BtrQL Documentation

Offline documentation generated from the same Eleventy data as the public documentation website. Includes user docs, the in-page dialect feature matrix, and generated example source/output panels embedded in documentation.

Contents

Core Ideas

BtrQL reuse is compile-time source organization. Modules, relation types, table-type column lists, extension methods, extension columns, val bindings, compile-time macros, and project compilation keep shared logic in checked BtrQL while generated SQL stays explicit.

Extension methods and extension columns use receiver-oriented reuse: a named row-shape contract owns reusable relation operations and scalar computations that apply to any compatible relation.

Reusable relation logic

This example shows reusable relation types and extension methods called from multiple use sites before SQL lowering.

BtrQL SQL
source.btrql

```
using analytics

val cutoff = 7 + 3

type AuditCols = [id: INT, created_at: TIMESTAMP]

type UserShape = AuditCols ++ [name: VARCHAR, active: BOOLEAN]

extension UserShape {
  method keepRecent =
    self
      .where(id >= cutoff)
}

extension [id: INT] {
  method keepRecentIds =
    self
      .where(id >= cutoff)
}

view recent_users =
  users
    .keepRecent()

view recent_active_users =
  users
    .keepRecent()
    .where(active == TRUE)
    .select(id, name)

view recent_user_ids =
  users
    .keepRecentIds()
    .select(id)

view recent_named_ids =
  users
    .keepRecentIds()
    .where(name != '')
    .select(id)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" },
        { "name": "created_at", "type": "TIMESTAMP" }
      ]
    }
  }
}
```

source.btrql schemas.json
postgresql.sql

```
CREATE VIEW "recent_users" AS
SELECT *
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q1"
WHERE "id" >= 10;

CREATE VIEW "recent_active_users" AS
SELECT
  "id",
  "name"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q3"
WHERE
  "id" >= 10
  AND "active" = TRUE;

CREATE VIEW "recent_user_ids" AS
SELECT "id"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q5"
WHERE "id" >= 10;

CREATE VIEW "recent_named_ids" AS
SELECT "id"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q7"
WHERE
  "id" >= 10
```

```
  AND "name" <> '';
```

clickhouse.sql

```
CREATE VIEW `recent_users` AS
SELECT *
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q1`
WHERE `id` >= 10;

CREATE VIEW `recent_active_users` AS
SELECT
  `id`,
  `name`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q3`
WHERE
  `id` >= 10
  AND `active` = TRUE;

CREATE VIEW `recent_user_ids` AS
SELECT `id`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q5`
WHERE `id` >= 10;

CREATE VIEW `recent_named_ids` AS
SELECT `id`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q7`
WHERE
  `id` >= 10
```

```
  AND `name` <> '';
```

PostgreSQL ClickHouse

Relation types and extension columns

A relation type names a reusable row contract. It can drive typed projections, grouped keys, column removal, and extension receivers.

```
type AuditCols = [id: INT, created_at: TIMESTAMP]

type UserShape = AuditCols ++ [name: VARCHAR, active: BOOLEAN]

users
  .select[UserShape]
```

An extension column names a receiver-bound scalar expression. A relation that satisfies the extension's minimum row contract can use that column in filters or request it by name in `select` and `addColumn`. Dependencies such as `is_engaged` are substituted into later extension-column expressions, but they are not added to the output unless requested.

```
using analytics

type EngagementInput = [
  posts_viewed: INT,
  comments_written: INT
]

extension EngagementInput {
  method activeOnly =
    self
      .where(is_engaged)

  method withMinimumPosts()(minPosts: INT) =
    self
      .where(posts_viewed >= minPosts)

  column engagement_score: INT =
    posts_viewed + comments_written * 5

  column is_engaged: BOOLEAN =
    posts_viewed > 10 and comments_written > 0

  column engagement_label =
    case when is_engaged then 'active' else 'inactive' end
}

view engaged_users =
  users
    .activeOnly()
    .withMinimumPosts()(5)
    .addColumn(engagement_score, engagement_label)
```

This example compiles with the PostgreSQL and ClickHouse C# prototypes when the `analytics.users` schema supplies the receiver columns. The former `column recipe` declaration and `addColumn[Type]` request are removed syntax and produce migration diagnostics.

Table type algebra

An extension method has a symbolic table return type. `T0` is the open type of the receiver (`self`); `T1`, `T2`, and later variables are table-kind parameters in declaration order. A variable is *open*: call-site columns beyond its minimum receiver contract remain present until a projection closes that variable. `ScalarType0`, `ScalarType1`, and later names are schematic scalar slots; when the compiler knows a scalar type, Outline shows the concrete type instead.

Form	Meaning
`A ++ B`	concatenate table shapes, or add the columns in `B`

Form	Meaning
`A -- B`	remove the named, type-compatible columns in `B`
`A && B`	keep the columns shared by both declared table types

Derived return types normally normalize to open variables plus ordered `++` and `--` deltas. Read the deltas in pipeline order. A rename is therefore a removal followed by an addition, and replacing a column's scalar type uses the same pair.

```
receiver        table argument        computed delta         removal delta
   T0       ++       T2         ++ [newColumn: ScalarType1] -- [removedColumn:
ScalarType2]
    └──────────────────────────────── pipeline result ──────────────────────────────┘

=> T0 ++ T2 ++ [newColumn: ScalarType1] -- [removedColumn: ScalarType2]
```

For example, an extension body that joins its second table argument, adds `newColumn`, and removes `removedColumn` produces exactly that return type. `++ T2` means row-shape concatenation; it does not mean SQL set union.

Return-type diagrams for table methods

The diagrams below cover every table method classified by extension-method return inference. Methods with the same shape rule are grouped together. `Tn` means whichever open table variable is on the left of the call, `R` is a right-side relation result, and `R?` is the nullable-relation constructor defined in the join section below.

Shape-preserving methods

Filtering, ordering, validation, paging, aliases, native clauses, and DML commands preserve the incoming table type. DML keeps this type until a `returning` step projects another shape.

```
Tn ── where | qualify | orderBy | distinct | distinctOn | rowShape
   ── limit | offset | as
   ── prewhere | sample | limitBy | settings | format
   ── insert | update | delete | deleteFrom | mergeInto ──────────────► Tn
```

Set methods

`union`, `unionAll`, `intersect`, and `except` return the type of their left operand. They validate the right operand but do not concatenate it.

```
self:T0  ── union | unionAll | intersect | except (other:T1) ──▶ T0
other:T1 ── union | unionAll | intersect | except (self:T0)  ──▶ T1
```

Column delta methods

`addColumn` appends new columns and replaces a collision with an ordered remove/add pair. `removeColumn(...)` and `removeColumn[Type]` subtract columns. ClickHouse `arrayJoin` and `leftArrayJoin` retain the input columns and append each named element alias.

```
Tn ── addColumn(expr -> newColumn) ──────────────▶ Tn ++ [newColumn: ScalarType0]
Tn ── removeColumn(oldColumn) ───────────────────▶ Tn -- [oldColumn: ScalarType1]
Tn ── addColumn(expr -> existingColumn) ─────────▶ Tn -- [existingColumn: OldType]
                                                    ++ [existingColumn: NewType]
Tn ── arrayJoin(items -> item) ──────────────────▶ Tn ++ [item: ElementType]
Tn ── leftArrayJoin(items -> item) ──────────────▶ Tn ++ [item: ElementType]
```

Projection methods

Projection first decides which input columns survive under their original names. The removed columns are the complement of the preserved columns; they are not a second set of changes.

```
Tn ── select / select[Type] / returning / returningWith
   ── (preserve existing columns under their original names)
   ──▶ Tn -- [columns not preserved under their original names]
       ≡ [preserved columns with their original names and types]
```

A computed projection output is added after that closing projection.

```
Tn ── select / returning / returningWith (..., expression -> result)
   ──▶ Tn -- [columns not preserved under their original names]
           ++ [result: ExpressionType]
```

Rename-only algebra

A direct-column alias is a rename. Its type rule is independent of the projection or grouping operation that contains it.

```
Tn ── sourceColumn -> renamedColumn ──▶ Tn -- [sourceColumn: SourceType]
                                           ++ [renamedColumn: SourceType]
```

Inside a projection, `sourceColumn` is already among the columns not preserved under their original names. The normalized projection therefore removes it once and adds `renamedColumn`; it does not apply a second removal.

Grouping methods

Grouping preserves direct keys under their original names, removes non-keys, and adds aggregate results. If a key is aliased, apply the rename-only rule above to that key separately.

```
Tn ── group { by(key columns) aggregate(...) } / group { by[Type] ... }
   ──► Tn -- [every non-key column]
           ++ [aggregate result columns: ResultTypes]
```

Join and APPLY methods

Each rule below is independent. In a symbolic relation result, `X?` means "the same columns as `X`, with every scalar made nullable." A bare `X` keeps the columns' existing nullability.

```
innerJoin / crossJoin : T0 ++ T1
leftJoin              : T0 ++ T1?
rightJoin             : T0? ++ T1
fullJoin              : T0? ++ T1?

crossApply(R)          : T0 ++ R
outerApply(R)          : T0 ++ R?
```

An inner or cross join keeps both sides as they are. A left join can produce a left row with no matching right row, so only `T1` becomes nullable. A right join is the mirror image; a full join may lack either side. Likewise, `outerApply` retains a left row even when its dependent result has no row, so the columns contributed by `R` become nullable.

The compiler and Outline now preserve this as a first-class symbolic relation type: an extension method containing a left join renders `T0 ++ T1?`, not a remove/add replacement for every column in `T1`. Instantiating `T1?` makes all bound `T1` columns nullable, including extra call-site columns beyond the method's minimum contract. Nested extension calls preserve the wrapper during type substitution.

`Tn?` is result-type notation, not BtrQL source syntax. PostgreSQL renders the resulting nullable *scalar* details as forms such as `INTEGER?`; ClickHouse uses forms such as `Nullable(Int32)`. A declared table function has a closed, concrete return shape and no open `Tn` variable, so its result continues to show per-column scalar types. If table functions later gain table-kind parameters, the same `Tn?` relation algebra can apply to them. There is no established `!` suffix; a bare relation variable means "preserve current nullability."

`jsonTable` and `rowsFromWithOrdinality` contribute their declared result columns through the APPLY rule. `outerApply` itself is available only in dialects that support that operation.

Extension-method composition

Think of an extension method as a reusable row-shape transformation:

1. `T0` stands for the actual relation to the left of the method call.
2. `T1`, `T2`, and later variables stand for table arguments in declaration order.
3. A call binds those placeholders to the receiver and arguments at that point in the pipeline.
4. The method's additions and removals are applied, and the next call continues from that result.

For a simple table-argument call, the binding is:

```
call:    current_users.join_orders(current_orders)()
bind:    T0 = current_users, T1 = current_orders
result:  current_users ++ current_orders
```

Here is a complete example that compiles with both C# prototypes:

```
using analytics

extension [user_id: INTEGER, user_name: VARCHAR, region: VARCHAR] {
  method only_region()(wanted_region: VARCHAR) =
    self
      .where(region == wanted_region)

  method join_orders(other: [
    order_id: INTEGER,
    buyer_id: INTEGER,
    amount: INTEGER,
    status: VARCHAR
  ]) =
    self
      .innerJoin(other on user_id == buyer_id)

  method reshaped_join(other: [
    order_id: INTEGER,
    buyer_id: INTEGER,
    amount: INTEGER,
    status: VARCHAR
  ]) =
    self
      .removeColumn(region)
      .crossJoin(
        other
          .removeColumn(buyer_id)
          .addColumn(amount + 1 -> adjusted_amount)
      )
}

view regional_orders =
  users
    .only_region()('EU')
    .join_orders(orders)()

view public_order_rows =
  users
    .reshaped_join(orders)()
```

Read its inferred shapes in ordinary language:

- `only_region` filters rows and keeps `T0` unchanged.
- `join_orders` appends the table argument, producing `T0 ++ T1`.
- `reshaped_join` keeps each change attached to its source: `T0 -- [region: VARCHAR] ++ T1 -- [buyer_id: INTEGER] ++ [adjusted_amount: INTEGER]`.

