and t.trans_date &lt;= #{endDate} </if>
        <if test="status!=null"> and t.status=#{status} </if>
        <if test="organId!=null"> and t.organ_id=#{organId} </if>
        <if test="staffCode!=null"> and t.staff_code=#{staffCode} </if>
        <if test="orderId!=null"> and t.order_id=#{orderId} </if>
    </where>
</select>
```

The sqltoy way (clean SQL that can be copied straight into a database client and run; when a parameter is null the corresponding fragment is dropped automatically):

```
<sql id="findOrder">
    <value><![CDATA[
        select * from sqltoy_order_info t
        where 1=1
        #[and t.trans_date>=:beginDate]
        #[and t.trans_date<=:endDate]
        #[and t.status=:status]
        #[and t.organ_id=:organId]
        #[and t.staff_code=:staffCode]
        #[and t.order_id=:orderId]
    ]]></value>
</sql>
```

The business meaning is obvious at a glance, which makes later maintenance, tuning and adjustment much easier. See Dynamic SQL Guide.

2. Fast pagination and count optimization

sqltoy provides intelligent pagination automatically, so developers don't need to know each database's pagination dialect. Its count query is **not** the brute-force `select count(1) from (your sql) as t` ; instead it:

- removes unnecessary `order by` clauses;
- removes unnecessary function computations and irrelevant information between `select` and `from` ;
- fully distinguishes `union` and `group` type queries.

Combined with `@fast()` (paginate first, then join) and `<page-optimize>` (count caching), a pagination request that normally costs 2 queries drops to about **1.35 queries**. See Pagination Optimize.

3. Cache Translate

Delegate the "code value → name" conversion to the cache instead of writing piles of `left join` . Without cache translate, multi-table-join SQL is long and slow; with it, the SQL stays concise and clear and efficiency improves by orders of magnitude. See Cache Translate in Action.

4. Function replacement (cross-database)

Object operations easily cross databases — but what about **hand-written SQL**? Can SQL written for MySQL run directly on Oracle/SQL Server? sqltoy gives the answer: on first load or dynamic execution it **automatically replaces functions in the SQL according to the database type** (e.g. `nvl` / `ifnull` / `isnull` , `concat` , `to_char` , `date_format` , etc.). See Dialect Adaptation & Functions.

5. Preventing SQL injection

Many people misunderstand how SQL injection works and try to defend by "filtering whether parameter values from the page contain SQL fragments" — entirely missing the point. For example, in `select * from table t where t.password=:password`, if password becomes `(select password from table where id=xxx)` such checks can be bypassed.

sqltoy eliminates the problem at its root: **all SQL parameters are ultimately converted into `?` placeholders and set via `PreparedStatement`**, so parameters are always treated as "values" and never as "SQL fragments".

```
-- What you write:
select * from sqltoy_staff_info t where t.staff_name=:staffName
-- What actually executes:
select * from sqltoy_staff_info t where t.staff_name=?
```

In addition, the `<valid-sqlInjection>` filter can validate parameters against injection (see Dynamic SQL Guide).

6. Comments are allowed inside SQL

sqltoy insists that developers copy SQL written in clients such as DBeaver **directly into the xml** (and copy it back from the xml into the client when debugging), so SQL may keep its comments:

- `--` and `/* */` comments are stripped automatically when the SQL is loaded;
- but `/*+ xxxx */` comments are **not** stripped — they usually serve as database hints for SQL performance tuning.

> For more practical patterns, see Common SQL Examples and Tags & Expressions Reference.

FAQ

FAQ

1. Why doesn't sharding support queries and pagination across multiple databases and tables?

The implementation itself is not hard; what's complex is the pagination merge after querying across multiple databases and tables. sqltoy deliberately skips this — **not because it can't be done, but because it isn't necessary** — such a requirement is essentially a design problem. The common practice today is to solve it with an MPP database such as ClickHouse or StarRocks, both of which sqltoy supports.

See Sharding.

2. How is sqltoy-orm's performance?

Solving performance problems was one of sqltoy's original motivations:

- **Primary key strategies**: supports decentralized, application-generated modes (22-digit and 26-digit ordered nanosecond IDs, snowflake algorithm, etc.). Database lock-based modes were tried in the early days — a painful lesson.
- **update operations**: abandons hibernate's "load first, then update" model in favor of a **single database interaction** (elastic update).
- All SQL queries go through `PreparedStatement`, reducing the database's SQL precompilation overhead.
- Batch and cascading loads are based on batch queries, with data assembled in the background by the application, avoiding multiple database round trips (see the underlying implementation of `loadAllCascade`).
- Beyond fast pagination and pagination optimize, the count query is specifically optimized — not the naive `select count(1) from (your sql) as t`.

Most system bottlenecks lie in IO (especially database IO), and sqltoy heavily optimizes database interactions. The framework's own overhead for organizing and processing SQL statements is negligible — typically **under 1 millisecond**.

3. How are database reserved words handled?

Specify the `reservedWords` property in `sqltoyContext` to declare the reserved words your system uses.

- **Object-based operations**: sqltoy handles reserved word compatibility automatically.
- **Custom SQL**: you must handle specific fields per the current database yourself, e.g. reserved words in SQL Server need `[]`:

```
select [maxvalue],name from table
```

Across databases, sqltoy adapts the reserved words in custom SQL according to the defined `reservedWords`. For example, the SQL above automatically becomes the following in a MySQL project:

```
select `maxvalue`,name from table
```

> Note: it converts `[maxvalue]` into `` `maxvalue` ``, not bare `maxvalue` into `` `maxvalue` ``.

4. How do I set up multiple datasources?

For the complete dynamic-datasource configuration, `@DS` switching and `@DSTransactional` cross-database transaction examples, see Multi-Datasource & Transactions.

sqltoy supports configuring a default datasource:

```
spring.sqltoy.defaultDataSource=primaryDB
```

Several implementation options:

1. **dynamic-datasource** (with transaction support): see Multi-datasource & transactions
2. **Define multiple `LightDao` beans**, injecting a different dataSource into each, and wire them by bean name in the service layer.
3. **Query-level switching**: in the `QueryExecutor` of `findByQuery` / `findPageByQuery` etc. you can call `.dataSource(xxx)` directly; the chain API also supports `.dataSource(...)` (see Link Chain API).
4. When **multi-database transactions** are involved, JTA can be used (e.g. Atomikos/Narayana). For sqltoy's native named multi-datasource demo see `MultiDataSourceTest` in sqltoy-showcase.

5. Can sqltoy be combined with other ORM frameworks?

Yes. You can use sqltoy standalone, or combine JPA with sqltoy queries, mixing them flexibly according to your situation.

6. How do I handle multi-column in conditions?

Use the tuple-in syntax directly, passing arrays/collections as parameters:

```
#[and (t.TECH_GROUP,t.PROD_GROUP) in (:techGroups,:prodGroups)]
```

When the in-condition parameter array exceeds 1000 entries (e.g. the Oracle limit), the framework (since 5.1.32 / 4.19.25) **splits it automatically**, so no manual handling is needed. You can also use `@secure-loop` to assemble complex conditions; see Tags & Expressions Reference.

7. Can sqltoy be used outside Spring and Solon?

Yes. The core module `sagacity-sqltoy` can be used in pure Java: build `SqlToyContext` + a datasource + `DefaultLightDaoImpl` yourself, and use `DBTransUtils` for transactions. See Pure Java Usage.

8. Startup fails with a duplicate sqlId — or a later SQL overwrites an earlier one?

By default `breakWhenSqlRepeat=true`: a duplicate `sqlId` **throws and aborts startup**. That is deliberate — when the same SQL name exists in two files, production may well execute the wrong one.

Diagnosis:

```
# For temporary diagnosis only: false lets later definitions overwrite earlier ones (never ship this)
spring.sqltoy.breakWhenSqlRepeat=false
```

- Usual cause: two `*.sql.xml` files reuse one `id` after a module split, or an `@Sql` annotation id collides with an xml id;
- **Remember**: `sqlId` is a global namespace with no module prefix, so add a business prefix (e.g. `crm_staff_findlist`);
- If you really need the same name per datasource, use `<sql dataSource="...">` or dialect variants (`xxx_mysql`) rather than two same-named files.

See Dynamic SQL Guide.

9. Cache translate does not work, or the page still shows old values

Work through these in order:

1. **Nothing was loaded**: `translateConfig` did not pick up your file (default `classpath:sqltoy-translate.xml;classpath:translates`); the startup log shows how many caches were loaded;
2. **Parameters do not line up**: the `columns` of `<translate columns="POST_TYPE" cache="dictCache">` must be **result-set column names**, not property names;
3. **Value types differ**: `1` (number) and `"1"` (string) do not match; normalize with the `to-string` filter;
4. **No incremental update**: by default the cache loads once and never changes; to follow the table configure `cache-update-checker` (timestamp/version based) or `rest-checker` (external endpoint);
5. **FIFO dynamic cache behaves as designed**: in `dynamic-cache` mode only keys **already present locally** are refreshed; a key that was never loaded naturally gets the new value on its next dynamic fetch — this is not a bug;
6. **Several instances**: local caches (ehcache/caffeine) are per-instance, so instances can disagree briefly inside a check interval; for strongly consistent code tables shorten the interval or use `rest-checker`.

See Cache Translate and Large-Scale Cache (FIFO).

10. The count and the data disagree after using `@fast`

`@fast` means "**paginate the inner block to pageSize rows first, then join those rows to outer tables**", so its count **defaults to the inner @fast SQL** (the simplified form, skipping the join filters). If outer join conditions change the row count, the count no longer matches the data.

Options, in order:

1. **Push the filter into the @fast inner block** (anything the inner query can filter should not live in outer join conditions);
2. Provide the count explicitly with `<count-sql>` (`QueryExecutor.countSql(...)` works too) — the most controllable option;
3. Check for `union`: union counts have a dedicated union-all-count optimization (summing branches), and when combined with `@fast` the explicit `count-sql` wins.

See Pagination Optimize.

11. Pagination returns nothing, or raises "maximum records per page exceeded"

Two safety mechanisms, both configurable:

Parameter	Default	Behaviour
`pageFetchSizeLimit`	100000	Maximum records per page; `pageNo=-1` (full extraction) trips it (`-1` disables the limit)
`overPageToFirst`	unset	Whether a page number beyond the last page returns to the first page; when unset the framework's built-in behaviour applies, so set it explicitly

```
# Raise it only for a deliberate large-page export (watch memory)
spring.sqltoy.pageFetchSizeLimit=500000
# Send out-of-range page numbers back to the first page
spring.sqltoy.overPageToFirst=true
```

> For large exports prefer `fetchStream` (streaming callbacks) or batched `findTop` over pagination. See Pagination Optimize.

12. Why does lambda's `getOne()` throw?

That is the **strict semantics** introduced in 6.0.8: 0 rows returns `null`, **more than one row throws** `DataAccessException` **(carrying the match count)**. The point is to stop a defective condition from being silently masked into "take the first row".

How to proceed:

- If the condition really should be unique, check for missing tenant/status conditions;
- If several rows are legitimate but you only want one, use `find()` and take the first, or add `orderBy` + `findTop(1)`;
- Likewise, `update()/delete()` **with no where condition throws by default**; use an explicit `.matchAll()` when a whole-table operation is truly intended.

See Lambda Conditions.

13. Lambda conditions are unavailable in a native image, or startup reports missing reflection

Lambda relies on `SerializedLambda` and serialization mirroring to extract property names, so a closed-world native image must register them explicitly:

- **On the application side**, register by "lambda defining class" (Spring: `lambdaCapturingTypes` in `META-INF/native-image/**/serialization-config.json`, or `registerLambda(definingClass, b -> b.withInterfaces(LambdaFn.class))`; Solon: `registerLambdaSerialization(definingClass)`);
- Without registration the whole lambda system is unavailable under native (JVM mode is unaffected);
- Separately, `updateSaveFetch` **under native needs** `native-entity-proxies` to pre-generate the proxy, otherwise that branch is unavailable.

The complete checklist (entity scanning, DTO reflection, extension classes, function classes, JSON/vector/geometry driver classes) is in the GraalVM Native Image Guide.

14. What must change when migrating from sqltoy-plus or MyBatis?

- **sqltoy-plus**: 6.0.8 folds the lambda capability into core. Drop the plugin dependency and `@EnableSqlToyPlus`, then follow the mapping table in upgrade guide section 10.2 for package/class renames, and regression-test the two behaviour changes (`getOne()` with multiple matches, unconditional update/delete);
- **MyBatis / MyBatis-Plus**: `<if>` → `#[]`, `QueryWrapper` → `EntityQuery` or lambda, `ResultMap` → result type/VO mapping, pagination plugin → `findPage`. The full mapping and rollout order are in Migrating from MyBatis.