`regional_orders` shows call-site composition: filter `users`, then join `orders`. The first call does not change the row shape, so the second call still receives the complete user relation as its `T0`. `public_order_rows`

invokes the origin-preserving reshape so that example is compiled too.

An extension column is the scalar special case: requesting `column score =...` has symbolic result `T0 ++ [score: ConcreteScalarType]`.

Modules and projects

BtrQL modules are file-based. A source file such as `input/reporting/shared.btrql` becomes module `reporting.shared`, and configured source roots decide where module lookup begins.

```
import reporting.shared { base_users -> users_base }
using analytics

view active_users =
  users_base
    .where(active == TRUE)
```

A project ties source files, generated output, target dialect, and schema metadata together.

```
{
  "output-folder": "generated",
  "target-dialect": "postgresql"
}
```

The schema cache stores database metadata used for offline checks: schemas, tables, columns, types, and callable signatures. It does not store table rows. A project build follows imports in dependency order, writes generated SQL, and reports missing modules, duplicate names, import cycles, stale schema references, and diagnostics before deployment.

Post-compile actions

Project compiler integrations should expose enough result data for post-compile work. The main project compiler supports configured post-compile hooks after SQL generation and reports hook diagnostics and artifacts.

The native project APIs currently return compiled modules with source paths, output paths, SQL, diagnostics, output roots, and relation-shape maps after `CompileDirectory(...)`, so callers can attach their own post-compile steps. Those APIs do not currently run configured hook files themselves; their editor build responses expose successful default post-compile status when no hook runner is configured.

VS Code Getting Started

BtrQL is easiest to try from Visual Studio Code: install the demo VSIX for your operating system and architecture, then use the bundled language support for syntax highlighting, diagnostics, formatting, and workspace commands.

Production does not need a BtrQL runtime. The editor helps you write and check source; the deliverable remains generated SQL that you review, test, and run with existing database drivers, migration tools, and runbooks.

Download The Demo VSIX

Direct demo downloads:

- VS Code package: use the download page to pick the demo VSIX for your operating system and architecture.
- Optional examples bundle: BtrQL-Examples.zip

Requirements:

- Visual Studio Code
- the demo VSIX matching your platform
- on Windows, the same.NET 11 runtime required by the Windows CLI toolchain

The macOS and Linux packages are self-contained. The Windows VSIX uses the framework-dependent Windows CLI/LSP toolchain built on macOS.

The Free Local license applies only to demo artifacts. The VSIX does not report project files, schemas, generated SQL, telemetry, or error reports over the internet.

Install The VSIX

Use the Visual Studio Code command palette and run **Extensions: Install from VSIX...**, then select the downloaded package such as `btrql-demo-osx-arm64.vsix`.

After installation, open a folder containing `.btrql` files and reload the window if VS Code prompts you.

The demo VSIX includes syntax highlighting, local language support, formatting, and workspace build commands. Debugger support is not included in the demo VSIX.

Run Commands In VS Code

Open the command palette and type `BtrQL` to see the installed commands.

- **BtrQL: Outline** shows inferred relation types, columns, imports, and lineage.
- **BtrQL: Configuration** controls optional Outline metadata.
- **BtrQL: Build Project** reads `project.json` and writes generated SQL into the configured `output-folder`.

- Full-capability packages also enable SQL conversion, schema comparison, and pipeline debugging commands.

While you edit `.btrql` files, diagnostics, completion, formatting, and navigation stay available directly in the editor. See the VS Code extension guide for the complete command reference, result-type completion, and troubleshooting.

Optional Examples Bundle

Download the examples ZIP for two ready-to-open, source-only projects: `postgresql/` and `clickhouse/`. Each project contains separate `.btrql` files for features supported by that target; generated SQL and unsupported-dialect diagnostics are intentionally omitted.

Extract the ZIP, open the target project in VS Code, and configure its live database through **BTrQL: Configure Live Database**. The included `project.json` uses obvious credential placeholders and the `vscode-secret` profile, so real credentials can stay in VS Code SecretStorage.

Open A BtrQL Project

The smallest useful project has source files, schema metadata, and a generated SQL folder:

```
project/
  project.json
  schema metadata
  src/
    main.btrql
  generated/
```

`project.json` names the target dialect, schema metadata path, and generated SQL folder:

```
{
  "output-folder": "generated",
  "target-dialect": "postgresql",
  "schema-cache-path": "schema metadata"
}
```

Open the folder in VS Code, edit `.btrql` files, and use the generated SQL as the review surface before deployment.

Existing SQL Files

BtrQL projects can contain both `.btrql` and `.sql` files. Keep existing SQL where it already works, then add BtrQL for new or repeated query logic.

When the compiler can parse a SQL file, objects defined there can expose typed symbols to the rest of the BtrQL project. If a SQL feature is not supported by the language yet, create the table, view, function, or stored procedure in a test database and import its catalog metadata into the BtrQL project.

Where To Go Next

- Use the VS Code extension guide for editor features, commands, Outline, and troubleshooting.
- Use Syntax overview for the grammar and core source surface in detail.
- Use Walkthroughs for end-to-end source and generated SQL walkthroughs.
- Use Actual novelty when moving from one source file to a project layout.
- Use Application integration for ownership, bind variables, result mapping, deployment boundaries, and examples in Java, Python, and TypeScript.
- Check Feature matrix before relying on a dialect or feature.
- Read Support scope to understand current non-goals and unsupported surfaces.

Frequently Asked Questions

How does the BtrQL compiler validate table names and columns offline?

BtrQL uses a cached metadata JSON file or active database connections specified in `project.json`. The compiler uses this information to build a local schema cache, allowing it to validate query shapes entirely offline in your editor without querying a live database.

Can I use the VS Code extension without installing the local BtrQL CLI?

The VS Code extension includes an embedded version of the compiler, so you can perform basic editing, syntax highlighting, and queries out-of-the-box. However, the CLI tool is required for advanced build scripts and CI/CD pipelines.

What is the purpose of the `project.json` file in BtrQL workspaces?

The `project.json` file serves as the configuration root for a BtrQL workspace, defining compiler options, dialect targets, schema metadata sources, and source file locations.

VS Code extension

What The VSIX Includes

The BtrQL VSIX bundles syntax highlighting, the PostgreSQL and ClickHouse language servers, compiler integration, formatting, navigation, and the BtrQL Outline. It runs locally and uses the `project.json` and schema metadata in the folder you open. Generated SQL remains the deployment artifact; the extension is an authoring and build-time tool.

The demo package includes compilation, formatting, diagnostics, completion, navigation, Outline, and project builds. Commands that require reverse conversion, schema comparison, or debugging remain visible but explain that a full-capability package is required.

Editor And Language Features

For `.btrql` files the extension provides:

- semantic syntax highlighting and live diagnostics;
- formatting through **Format Document**;
- completion for query operations, tables, fields, imports, extension members, and earlier sequential aliases;
- hover, signature help, go to definition, references, document highlights, workspace symbols, and inlay hints;
- the standard VS Code Outline and the richer BtrQL Outline;
- local project builds using the bundled compiler.

Completion is relation-aware and position-specific. At top level it offers only top-level statements such as `extension`; at a direct extension-member boundary it offers only `method` and `column`. A relation step after `.` offers applicable relation methods, a scalar expression offers fields, constants, scalar functions, and applicable extension columns, and a relation operand or table-kind argument offers compatible relations and table functions. Scalar- and relation-type positions receive their corresponding type categories. Qualified scalar suggestions are resolved against the relation alias, and earlier aliases in `select` or `addColumn` are available only to later expressions.

Freeze A Derived Extension Result

An extension method can normally omit its result annotation and let the compiler infer it. To protect a reusable method from unintended shape changes, insert `=>` immediately before its existing `=`. In that empty annotation slot, the editor offers exactly one completion: the complete currently derived symbolic result.

```
extension [user_id: INTEGER] {
  method joinOrders(other: [buyer_id: INTEGER]) => T0 ++ T1 =
    self
      .innerJoin(other on user_id == buyer_id)
}
```

Receiver and table-argument shapes are minimum contracts. This method requires only the join key from `T0` and the join key from `T1`; wider relations satisfy those contracts and retain their additional columns when `T0 ++ T1` is instantiated.

The suggestion is generated from the same semantic renderer as Outline. Accepting it does not override inference: it adds a checked annotation, and a later implementation change that produces a different normalized relation type becomes an error. A body must already follow `=` so the editor has a result to derive.

BtrQL Outline

Run **BtrQL: Outline** or open **BtrQL Outline** in Explorer. It shows relations, imported modules, schema symbols, output columns, inferred scalar types, lineage, extension columns, and symbolic extension-method results. Select an entry to navigate to its declaration.

The standard VS Code Outline and BtrQL Outline use the same result-type renderer. Relation variables are displayed as `T0`, `T1`, and later variables; concrete scalar types retain names such as `INTEGER`, `VARCHAR`, `Int32`, and `String`. Use **BtrQL: Configuration** to choose optional column metadata shown in the BtrQL Outline.

Command Reference

Open the command palette and type `BtrQL`.

Command	Availability	Purpose
BtrQL: Outline	Demo and full	Focus the BtrQL Outline and refresh inferred relation types.
BtrQL: Configuration	Demo and full	Select optional Outline metadata columns.
BtrQL: Build Project	Demo and full	Compile the current workspace using `project.json`.
BtrQL: Convert SQL Selection To BtrQL	Full	Reverse the selected SQL into BtrQL.
BtrQL: Show Schema Comparison	Full	Compare declared/cached and live schema metadata.
BtrQL: Debug Pipeline At Cursor	Full	Start a relation-pipeline debugging session at the cursor.
BtrQL: Show Debug History	Full	Reopen recent debugger sessions and evaluations.
BtrQL: Smart Delete Left	Demo and full	Preserve BtrQL-aware deletion behavior; bound to Backspace in writable BtrQL editors.

VS Code's built-in **Format Document**, **Go to Definition**, **Go to References**, **Show Hover**, and **Go to Symbol** commands invoke the corresponding BtrQL language features when a `.btrql` document is active.

Settings

The extension exposes these workspace settings:

- `btrql.lsp.logLevel` controls language-server logging.
- `btrql.lsp.wireTrace` writes JSON-RPC traffic to the local runtime log directory for troubleshooting.
- `btrql.lsp.targetDialectVersion` temporarily overrides the dialect version from `project.json`.
- `btrql.lsp.maxRestartCount` limits automatic language-server restarts before crash-loop protection stops retrying.

Prefer project configuration for checked-in dialect and schema behavior. Use editor settings for local diagnostics and troubleshooting.

Installation And Troubleshooting

Install the package with **Extensions: Install from VSIX...**, open the folder containing `project.json`, and reload the window if prompted. The VSIX must match the operating system and architecture because it contains local native executables.

If language features do not start:

1. confirm that VS Code opened the project folder rather than a single file;
2. verify that `project.json` names `postgresql` or `clickhouse` and points to readable schema metadata;
3. open **Output** → **BtrQL** to inspect startup diagnostics;
4. enable `btrql.lsp.wireTrace` only while investigating protocol behavior;
5. run **BtrQL: Build Project** to distinguish editor state from compiler diagnostics.

CLI tooling

Use the BtrQL CLI when you want reproducible commands outside an editor. The CLI is build-time tooling only: production runs generated SQL through your normal database path.

Download a demo package

Download a PostgreSQL or ClickHouse demo package from the download page, then choose the archive matching your operating system and CPU architecture. Download the curated examples for source-only PostgreSQL and ClickHouse projects.

The demo package includes command-line compilation, formatting, and local language support binaries.

Verify the CLI

After extracting a package, verify the launcher:

```
./tools/btrql-postgresql/bin/btrql --help
```

Common workflow commands compile a file or project, format BtrQL source, and report diagnostics before SQL is written.

Application integration

BtrQL emits SQL. Application code, database drivers, migration tools, schedulers, and deployment systems continue to do the work they already own.

Ownership

The default ownership rule is simple: whoever owned the SQL before should own the BtrQL source and generated SQL. Platform or CI teams can provide compiler, formatter, and review gates without becoming owners of query semantics.

Workflow

- write or update BtrQL source
- compile to SQL before deployment
- review generated SQL in code review
- run database-specific tests or migrations as usual
- execute deployed SQL through existing application code and database drivers

Do not generate SQL during application startup. Treat generated SQL as a build artifact that can be reviewed, tested, and deployed deliberately.

Feature Matrix

This matrix is derived from the current native PostgreSQL and ClickHouse implementations. Status values are intentionally simple:

- Supported: the checked native subset is available.
- Partial or gated: the surface or its proof is intentionally bounded.
- Unsupported: the native implementation does not provide it.
- Not applicable: the feature does not make sense for that direction.

Dialect boundary

Read the matrix as a planning guide, not a promise that every vendor-specific SQL feature is abstracted. BtrQL favors explicit dialect errors over generating SQL that only looks portable.

PostgreSQL

Native compiler for PostgreSQL 16–18. Project compilation and incremental LSP are implemented; reverse is a bounded full-profile subset, while full debugger execution and post-compile hooks remain gaps.

● Supported ● Partial or gated ● Unsupported ● Not applicable
BtrQL to SQL ●
SQL import ●
Editor ●

Compiler and project

Feature	BtrQL syntax	Examples	Support
Native parse, check, lower, render Tree-sitter source mapping, typed model, semantic checks, and native dialect rendering.	`users.where(active == TRUE).select(id)`	Query Pipeline	BtrQL to SQL: ● SQL import: ● Editor: ●
Mixed projects and incremental builds Compiles.btrql and.sql modules with dependency invalidation, source overrides, and relative outputs.	`import reporting.shareusers.select(id)`	Project Compilation	BtrQL to SQL: ● SQL import: ● Editor: ●

Feature	BtrQL syntax	Examples	Support
Values, modules, extensions, and declared shapes Reusable compile-time source and typed project symbols.	`val cutoff = 10extension onlyActive(source) = source.where(active == TRUE)`	Relation Type And Extension Reuse	BtrQL to SQL: ● SQL import: ● Editor: ●

Query and dialect surface

Feature	BtrQL syntax	Examples	Support
Joins, CTEs, grouping, windows, and set operations The shared promoted query-composition subset.	`with { recent = users.where(active) } recent.groupBy(team_id).select(team_id, count())`	Joins, WITH Bindings, And Set Operations	BtrQL to SQL: ● SQL import: ● Editor: ●
PostgreSQL versioned forms DISTINCT ON, ROWS FROM WITH ORDINALITY, JSON_TABLE, RETURNING WITH, and temporal-key forms in their checked versions.	`users.distinctOn(team_id)rowsFromWithOrdinality(...)`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●

Feature	BtrQL syntax	Examples	Support
DML, schema objects, and routines PostgreSQL has the broader promoted authoring surface; ClickHouse is intentionally narrower.	`users.where(id == 1).update(name -> 'Ada').returning(id)`	RETURNING Across Dialects	BtrQL to SQL: ● SQL import: ● Editor: ●

Tooling and known gaps

Feature	BtrQL syntax	Examples	Support
Incremental LSP and navigation Ranged document sync is implemented; navigation is strongest in the active document and is not rename-complete across the workspace.	`textDocumentSync.change = 2`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●
Trace and debugger runtime Full-profile custom LSP/MCP endpoints exist, but the native CLI has no trace/debug commands and no general transaction/savepoint runtime.	`native CLI trace/debug: unavailable`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●
Post-compile project hooks Missing in the native implementation. PostgreSQL is planned first; ClickHouse is outside the initial scope.	`post-compile-hooks.json`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●

ClickHouse

Native compiler targeting ClickHouse 24.8, including native query clauses and types. Reverse/editor support is bounded; routines, broad lifecycle DDL, post-compile hooks, and the final long soak are not promoted.

● Supported ● Partial or gated ● Unsupported ● Not applicable
BtrQL to SQL ●
SQL import ●
Editor ●

Compiler and project

Feature	BtrQL syntax	Examples	Support
Native parse, check, lower, render Tree-sitter source mapping, typed model, semantic checks, and native dialect rendering.	`users.where(active == TRUE).select(id)`	Query Pipeline	BtrQL to SQL: ● · SQL import: ● · Editor: ●
Mixed projects and incremental builds Compiles.btrql and.sql modules with dependency invalidation, source overrides, and relative outputs.	`import reporting.shareusers.select(id)`	Project Compilation	BtrQL to SQL: ● · SQL import: ● · Editor: ●
Values, modules, extensions, and declared shapes Reusable compile-time source and typed project symbols.	`val cutoff = 10extension onlyActive(source) = source.where(active == TRUE)`	Relation Type And Extension Reuse	BtrQL to SQL: ● · SQL import: ● · Editor: ●

Query and dialect surface

Feature	BtrQL syntax	Examples	Support
Joins, CTEs, grouping, windows, and set operations The shared promoted query-composition subset.	`with { recent = users.where(active) } recent.groupBy(team_id).select(team_id, count())`	Joins, WITH Bindings, And Set Operations	BtrQL to SQL: ● SQL import: ● Editor: ●
ClickHouse native query clauses and types PREWHERE, ARRAY JOIN, SAMPLE, LIMIT BY, SETTINGS, FORMAT, declared native types, and table functions.	`orders.prewhere(active).limitBy(1, status).settings(max_threads = 1).format(JSONEachRow)`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●
DML, schema objects, and routines PostgreSQL has the broader promoted authoring surface; ClickHouse is intentionally narrower.	`users.where(id == 1).update(name -> 'Ada').returning(id)`	RETURNING Across Dialects	BtrQL to SQL: ● SQL import: ● Editor: ●

Tooling and known gaps

Feature	BtrQL syntax	Examples	Support
Incremental LSP and navigation Ranged document sync is implemented; navigation is strongest in the active document and is not rename-complete across the workspace.	`textDocumentSync.change = 2`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●

Feature	BtrQL syntax	Examples	Support
Trace and debugger runtime Full-profile custom LSP/MCP endpoints exist, but the native CLI has no trace/debug commands and no general transaction/savepoint runtime.	`native CLI trace/debug: unavailable`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●
Post-compile project hooks Missing in the native implementation. PostgreSQL is planned first; ClickHouse is outside the initial scope.	`post-compile-hooks.json`	No example yet	BtrQL to SQL: ● SQL import: ● Editor: ●

Support Scope

BtrQL focuses on SQL that application developers, data engineers, and DBAs write, review, and tune as part of application and analytics delivery. It is not a database administration console.

In scope

- query composition: `SELECT`, CTEs, joins, grouping, windows, set operations, and supported DML
- reusable source: modules, imports, `val` bindings, relation types, extension methods, macros, and callable entrypoints
- schema objects: tables, views, materialized views, indexes, constraints, and routine definitions where supported
- dialect-aware SQL generation: explicit target dialects, supported native clauses, and clear errors when a construct cannot be emitted

Out of scope

Keep operational work in database tools, migration systems, infrastructure automation, or runbooks:

- backup, restore, retention, and archive jobs
- server, cluster, storage, role, and security-policy administration
- replication topology and queue or broker management
- application runtime execution, scheduling, connection pooling, and migrations
- cost-based query optimization and vendor-specific administration consoles

PostgreSQL Dialect

PostgreSQL is the broadest current target for BtrQL examples, generated SQL checks, and real database execution.

Use this section for PostgreSQL-specific SQL forms that BtrQL exposes directly while keeping the surrounding source checked.

DISTINCT ON

BtrQL SQL

query.btrql

```
users
  .select(region, user_name)
  .distinctOn(region)
  .orderBy(region.asc, user_name.asc)
```

generated/query.sql

```
SELECT DISTINCT ON (region)
  region,
  user_name
FROM users
ORDER BY region ASC, user_name ASC;
```

RETURNING

BtrQL SQL

query.btrql

```
users
  .where(id == 1)
  .update {
    into users
    set(TRUE -> active)
  }
  .returning(id, active)
```

generated/query.sql

```
UPDATE users
SET active = TRUE
WHERE id = 1
RETURNING id, active;
```

Materialized Views

BtrQL SQL

query.btrql

```
materialized view active_user_snapshot =
  users
    .where(active == TRUE)
    .select(id, name)
```

generated/query.sql

```
CREATE MATERIALIZED VIEW active_user_snapshot AS
SELECT
  id,
  name
FROM users
WHERE active = TRUE;
```

Upsert With ON CONFLICT

For PostgreSQL, `mergeInto(...)` lowers to `INSERT... ON CONFLICT` instead of native SQL `MERGE` in the current PostgreSQL package.

The conflict predicate must be a strict equality between the target key and the incoming/source key. Matched assignments are rendered through PostgreSQL's `excluded` row, insert-only forms lower to `DO NOTHING`, and `returning(...)` stays available on the generated DML when the projected columns are valid.

Procedure Bodies

PostgreSQL procedure bodies render as PL/pgSQL blocks. The supported body surface includes local `var` declarations, assignment, `if`/`else`, procedure `case`, `while`, `for (row in query)`, `loop`, bare `return`, nested `call`, and embedded DML.

PostgreSQL procedures do not return values, so `return expr;` is a diagnostic. Bare embedded SELECT statements are also rejected in procedure bodies; use DML or a declaration form that has executable effect. Empty procedure bodies render `NULL;`.

Dialect-specific syntax diagnostics

The PostgreSQL syntax gate reports other-dialect constructs with target-specific wording. Examples:

- `prewhere(...)`, `arrayJoin(...)`, configured table `engine`, `index`, and `indexes` are ClickHouse-specific and unsupported by PostgreSQL.
- `crossJoin(relation)` is the PostgreSQL form; `crossJoin(relation on predicate)` is rejected for PostgreSQL.

These diagnostics are distinct from "not implemented yet" messages: they mean the syntax belongs to another DBMS target, not to the PostgreSQL surface.

Actual novelty

BtrQL facilitates code reuse via modules, relation types, extension methods, extension columns, and macros. These features let you write clean, shared logic that the compiler checks for you, while still generating explicit, predictable SQL.

Extensions draw inspiration from receiver-oriented languages: define a row-shape contract once, then attach multiple reusable relation methods and scalar column computations that work on any compatible relation.

Reusable relation logic

One receiver exposes multiple relation methods and computed columns.

BtrQL SQL

query.btrql

```
using analytics

type OrderInput = [amount: DECIMAL, tax_rate: DECIMAL]

extension OrderInput {
  method taxable =
    self
      .where(amount > 0)
  method withTax =
    self
      .addColumn(tax)
  column tax: DECIMAL = amount * tax_rate
  column gross: DECIMAL = amount + tax
}

view enriched_orders =
  orders
    .taxable()
    .addColumn(tax, gross)
```

generated/postgresql.sql

```
CREATE VIEW "enriched_orders" AS
SELECT
  "order_id",
  "amount",
  "tax_rate",
  "tax",
  "_q5"."amount" + "_q5"."tax" AS "gross"
FROM (
  SELECT
    "order_id",
    "amount",
    "tax_rate",
    "_q3"."amount" * "_q3"."tax_rate" AS "tax"
  FROM (
    SELECT *
    FROM (
      SELECT
        "orders"."order_id",
        "orders"."amount",
        "orders"."tax_rate"
      FROM "analytics"."orders" AS "orders"
    ) AS "_q1"
    WHERE "amount" > 0
  ) AS "_q3"
) AS "_q5";
```

generated/clickhouse.sql

```
CREATE VIEW `enriched_orders` AS
SELECT
  `order_id`,
  `amount`,
  `tax_rate`,
  `tax`,
  `_q5`.`amount` + `_q5`.`tax` AS `gross`
FROM (
  SELECT
    `order_id`,
    `amount`,
    `tax_rate`,
    `_q3`.`amount` * `_q3`.`tax_rate` AS `tax`
  FROM (
    SELECT *
    FROM (
      SELECT
        `orders`.`order_id`,
        `orders`.`amount`,
        `orders`.`tax_rate`
      FROM `analytics`.`orders` AS `orders`
    ) AS `_q1`
    WHERE `amount` > 0
  ) AS `_q3`
) AS `_q5`;
```

PostgreSQL ClickHouse

Relation types and extension columns

A relation type names a reusable row contract. It can drive typed projections, grouped keys, column removal, and extension receivers.

```
type AuditCols = [id: INT, created_at: TIMESTAMP]

type UserShape = AuditCols ++ [name: VARCHAR, active: BOOLEAN]

users
  .select[UserShape]
```

An extension can define multiple relation methods and reusable scalar columns. Extra receiver columns are allowed. A dependency is substituted into the scalar expression but does not become an output unless requested.

```
using analytics

type EngagementInput = [
  posts_viewed: INT,
  comments_written: INT
]

extension EngagementInput {
  method activeOnly =
    self
      .where(is_engaged)
  method withMinimumPosts()(minPosts: INT) =
    self
      .where(posts_viewed >= minPosts)

  column engagement_score: INT = posts_viewed + comments_written * 5
  column is_engaged: BOOLEAN =
    posts_viewed > 10 and comments_written > 0
  column engagement_label =
    case when is_engaged then 'active' else 'inactive' end
}

view engaged_users =
  users
    .activeOnly()
    .withMinimumPosts()(5)
    .addColumn(engagement_score, engagement_label)
```

Both examples compile with the PostgreSQL and ClickHouse C# prototypes when their `analytics` schema metadata supplies the receiver columns. The former `column recipe` and `addColumn[Type]` forms are retained only as migration diagnostics.

Modules and projects

BtrQL modules are file-based. A source file such as `input/reporting/shared.btrql` becomes module `reporting.shared`, and configured source roots decide where module lookup begins.

```
import reporting.shared { base_users -> users_base }
using analytics

view active_users =
  users_base
    .where(active == TRUE)
```

A project ties source files, generated output, target dialect, and schema metadata together.

```
{
  "output-folder": "generated",
  "target-dialect": "postgresql"
}
```

The schema cache stores database metadata used for offline checks: schemas, tables, columns, types, and callable signatures. It does not store table rows. A project build follows imports in dependency order, writes generated SQL, and reports missing modules, duplicate names, import cycles, stale schema references, and diagnostics before deployment.

Post-compile actions

Project compiler integrations should expose enough result data for post-compile work. The main project compiler supports configured post-compile hooks after SQL generation and reports hook diagnostics and artifacts.

The native project APIs currently return compiled modules with source paths, output paths, SQL, diagnostics, output roots, and relation-shape maps after `CompileDirectory(...)`, so callers can attach their own post-compile steps. Those APIs do not currently run configured hook files themselves; their editor build responses expose successful default post-compile status when no hook runner is configured.

Core Ideas

BtrQL reuse is compile-time source organization. Modules, relation types, table-type column lists, extension methods, extension columns, val bindings, compile-time macros, and project compilation keep shared logic in checked BtrQL while generated SQL stays explicit.

Extension methods and extension columns use receiver-oriented reuse: a named row-shape contract owns reusable relation operations and scalar computations that apply to any compatible relation.

Reusable relation logic

This example shows reusable relation types and extension methods called from multiple use sites before SQL lowering.

BtrQL SQL
source.btrql

```
using analytics

val cutoff = 7 + 3

type AuditCols = [id: INT, created_at: TIMESTAMP]

type UserShape = AuditCols ++ [name: VARCHAR, active: BOOLEAN]

extension UserShape {
  method keepRecent =
    self
      .where(id >= cutoff)
}

extension [id: INT] {
  method keepRecentIds =
    self
      .where(id >= cutoff)
}

view recent_users =
  users
    .keepRecent()

view recent_active_users =
  users
    .keepRecent()
    .where(active == TRUE)
    .select(id, name)

view recent_user_ids =
  users
    .keepRecentIds()
    .select(id)

view recent_named_ids =
  users
    .keepRecentIds()
    .where(name != '')
    .select(id)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" },
        { "name": "created_at", "type": "TIMESTAMP" }
      ]
    }
  }
}
```

source.btrql schemas.json
postgresql.sql

```
CREATE VIEW "recent_users" AS
SELECT *
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q1"
WHERE "id" >= 10;

CREATE VIEW "recent_active_users" AS
SELECT
  "id",
  "name"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q3"
WHERE
  "id" >= 10
  AND "active" = TRUE;

CREATE VIEW "recent_user_ids" AS
SELECT "id"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q5"
WHERE "id" >= 10;

CREATE VIEW "recent_named_ids" AS
SELECT "id"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q7"
WHERE
  "id" >= 10
```

```
  AND "name" <> '';
```

clickhouse.sql

```
CREATE VIEW `recent_users` AS
SELECT *
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q1`
WHERE `id` >= 10;

CREATE VIEW `recent_active_users` AS
SELECT
  `id`,
  `name`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q3`
WHERE
  `id` >= 10
  AND `active` = TRUE;

CREATE VIEW `recent_user_ids` AS
SELECT `id`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q5`
WHERE `id` >= 10;

CREATE VIEW `recent_named_ids` AS
SELECT `id`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q7`
WHERE
  `id` >= 10
```

```
  AND `name` <> '';
```

PostgreSQL ClickHouse

Relation types and extension columns

A relation type names a reusable row contract. It can drive typed projections, grouped keys, column removal, and extension receivers.

```
type AuditCols = [id: INT, created_at: TIMESTAMP]

type UserShape = AuditCols ++ [name: VARCHAR, active: BOOLEAN]

users
  .select[UserShape]
```

An extension column names a receiver-bound scalar expression. A relation that satisfies the extension's minimum row contract can use that column in filters or request it by name in `select` and `addColumn`. Dependencies such as `is_engaged` are substituted into later extension-column expressions, but they are not added to the output unless requested.

```
using analytics

type EngagementInput = [
  posts_viewed: INT,
  comments_written: INT
]

extension EngagementInput {
  method activeOnly =
    self
      .where(is_engaged)

  method withMinimumPosts()(minPosts: INT) =
    self
      .where(posts_viewed >= minPosts)

  column engagement_score: INT =
    posts_viewed + comments_written * 5

  column is_engaged: BOOLEAN =
    posts_viewed > 10 and comments_written > 0

  column engagement_label =
    case when is_engaged then 'active' else 'inactive' end
}

view engaged_users =
  users
    .activeOnly()
    .withMinimumPosts()(5)
    .addColumn(engagement_score, engagement_label)
```

This example compiles with the PostgreSQL and ClickHouse C# prototypes when the `analytics.users` schema supplies the receiver columns. The former `column recipe` declaration and `addColumn[Type]` request are removed syntax and produce migration diagnostics.

Table type algebra

An extension method has a symbolic table return type. `T0` is the open type of the receiver (`self`); `T1`, `T2`, and later variables are table-kind parameters in declaration order. A variable is *open*: call-site columns beyond its minimum receiver contract remain present until a projection closes that variable. `ScalarType0`, `ScalarType1`, and later names are schematic scalar slots; when the compiler knows a scalar type, Outline shows the concrete type instead.

Form	Meaning
`A ++ B`	concatenate table shapes, or add the columns in `B`

Form	Meaning
`A -- B`	remove the named, type-compatible columns in `B`
`A && B`	keep the columns shared by both declared table types

Derived return types normally normalize to open variables plus ordered `++` and `--` deltas. Read the deltas in pipeline order. A rename is therefore a removal followed by an addition, and replacing a column's scalar type uses the same pair.

```
receiver        table argument        computed delta         removal delta
   T0       ++       T2         ++ [newColumn: ScalarType1] -- [removedColumn:
ScalarType2]
    └──────────────────────────────── pipeline result ──────────────────────────────┘

=> T0 ++ T2 ++ [newColumn: ScalarType1] -- [removedColumn: ScalarType2]
```

For example, an extension body that joins its second table argument, adds `newColumn`, and removes `removedColumn` produces exactly that return type. `++ T2` means row-shape concatenation; it does not mean SQL set union.

Return-type diagrams for table methods

The diagrams below cover every table method classified by extension-method return inference. Methods with the same shape rule are grouped together. `Tn` means whichever open table variable is on the left of the call, `R` is a right-side relation result, and `R?` is the nullable-relation constructor defined in the join section below.

Shape-preserving methods

Filtering, ordering, validation, paging, aliases, native clauses, and DML commands preserve the incoming table type. DML keeps this type until a `returning` step projects another shape.

```
Tn ── where | qualify | orderBy | distinct | distinctOn | rowShape
   ── limit | offset | as
   ── prewhere | sample | limitBy | settings | format
   ── insert | update | delete | deleteFrom | mergeInto ──────────────▶ Tn
```

Set methods

`union`, `unionAll`, `intersect`, and `except` return the type of their left operand. They validate the right operand but do not concatenate it.

```
self:T0  ── union | unionAll | intersect | except (other:T1) ──▶ T0
other:T1 ── union | unionAll | intersect | except (self:T0)  ──▶ T1
```

Column delta methods

`addColumn` appends new columns and replaces a collision with an ordered remove/add pair. `removeColumn(...)` and `removeColumn[Type]` subtract columns. ClickHouse `arrayJoin` and `leftArrayJoin` retain the input columns and append each named element alias.

```
Tn ── addColumn(expr -> newColumn) ──────────────▶ Tn ++ [newColumn: ScalarType0]
Tn ── removeColumn(oldColumn) ───────────────────▶ Tn -- [oldColumn: ScalarType1]
Tn ── addColumn(expr -> existingColumn) ─────────▶ Tn -- [existingColumn: OldType]
                                                    ++ [existingColumn: NewType]
Tn ── arrayJoin(items -> item) ──────────────────▶ Tn ++ [item: ElementType]
Tn ── leftArrayJoin(items -> item) ──────────────▶ Tn ++ [item: ElementType]
```

Projection methods

Projection first decides which input columns survive under their original names. The removed columns are the complement of the preserved columns; they are not a second set of changes.

```
Tn ── select / select[Type] / returning / returningWith
   ── (preserve existing columns under their original names)
   ──▶ Tn -- [columns not preserved under their original names]
       ≡ [preserved columns with their original names and types]
```

A computed projection output is added after that closing projection.

```
Tn ── select / returning / returningWith (..., expression -> result)
   ──▶ Tn -- [columns not preserved under their original names]
           ++ [result: ExpressionType]
```

Rename-only algebra

A direct-column alias is a rename. Its type rule is independent of the projection or grouping operation that contains it.

```
Tn ── sourceColumn -> renamedColumn ──▶ Tn -- [sourceColumn: SourceType]
                                            ++ [renamedColumn: SourceType]
```

Inside a projection, `sourceColumn` is already among the columns not preserved under their original names. The normalized projection therefore removes it once and adds `renamedColumn`; it does not apply a second removal.

Grouping methods

Grouping preserves direct keys under their original names, removes non-keys, and adds aggregate results. If a key is aliased, apply the rename-only rule above to that key separately.

```
Tn ── group { by(key columns) aggregate(...) } / group { by[Type] ... }
   ──► Tn -- [every non-key column]
           ++ [aggregate result columns: ResultTypes]
```

Join and APPLY methods

Each rule below is independent. In a symbolic relation result, `X?` means "the same columns as `X`, with every scalar made nullable." A bare `X` keeps the columns' existing nullability.

```
innerJoin / crossJoin : T0 ++ T1
leftJoin              : T0 ++ T1?
rightJoin             : T0? ++ T1
fullJoin              : T0? ++ T1?

crossApply(R)          : T0 ++ R
outerApply(R)          : T0 ++ R?
```

An inner or cross join keeps both sides as they are. A left join can produce a left row with no matching right row, so only `T1` becomes nullable. A right join is the mirror image; a full join may lack either side. Likewise, `outerApply` retains a left row even when its dependent result has no row, so the columns contributed by `R` become nullable.

The compiler and Outline now preserve this as a first-class symbolic relation type: an extension method containing a left join renders `T0 ++ T1?`, not a remove/add replacement for every column in `T1`. Instantiating `T1?` makes all bound `T1` columns nullable, including extra call-site columns beyond the method's minimum contract. Nested extension calls preserve the wrapper during type substitution.

`Tn?` is result-type notation, not BtrQL source syntax. PostgreSQL renders the resulting nullable *scalar* details as forms such as `INTEGER?`; ClickHouse uses forms such as `Nullable(Int32)`. A declared table function has a closed, concrete return shape and no open `Tn` variable, so its result continues to show per-column scalar types. If table functions later gain table-kind parameters, the same `Tn?` relation algebra can apply to them. There is no established `!` suffix; a bare relation variable means "preserve current nullability."

`jsonTable` and `rowsFromWithOrdinality` contribute their declared result columns through the APPLY rule. `outerApply` itself is available only in dialects that support that operation.

Extension-method composition

Think of an extension method as a reusable row-shape transformation:

1. `T0` stands for the actual relation to the left of the method call.
2. `T1`, `T2`, and later variables stand for table arguments in declaration order.
3. A call binds those placeholders to the receiver and arguments at that point in the pipeline.
4. The method's additions and removals are applied, and the next call continues from that result.

For a simple table-argument call, the binding is:

```
call:    current_users.join_orders(current_orders)()
bind:    T0 = current_users, T1 = current_orders
result:  current_users ++ current_orders
```

Here is a complete example that compiles with both C# prototypes:

```
using analytics

extension [user_id: INTEGER, user_name: VARCHAR, region: VARCHAR] {
  method only_region()(wanted_region: VARCHAR) =
    self
      .where(region == wanted_region)

  method join_orders(other: [
    order_id: INTEGER,
    buyer_id: INTEGER,
    amount: INTEGER,
    status: VARCHAR
  ]) =
    self
      .innerJoin(other on user_id == buyer_id)

  method reshaped_join(other: [
    order_id: INTEGER,
    buyer_id: INTEGER,
    amount: INTEGER,
    status: VARCHAR
  ]) =
    self
      .removeColumn(region)
      .crossJoin(
        other
          .removeColumn(buyer_id)
          .addColumn(amount + 1 -> adjusted_amount)
      )
}

view regional_orders =
  users
    .only_region()('EU')
    .join_orders(orders)()

view public_order_rows =
  users
    .reshaped_join(orders)()
```

Read its inferred shapes in ordinary language:

- `only_region` filters rows and keeps `T0` unchanged.
- `join_orders` appends the table argument, producing `T0 ++ T1`.
- `reshaped_join` keeps each change attached to its source: `T0 -- [region: VARCHAR] ++ T1 -- [buyer_id: INTEGER] ++ [adjusted_amount: INTEGER]`.

`regional_orders` shows call-site composition: filter `users`, then join `orders`. The first call does not change the row shape, so the second call still receives the complete user relation as its `T0`. `public_order_rows`

invokes the origin-preserving reshape so that example is compiled too.

An extension column is the scalar special case: requesting `column score =...` has symbolic result `T0 ++ [score: ConcreteScalarType]`.

Modules and projects

BtrQL modules are file-based. A source file such as `input/reporting/shared.btrql` becomes module `reporting.shared`, and configured source roots decide where module lookup begins.

```
import reporting.shared { base_users -> users_base }
using analytics

view active_users =
  users_base
    .where(active == TRUE)
```

A project ties source files, generated output, target dialect, and schema metadata together.

```
{
  "output-folder": "generated",
  "target-dialect": "postgresql"
}
```

The schema cache stores database metadata used for offline checks: schemas, tables, columns, types, and callable signatures. It does not store table rows. A project build follows imports in dependency order, writes generated SQL, and reports missing modules, duplicate names, import cycles, stale schema references, and diagnostics before deployment.

Post-compile actions

Project compiler integrations should expose enough result data for post-compile work. The main project compiler supports configured post-compile hooks after SQL generation and reports hook diagnostics and artifacts.

The native project APIs currently return compiled modules with source paths, output paths, SQL, diagnostics, output roots, and relation-shape maps after `CompileDirectory(...)`, so callers can attach their own post-compile steps. Those APIs do not currently run configured hook files themselves; their editor build responses expose successful default post-compile status when no hook runner is configured.

Builtin functions

BtrQL recognizes common function families so calls can be checked and emitted in the selected SQL dialect. Function names stay close to SQL; the compiler only changes spelling when a dialect needs a different native form.

Function families

Family	Typical calls	Notes
Text	`lower`, `upper`, `length`, `concat`	String shaping and comparisons.
Numeric	`abs`, `round`, `floor`, `ceiling`	Arithmetic helpers and numeric coercions.
Date and time	`dateAdd`, `dateDiff`, `now`	Dialect spelling varies the most here.
Aggregate	`count`, `sum`, `avg`, `min`, `max`	Works with `group` and aggregate filters.
Window	`row_number`, `rank`, `dense_rank`	Used with `over(...)`.
JSON	`jsonEach`, `jsonTable`, JSON accessors	Dialect support is intentionally uneven.
Table functions	`json_each(...)`, `rows_from(...)`	Produce relation sources, not scalar values.

Use the dialect pages under BtrQL Syntax when a function depends on a database-specific feature.

Dialect notes

The same BtrQL call can be supported, partially supported, or unavailable depending on the target dialect. Unsupported calls fail before SQL is emitted instead of silently producing an approximate query.

Walkthroughs

Use walkthroughs after the syntax pages. Each example keeps BtrQL source on the left and generated SQL on the right, so you can see how several constructs work together.

Recommended order:

1. Query walkthroughs for projection, filtering, ordering, grouping, joins, set operations, windows, and shared helper patterns.
2. Modules and projects for multi-file source and project-level compilation.
3. Cross-dialect workflows for syntax that changes shape between SQL targets.

Short single-construct examples live in Syntax overview. Diagnostic examples are covered in Errors.

Query walkthroughs

BtrQL source patterns that combine multiple syntax elements with generated SQL.

PostgreSQL ClickHouse

Query Pipeline

This example shows a bare top-level executable statement with a compact where -> addColumn -> orderBy -> limit pipeline.

BtrQL SQL

source.btrql

```
using analytics

users
  .where(active == TRUE)
  .addColumn(active -> active_flag)
  .orderBy(created_at.desc)
  .limit(10)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" },
        { "name": "created_at", "type": "TIMESTAMP" }
      ]
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
SELECT
  "users"."id",
  "users"."name",
  "users"."active",
  "users"."created_at",
  "users"."active" AS "active_flag"
FROM "analytics"."users" AS "users"
WHERE "users"."active" = TRUE
ORDER BY "created_at" DESC
FETCH NEXT 10 ROWS ONLY;
```

clickhouse.sql

```
SELECT
  `users`.`id`,
  `users`.`name`,
  `users`.`active`,
  `users`.`created_at`,
  `users`.`active` AS `active_flag`
FROM `analytics`.`users` AS `users`
WHERE `users`.`active` = TRUE
ORDER BY `created_at` DESC
LIMIT 10;
```

PostgreSQL | ClickHouse

Joins, WITH Bindings, And Set Operations

This example shows statement-local with {... } bindings feeding a unionAll(...) composition and then a joined query with an explicit projection and orderBy(...).

BtrQL SQL

source.btrql

```
using analytics

with {
  active_users =
    users
      .where(active == TRUE)
      .select(id, name)
  inactive_users =
    users
      .where(active == FALSE)
      .select(id, name)
  selected_users =
    active_users
      .unionAll(inactive_users)
}
selected_users
  .innerJoin(orders on selected_users.id == orders.user_id)
  .select(
    selected_users.id -> user_id,
    selected_users.name,
    orders.order_id,
    orders.amount
  )
  .orderBy(orders.amount.desc)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ],
      "orders": [
        { "name": "order_id", "type": "INTEGER" },
        { "name": "user_id", "type": "INTEGER" },
        { "name": "amount", "type": "DECIMAL" }
      ]
    }
  }
}
```

source.btrql schemas.json
postgresql.sql

```
WITH "active_users" AS (
  SELECT
    "users"."id",
    "users"."name"
  FROM "analytics"."users" AS "users"
  WHERE "users"."active" = TRUE
),

"inactive_users" AS (
  SELECT
    "users"."id",
    "users"."name"
  FROM "analytics"."users" AS "users"
  WHERE "users"."active" = FALSE
),

"selected_users" AS (
  SELECT *
  FROM "active_users"
  UNION ALL
  SELECT *
  FROM "inactive_users"
)

SELECT
  "selected_users"."id" AS "user_id",
  "selected_users"."name",
  "orders"."order_id",
  "orders"."amount"
FROM "selected_users"
INNER JOIN "analytics"."orders" AS "orders"
  ON "selected_users"."id" = "orders"."user_id"
ORDER BY "orders"."amount" DESC;
```

clickhouse.sql

```
WITH `active_users` AS (
  SELECT
    `users`.`id`,
    `users`.`name`
  FROM `analytics`.`users` AS `users`
  WHERE `users`.`active` = TRUE
),

`inactive_users` AS (
  SELECT
    `users`.`id`,
    `users`.`name`
  FROM `analytics`.`users` AS `users`
  WHERE `users`.`active` = FALSE
),

`selected_users` AS (
  SELECT *
  FROM `active_users`
  UNION ALL
  SELECT *
  FROM `inactive_users`
)

SELECT
  `selected_users`.`id` AS `user_id`,
  `selected_users`.`name`,
  `orders`.`order_id`,
  `orders`.`amount`
FROM `selected_users`
INNER JOIN `analytics`.`orders` AS `orders`
  ON `selected_users`.`id` = `orders`.`user_id`
ORDER BY `orders`.`amount` DESC;
```

PostgreSQL ClickHouse

UNION Column Order

This example shows that BtrQL aligns a unionAll right-hand query by column name before emitting SQL, so positional SQL set operations keep the left-hand column order.

BtrQL SQL

source.btrql

```
using analytics

view reordered_union =
  users
    .select(id, name)
    .unionAll(
      users
        .select(name, id)
    )
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
CREATE VIEW "reordered_union" AS
SELECT
  "users"."id",
  "users"."name"
FROM "analytics"."users" AS "users"
UNION ALL
SELECT
  "id",
  "name"
FROM (
  SELECT
    "users"."name",
    "users"."id"
  FROM "analytics"."users" AS "users"
) AS "_q1";
```

clickhouse.sql

```
CREATE VIEW `reordered_union` AS
SELECT
  `users`.`id`,
  `users`.`name`
FROM `analytics`.`users` AS `users`
UNION ALL
SELECT
  `id`,
  `name`
FROM (
  SELECT
    `users`.`name`,
    `users`.`id`
  FROM `analytics`.`users` AS `users`
) AS `_q1`;
```

PostgreSQL ClickHouse

Grouped Queries, Aggregates, And Builtins

This example shows grouped projections, aggregate builtin functions, having(...), and an orderBy(...) applied after aggregation.

BtrQL SQL

source.btrql

```
using analytics

with {
  normalized_orders =
    orders
      .select(
        order_id,
        lower(coalesce(status, 'unknown')) -> normalized_status,
        round(amount) -> rounded_amount,
        amount
      )
}
normalized_orders
  .group {
    by(normalized_status)
    aggregate(
      count(order_id) -> order_count,
      sum(rounded_amount) -> rounded_total_amount,
      avg(amount) -> average_amount,
      min(amount) -> minimum_amount,
      max(amount) -> maximum_amount
    )
    having(count(order_id) > 1)
  }
  .orderBy(normalized_status.asc)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "orders": [
        { "name": "order_id", "type": "INTEGER" },
        { "name": "status", "type": "VARCHAR" },
        { "name": "amount", "type": "DECIMAL" }
      ]
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
WITH "normalized_orders" AS (
  SELECT
    "orders"."order_id",
    LOWER(COALESCE("orders"."status", 'unknown')) AS "normalized_status",
    ROUND("orders"."amount") AS "rounded_amount",
    "orders"."amount"
  FROM "analytics"."orders" AS "orders"
)

SELECT
  "normalized_status",
  COUNT("order_id") AS "order_count",
  SUM("rounded_amount") AS "rounded_total_amount",
  AVG("amount") AS "average_amount",
  MIN("amount") AS "minimum_amount",
  MAX("amount") AS "maximum_amount"
FROM "normalized_orders"
GROUP BY "normalized_status"
HAVING COUNT("order_id") > 1
ORDER BY "normalized_status";
```

clickhouse.sql

```
WITH `normalized_orders` AS (
  SELECT
    `orders`.`order_id`,
    LOWER(COALESCE(`orders`.`status`, 'unknown')) AS `normalized_status`,
    ROUND(`orders`.`amount`) AS `rounded_amount`,
    `orders`.`amount`
  FROM `analytics`.`orders` AS `orders`
)

SELECT
  `normalized_status`,
  COUNT(`order_id`) AS `order_count`,
  SUM(`rounded_amount`) AS `rounded_total_amount`,
  AVG(`amount`) AS `average_amount`,
  MIN(`amount`) AS `minimum_amount`,
  MAX(`amount`) AS `maximum_amount`
FROM `normalized_orders`
GROUP BY `normalized_status`
HAVING COUNT(`order_id`) > 1
ORDER BY `normalized_status`;
```

PostgreSQL ClickHouse

Window And QUALIFY

This example shows a statement-local with {... } CTE, a window expression, distinct, orderBy, and qualify. PostgreSQL lowers the post-window filter through a derived-table WHERE.

BtrQL SQL

source.btrql

```
using analytics

with {
  active_users =
    users
      .where(active == TRUE)
}
active_users
  .select(
    id,
    row_number().over {
      partitionBy(id)
      orderBy(id.desc)
    } -> rn
  )
  .distinct
  .orderBy(rn.desc)
  .qualify(rn > 0)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    }
  }
}
```

source.btrql schemas.json

postgresql.sql

```
SELECT * FROM (WITH "active_users" AS (
  SELECT
    "users"."id",
    "users"."active"
  FROM "analytics"."users" AS "users"
  WHERE "users"."active" = TRUE
)

SELECT DISTINCT
  "id",
  ROW_NUMBER() OVER (
    PARTITION BY "id"
    ORDER BY "id" DESC
  ) AS "rn"
FROM "active_users") AS "_q1"
WHERE "rn" > 0
ORDER BY "rn" DESC;
```

clickhouse.error.txt

```
Dialect 'clickhouse' does not support QUALIFY: Current ClickHouse lowering rejects
QUALIFY.
```

PostgreSQL ClickHouse

Schema Using And Callables

This example shows how using analytics loads both tables and callable signatures.

BtrQL SQL

source.btrql

```
using analytics

view active_users =
  users
    .where(is_active(active))

call ping(1)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    },
    "callableSignatures": {
      "is_active": "(BOOLEAN) -> BOOLEAN",
      "ping": "(INTEGER)"
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
CREATE VIEW "active_users" AS
SELECT
  "users"."id",
  "users"."name",
  "users"."active"
FROM "analytics"."users" AS "users"
WHERE "is_active"("users"."active");

CALL "ping"(1);
```

clickhouse.error.txt

```
Dialect 'clickhouse' does not support procedure calls: Procedure calls are not enabled
for ClickHouse in the current support surface.
```

PostgreSQL | ClickHouse

Relation Type And Extension Reuse

This example shows reusable relation types and extension methods called from multiple use sites before SQL lowering.

BtrQL | SQL

source.btrql

```
using analytics

val cutoff = 7 + 3

type AuditCols = [id: INT, created_at: TIMESTAMP]

type UserShape = AuditCols ++ [name: VARCHAR, active: BOOLEAN]

extension UserShape {
  method keepRecent =
    self
      .where(id >= cutoff)
}

extension [id: INT] {
  method keepRecentIds =
    self
      .where(id >= cutoff)
}

view recent_users =
  users
    .keepRecent()

view recent_active_users =
  users
    .keepRecent()
    .where(active == TRUE)
    .select(id, name)

view recent_user_ids =
  users
    .keepRecentIds()
    .select(id)

view recent_named_ids =
  users
    .keepRecentIds()
    .where(name != '')
    .select(id)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" },
        { "name": "created_at", "type": "TIMESTAMP" }
      ]
    }
  }
}
```

source.btrql schemas.json
postgresql.sql

```
CREATE VIEW "recent_users" AS
SELECT *
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q1"
WHERE "id" >= 10;

CREATE VIEW "recent_active_users" AS
SELECT
  "id",
  "name"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q3"
WHERE
  "id" >= 10
  AND "active" = TRUE;

CREATE VIEW "recent_user_ids" AS
SELECT "id"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q5"
WHERE "id" >= 10;

CREATE VIEW "recent_named_ids" AS
SELECT "id"
FROM (
  SELECT
    "users"."id",
    "users"."name",
    "users"."active",
    "users"."created_at"
  FROM "analytics"."users" AS "users"
) AS "_q7"
WHERE
  "id" >= 10
```

```
  AND "name" <> '';
```

clickhouse.sql

```
CREATE VIEW `recent_users` AS
SELECT *
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q1`
WHERE `id` >= 10;

CREATE VIEW `recent_active_users` AS
SELECT
  `id`,
  `name`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q3`
WHERE
  `id` >= 10
  AND `active` = TRUE;

CREATE VIEW `recent_user_ids` AS
SELECT `id`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q5`
WHERE `id` >= 10;

CREATE VIEW `recent_named_ids` AS
SELECT `id`
FROM (
  SELECT
    `users`.`id`,
    `users`.`name`,
    `users`.`active`,
    `users`.`created_at`
  FROM `analytics`.`users` AS `users`
) AS `_q7`
WHERE
  `id` >= 10
```

```
  AND `name` <> '';
```

PostgreSQL ClickHouse

Modules and projects

Multi-file source and project-level compilation examples.

PostgreSQL ClickHouse

Module Imports

This example shows qualified, unqualified, and aliased access to the same imported module.

BtrQL SQL

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    }
  }
}
```

postgresql.sql

```
CREATE VIEW "copied_qualified" AS
SELECT "reporting_shared_base"."id"
FROM "reporting"."reporting_shared_base" AS "reporting_shared_base";

CREATE VIEW "copied_unqualified" AS
SELECT "reporting_shared_base"."id"
FROM "reporting"."reporting_shared_base" AS "reporting_shared_base";

CREATE VIEW "copied_aliased" AS
SELECT "reporting_shared_base"."id"
FROM "reporting"."reporting_shared_base" AS "reporting_shared_base";
```

clickhouse.sql

```
CREATE VIEW `copied_qualified` AS
SELECT `base`.`id`
FROM `base` AS `base`;

CREATE VIEW `copied_unqualified` AS
SELECT `base`.`id`
FROM `base` AS `base`;

CREATE VIEW `copied_aliased` AS
SELECT `base`.`id`
FROM `base` AS `base`;
```

PostgreSQL ClickHouse

Project Compilation

This example shows a minimal multi-file project compiled through project.json.

BtrQL SQL

project/project.json

```
{
  "output-folder": "generated",
  "target-dialect": "postgresql",
  "schema-cache-path": "schemas.json"
}
```

project/schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    },
    "callableSignatures": {
      "ping": "(INTEGER)"
    }
  }
}
```

project/src/main.btrql

```
using analytics
import utils.testDependency { active_users -> imported_active }

val cutoff = 10

view recent_users =
  imported_active
    .where(id >= cutoff)
    .select(id, name)
```

project/src/utils/testDependency.btrql

```
using analytics

view active_users =
  users
    .where(active == TRUE)
```

project/project.json | project/schemas.json | project/src/main.btrql | project/src/utils/testDependency.btrql
expected/postgresql/main.sql

```
CREATE VIEW "recent_users" AS
SELECT
  "active_users"."id",
  "active_users"."name"
FROM "active_users" AS "active_users"
WHERE "active_users"."id" >= 10;
```

expected/postgresql/utils/testDependency.sql

```
CREATE VIEW "active_users" AS
SELECT
  "users"."id",
  "users"."name",
  "users"."active"
FROM "analytics"."users" AS "users"
WHERE "users"."active" = TRUE;
```

expected/clickhouse/main.sql

```
CREATE VIEW `recent_users` AS
SELECT
  `active_users`.`id`,
  `active_users`.`name`
FROM `active_users` AS `active_users`
WHERE `active_users`.`id` >= 10;
```

expected/clickhouse/utils/testDependency.sql

```
CREATE VIEW `active_users` AS
SELECT
  `users`.`id`,
  `users`.`name`,
  `users`.`active`
FROM `analytics`.`users` AS `users`
WHERE `users`.`active` = TRUE;
```

main.sql | utils/testDependency.sql | main.sql | utils/testDependency.sql

Cross-dialect workflows

Feature workflows across multiple SQL targets.

PostgreSQL | ClickHouse

RETURNING Across Dialects

This example shows a single UPDATE... RETURNING shape across the current public dialect set.

BtrQL | SQL

source.btrql

```
using analytics

users
  .where(active == TRUE)
  .update {
    into users
    set(FALSE -> active)
  }
  .returning(id, active)
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
UPDATE "analytics"."users" SET
  "active" = FALSE
WHERE "active" = TRUE
RETURNING "id", "active";
```

clickhouse.error.txt

```
Dialect 'clickhouse' does not support RETURNING / OUTPUT: Current ClickHouse lowering
rejects RETURNING.
```

PostgreSQL | ClickHouse

Materialized Views Across Dialects

This example shows the current PostgreSQL materialized-view path and the explicit compile-time unsupported diagnostics for the rest of the current public dialect set.

BtrQL | SQL

source.btrql

```
using analytics

materialized view active_snapshot =
  users
    .where(active == TRUE)
    .select(id, name)

refresh materialized view active_snapshot

drop materialized view active_snapshot
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "id", "type": "INTEGER" },
        { "name": "name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" }
      ]
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
CREATE MATERIALIZED VIEW "active_snapshot" AS
SELECT
  "users"."id",
  "users"."name"
FROM "analytics"."users" AS "users"
WHERE "users"."active" = TRUE;

REFRESH MATERIALIZED VIEW "active_snapshot";

DROP MATERIALIZED VIEW "active_snapshot";
```

clickhouse.error.txt

```
Dialect 'clickhouse' does not support materialized views: Current ClickHouse lowering
rejects materialized view statements.
```

PostgreSQL ClickHouse

JSON_TABLE Projection

This example documents the public jsonTable(...) surface for the currently promoted subset.

BtrQL SQL
source.btrql

```
jsonTable('[{"status":"open"}]', '$[*]', status, TEXT, '$.status')
  .select(status)
```

postgresql.sql

```
SELECT "status"
FROM JSON_TABLE('[{"status":"open"}]', '$[*]' COLUMNS("status" TEXT PATH '$.status'));
```

mergeInto Upsert

This example documents the shared public mergeInto(...) surface that backs the promoted PostgreSQL upsert subsets.

BtrQL SQL
source.btrql

```
using analytics

user_import
  .mergeInto(users) {
    on(users.user_id == user_import.user_id)
    whenMatchedUpdate(
      user_import.user_name -> user_name,
      user_import.active -> active,
      user_import.region -> region
    )
    whenNotMatchedInsert(user_id, user_name, active, region)
  }
```

schemas.json

```
{
  "analytics": {
    "tables": {
      "users": [
        { "name": "user_id", "type": "INTEGER" },
        { "name": "user_name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" },
        { "name": "region", "type": "VARCHAR" }
      ],
      "user_import": [
        { "name": "user_id", "type": "INTEGER" },
        { "name": "user_name", "type": "VARCHAR" },
        { "name": "active", "type": "BOOLEAN" },
        { "name": "region", "type": "VARCHAR" }
      ]
    }
  }
}
```

source.btrql | schemas.json

postgresql.sql

```
INSERT INTO "analytics"."users" ("user_id", "user_name", "active", "region")
SELECT
  "user_import"."user_id",
  "user_import"."user_name",
  "user_import"."active",
  "user_import"."region"
FROM "analytics"."user_import" AS "user_import"
ON CONFLICT ("user_id") DO UPDATE
  SET
    "user_name" = "excluded"."user_name",
    "active" = "excluded"."active",
    "region" = "excluded"."region";
```

Errors

BtrQL checks source before emitting SQL. Errors point at the construct that failed and stop generation for that file or statement.

Parse errors

Parse errors mean the source shape is incomplete or malformed. Common cases:

- `users.`: a pipeline step name is missing after the dot.
- `orders.innerJoin(users)`: a join needs a source and an `on` predicate.
- `view broken users.select(id)`: object definitions need `=` before the relation body.
- `table users [id: INTEGER, name]`: every declared column needs a type.
- `import shared { exported as local }`: import aliases use `exported -> local`.

Fix the local syntax first; later name and type checks depend on the parsed shape.

Name and type errors

Name and type errors mean the source parses, but a referenced object, column, callable, argument, or row shape does not match the available metadata.

Typical fixes:

- qualify a relation when multiple relation instances make an extension-column reference ambiguous
- alias one imported extension member with `import module { name -> local_name }` when modules export the same applicable name
- correct a column name before it is used in `select`, `where`, `group`, joins, or `returning`
- break a reported extension-column dependency cycle; the diagnostic prints the complete cycle
- move a sequential alias reference after the expression that declares it
- give every final `select` or `addColumn` output a unique name
- resolve join output collisions with explicit projection aliases
- align extension receiver, table-kind arguments, scalar-kind arguments, and optional result annotations with the derived types
- keep non-grouped columns out of grouped projections unless they are grouped or aggregated
- update cached schema metadata when the database shape changes

Removed recipe syntax

The PostgreSQL and ClickHouse C# prototypes reject `column recipe` and `addColumn[Type]` with targeted migration errors. Move each reusable scalar expression into an extension member such as `column tax: DECIMAL = amount * tax_rate`, then request it with `orders.addColumn(tax)`. A written column type is optional, but if present it must equal the normalized inferred scalar type.

Dialect capability errors

Dialect errors mean the BtrQL source is valid, but the selected SQL target cannot emit that construct. Use the Feature Matrix to check support, then choose one of three options:

- rewrite the source using a supported construct
- switch the target dialect
- keep that statement in dialect-native SQL until BtrQL supports the shape you need

ClickHouse syntax

Array join aliases

ClickHouse `arrayJoin` and `leftArrayJoin` accept `expr -> alias` to name the flattened value.

BtrQL SQL
query.btrql

```
events
  .arrayJoin(tags -> tag)
  .select(event_id, tag)
```

generated/query.sql

```
SELECT
  event_id,
  tag
FROM events array JOIN tags AS tag;
```

`prewhere`

`prewhere(...)` emits ClickHouse PREWHERE before the normal WHERE stage.

BtrQL SQL
query.btrql

```
orders
  .prewhere(active == TRUE)
  .where(amount > 0)
  .select(order_id)
```

generated/query.sql

```
SELECT
    order_id
FROM orders PREWHERE active = TRUE
WHERE amount > 0;
```

`sample`

`sample(...)` emits ClickHouse SAMPLE.

BtrQL SQL

query.btrql

```
orders
  .sample(0.25)
  .select(order_id)
```

generated/query.sql

```
SELECT
    order_id
FROM orders SAMPLE 0.25;
```

`limitBy`

`limitBy(limit, keys...)` emits ClickHouse per-key limiting.

BtrQL SQL
query.btrql

```
orders
  .orderBy(created_at.desc)
  .limitBy(3, status)
  .select(status, order_id)
```

generated/query.sql

```
SELECT
    status,
    order_id
FROM orders ORDER BY created_at DESC LIMIT 3 BY status;
```

`arrayJoin`

`arrayJoin` and `leftArrayJoin` flatten array expressions.

BtrQL SQL
query.btrql

```
events
  .arrayJoin(tags -> tag)
  .select(event_id, tag)
```

generated/query.sql

```
SELECT
  event_id,
  tag
FROM events array JOIN tags AS tag;
```

`settings`

`settings(...)` appends ClickHouse query settings.

BtrQL | SQL

query.btrql

```
orders
  .select(order_id)
  .settings(max_threads = 1)
```

generated/query.sql

```
SELECT
    order_id
FROM orders SETTINGS max_threads = 1;
```

`format`

`format(...)` appends a ClickHouse output format.

BtrQL | SQL

query.btrql

```
orders
  .select(order_id)
  .format(JSONEachRow)
```

generated/query.sql

```
SELECT
    order_id
FROM orders FORMAT JSONEachRow;
```

Query pipeline

Relation source

A query pipeline begins from a table, view, imported module symbol, or named binding.

BtrQL SQL
query.btrql

```
import reporting.shared

reporting.shared.users
  .select(id, name)
```

generated/query.sql

```
SELECT
  id,
  name
FROM reporting.shared.users;
```

Subquery source

A parenthesized BtrQL query can be used as a source for later steps.

BtrQL SQL
query.btrql

```
(
  users
    .where(active == TRUE)
    .select(id, name)
).as(active_users)
  .select(active_users.id)
```

generated/query.sql

```
SELECT active_users.id FROM (SELECT
  id,
  name
FROM users
WHERE active = TRUE) AS active_users;
```

Single-row relation

A bracket literal creates one inline row for query composition.

BtrQL SQL
query.btrql

```
[id -> 1, name -> 'Ada']
  .select(id, name)
```

generated/query.sql

```
SELECT
  1 AS id,
  'Ada' AS name;
```

Table literal

A `<[columns values]>` literal creates multiple inline rows with a fixed column header.

BtrQL SQL
query.btrql

```
<[id, name 1, 'Ada' 2, 'Grace']>
  .select(id, name)
```

generated/query.sql

```
SELECT * FROM (VALUES (1, 'Ada'), (2, 'Grace')) AS v (id, name);
```

as

`.as(alias)` assigns a relation alias for qualified references.

BtrQL SQL
query.btrql

```
users.as(u)
  .select(u.id, u.name)
```

generated/query.sql

```
SELECT
  u.id,
  u.name
FROM users AS u;
```

Qualified names

`alias.column` reads a field from a named source alias.

BtrQL | SQL

query.btrql

```
users.as(u)
  .innerJoin(orders.as(o) on u.id == o.user_id)
  .select(u.id, o.order_id)
```

generated/query.sql

```
SELECT
  u.id,
  o.order_id
FROM users AS u INNER JOIN orders AS o ON u.id = o.user_id;
```

`select`

`select(...)` projects checked columns and expressions.

BtrQL | SQL

query.btrql

```
users
  .select(id, upper(name) -> display_name)
```

generated/query.sql

```
SELECT
  id,
  UPPER(name) AS display_name
FROM users;
```

`where`

`where(...)` filters rows with a checked boolean expression.

BtrQL SQL
query.btrql

```
users
  .where(active == TRUE)
  .select(id, name)
```

generated/query.sql

```
SELECT
  id,
  name
FROM users
WHERE active = TRUE;
```

`orderBy`

`orderBy(...)` sorts by checked expressions.

BtrQL SQL
query.btrql

```
users
  .where(active == TRUE)
  .orderBy(created_at.desc, id.asc)
  .select(id)
```

generated/query.sql

```
SELECT id FROM users
WHERE active = TRUE
ORDER BY created_at DESC, id ASC;
```

`limit`

`limit(...)` caps the result set with target-dialect syntax.

BtrQL SQL
query.btrql

```
users
  .orderBy(id.asc)
  .limit(25)
```

generated/query.sql

```
SELECT * FROM users
ORDER BY id ASC LIMIT 25;
```

`offset`

`offset(...)` skips rows after sorting.

BtrQL | SQL
query.btrql

```
users
  .orderBy(id.asc)
  .offset(50)
  .limit(25)
```

generated/query.sql

```
SELECT * FROM users
ORDER BY id ASC OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

Literals

String, numeric, boolean, and null literals can appear in scalar expressions.

BtrQL | SQL
query.btrql

```
users
  .select(id, 'active' -> status_label, 42 -> score)
```

generated/query.sql

```
SELECT
  id,
  'active' AS status_label,
  42 AS score
FROM users;
```

Identifiers

A bare identifier refers to a column or named scalar in scope.

BtrQL SQL
query.btrql

```
users
  .where(active == TRUE)
  .select(id)
```

generated/query.sql

```
SELECT id FROM users
WHERE active = TRUE;
```

Unary operators

Unary operators apply to one scalar expression.

BtrQL SQL
query.btrql

```
orders
  .select(order_id, -amount -> reversal_amount)
```

generated/query.sql

```
SELECT
  order_id,
  -amount AS reversal_amount
FROM orders;
```

Binary operators

Binary operators include comparisons, boolean operators, arithmetic, LIKE, and concatenation-like forms supported by the language.

BtrQL SQL
query.btrql

```
orders
  .where(amount >= 100 and status == 'paid')
  .select(order_id)
```

generated/query.sql

```
SELECT order_id FROM orders
WHERE amount >= 100 AND status = 'paid';
```

Function calls

`name(args...)` calls a scalar builtin, imported routine, or user-defined scalar function that is visible in the current source context.

BtrQL | SQL

query.btrql

```
users
  .select(id, lower(name) -> normalized_name)
```

generated/query.sql

```
SELECT
  id,
  LOWER(name) AS normalized_name
FROM users;
```

`case when`

`case when ... then ... else ... end` returns a scalar value from checked conditions.

BtrQL | SQL

query.btrql

```
users
  .select(
    id,
    case when active then 'active' else 'inactive' end -> active_label
  )
```

generated/query.sql

```
SELECT
  id,
  CASE WHEN active THEN 'active' ELSE 'inactive' END AS active_label
FROM users;
```

Sort direction

Sort expressions use `.asc` or `.desc` suffixes on checked scalar expressions.

BtrQL SQL

query.btrql

```
created_at.desc

users
  .orderBy(created_at.desc, id.asc)
```

generated/query.sql

```
ORDER BY created_at DESC, id ASC
```

CTEs and analytics

over

A scalar function can use `.over { ... }` with checked partitions and ordering.

BtrQL SQL
query.btrql

```
users
  .select(
    id,
    row_number().over {
      partitionBy(region)
      orderBy(created_at.desc)
    } -> rn
  )
```

generated/query.sql

```
SELECT
  id,
  ROW_NUMBER() OVER (
    PARTITION BY region
    ORDER BY created_at DESC
  ) AS rn
FROM users;
```

with

`with { name = query }` names statement-local row sets.

BtrQL SQL
query.btrql

```
with {
  active_users =
    users
      .where(active == TRUE)
}
active_users
  .select(id)
```

generated/query.sql

```
WITH active_users AS (
  SELECT * FROM users
  WHERE active = TRUE
)

SELECT id FROM active_users;
```

`qualify`

`qualify(...)` filters rows after window expressions on dialects that support it.

BtrQL SQL

query.btrql

```
users
  .select(
    id,
    row_number().over {
      orderBy(id.asc)
    } -> rn
  )
  .qualify(rn == 1)
```

generated/query.sql

```
SELECT
  id,
  ROW_NUMBER() OVER (ORDER BY id ASC) AS rn
FROM users QUALIFY rn = 1;
```

qualify(...)

The supported qualify(...) surface is now called out directly for the dialects that lower it natively and preserve it through reverse, runtime, and tooling.

Mutations and RETURNING

Update aliases

Update assignments reuse `expr -> column` so the target column is explicit.

BtrQL SQL
query.btrql

```
users
  .where(id == 1)
  .update {
    into users
    set(FALSE -> active)
  }
```

generated/query.sql

```
UPDATE users SET active = FALSE
WHERE id = 1;
```

Returning aliases

`returningWith(old, new, ...)` names old/new row images for supported RETURNING forms.

BtrQL SQL
query.btrql

```
users
  .where(id == 1)
  .update {
    into users
    set(TRUE -> active)
  }
  .returningWith(
    old_row,
    new_row,
    old_row.active -> was_active,
    new_row.active -> is_active
  )
```

generated/query.sql

```
UPDATE users SET active = TRUE
WHERE id = 1 RETURNING old.active AS was_active, new.active AS is_active;
```

insert

`insert(...)` writes the current row shape into a target table.

BtrQL | SQL

query.btrql

```
new_users
  .select(id, name)
  .insert(users)
```

generated/query.sql

```
INSERT INTO users (id, name) SELECT
  id,
  name
FROM new_users;
```

insertByName

`insertByName(...)` inserts by matching column names where supported.

BtrQL | SQL

query.btrql

```
new_users
  .insertByName(users)
```

generated/query.sql

```
INSERT INTO users BY NAME SELECT * FROM new_users;
```

update

`update { into ... set(...) }` updates a target from the current filtered pipeline.

BtrQL | SQL

query.btrql

```
users
  .where(id == 1)
  .update {
    into users
    set(TRUE -> active)
  }
```

generated/query.sql

```
UPDATE users SET active = TRUE
WHERE id = 1;
```

`delete`

`delete` or `deleteFrom(target)` deletes rows selected by the pipeline.

BtrQL | SQL
query.btrql

```
users
  .where(active == FALSE)
  .deleteFrom(users)
```

generated/query.sql

```
DELETE FROM users
WHERE active = FALSE;
```

`mergeInto`

`mergeInto` expresses a checked upsert/merge shape.

BtrQL | SQL
query.btrql

```
incoming
  .mergeInto(users) {
    on(users.id == incoming.id)
    whenMatchedUpdate(incoming.name -> name)
    whenNotMatchedInsert(id, name)
  }
```

generated/query.sql

```
MERGE INTO users USING incoming ON users.id = incoming.id WHEN MATCHED THEN
  UPDATE SET name = incoming.name
WHEN NOT MATCHED THEN INSERT (id, name) VALUES (incoming.id, incoming.name);
```

`returning`

`returning(...)` projects columns from affected DML rows where supported.

BtrQL SQL

query.btrql

```
users
  .where(id == 1)
  .update {
    into users
    set(TRUE -> active)
  }
  .returning(id, active)
```

generated/query.sql

```
UPDATE users SET active = TRUE
WHERE id = 1 RETURNING id, active;
```

`returningWith`

`returningWith(old, new, ...)` names row images before projecting returned expressions.

BtrQL SQL

query.btrql

```
users
  .where(id == 1)
  .update {
    into users
    set(TRUE -> active)
  }
  .returningWith(
    old_row,
    new_row,
    old_row.active -> was_active,
    new_row.active -> is_active
  )
```

generated/query.sql

```
UPDATE users SET active = TRUE
WHERE id = 1 RETURNING old.active AS was_active, new.active AS is_active;
```

returning(...) across supported DML subsets

The public returning(...) surface is documented as the shared DML-output path for the supported DML subsets.

Types and constraints

`withoutOverlaps`

A table type constraint can model PostgreSQL temporal-key overlap checks.

BtrQL SQL

query.btrql

```
table bookings [room_id: BIGINT, valid_at: TSRANGE]
constraints { withoutOverlaps(room_id, valid_at) }
```

generated/query.sql

```
CREATE TABLE bookings (..., PRIMARY KEY (room_id, valid_at WITHOUT OVERLAPS));
```

Grouping and HAVING

by

`group { by(...) aggregate(...) }` groups rows and projects aggregate outputs.

BtrQL SQL
query.btrql

```
orders
  .group {
    by(status)
    aggregate(count(order_id) -> order_count)
  }
  .select(status, order_count)
```

generated/query.sql

```
SELECT
  status,
  COUNT(order_id) AS order_count
FROM orders
GROUP BY status;
```

Aggregate aliases

Aggregate expressions require `-> alias` so grouped outputs have stable names.

BtrQL SQL
query.btrql

```
orders
  .group {
    by(status)
    aggregate(sum(amount) -> total_amount)
  }
```

generated/query.sql

```
SELECT
  status,
  SUM(amount) AS total_amount
FROM orders
GROUP BY status;
```

Aggregate filters

`.filter(predicate)` applies a predicate to one aggregate.

BtrQL | SQL

query.btrql

```
orders
  .group {
    by(customer_id)
    aggregate(sum(amount).filter(status == 'paid') -> paid_amount)
  }
```

generated/query.sql

```
SELECT
  customer_id,
  SUM(amount) FILTER (WHERE status = 'paid') AS paid_amount
FROM orders
GROUP BY customer_id
;
```

`having`

`having(...)` filters grouped rows.

BtrQL | SQL

query.btrql

```
orders
  .group {
    by(status)
    aggregate(count(*) -> row_count)
    having(count(*) > 10)
  }
```

generated/query.sql

```
SELECT
  status,
  COUNT(*) AS row_count
FROM orders
GROUP BY status
HAVING COUNT(*) > 10;
```

Top-level statements

`schema`

`schema` declares the physical schema used for objects emitted from the file.

BtrQL | SQL

query.btrql

```
schema analytics
using live_catalog

view active_users =
  users
    .where(active == TRUE)
    .select(id, name)
```

generated/query.sql

```
CREATE VIEW analytics.active_users AS SELECT
  id,
  name
FROM live_catalog.users
WHERE active = TRUE;
```

`using`

`using` selects the namespace used to resolve unqualified source and callable names.

BtrQL | SQL

query.btrql

```
using analytics

users
  .select(id, name)
```

generated/query.sql

```
SELECT
  id,
  name
FROM analytics.users;
```

`call`

`call` invokes a visible procedure or callable statement from the top level after arguments are checked.

BtrQL | SQL

query.btrql

```
using analytics

call refresh_user_totals(2024)
```

generated/query.sql

```
CALL analytics.refresh_user_totals(2024);
```

<code>import</code>

File-based module names, `.btrql` and `.sql` imports, and selected import aliases.

See Module imports and aliases for the full syntax, examples, and boundaries.

Joins and APPLY

Join steps

Join steps combine two checked row shapes with explicit predicates.

BtrQL SQL
query.btrql

```
users.as(u)
  .innerJoin(orders.as(o) on u.id == o.user_id)
  .select(u.id, o.order_id)
```

generated/query.sql

```
SELECT
  u.id,
  o.order_id
FROM users AS u INNER JOIN orders AS o ON u.id = o.user_id;
```

`asofJoin`

`asofJoin` and `leftAsofJoin` model nearest-preceding joins where supported.

BtrQL SQL
query.btrql

```
trades
  .leftAsofJoin(
    quotes
    on(trades.symbol == quotes.symbol and trades.ts >= quotes.ts)
  )
  .select(trade_id, price)
```

generated/query.sql

```
SELECT
  trade_id,
  price
FROM trades
ASOF LEFT JOIN quotes ON trades.symbol = quotes.symbol AND trades.ts >= quotes.ts;
```

`crossApply` / `outerApply`

`crossApply` and `outerApply` run a right-side relation per left row.

BtrQL SQL
query.btrql

```
users
  .crossApply(
    orders
      .where(user_id == users.id)
  )
  .select(id, order_id)
```

generated/query.sql

```
SELECT
    id,
    order_id
FROM users CROSS APPLY (SELECT * FROM orders
WHERE user_id = users.id) AS orders;
```

crossApply(...) and outerApply(...)

crossApply and outerApply run a right-side relation for each left-side row. Use them only when the target dialect supports APPLY or LATERAL.

JSON and table functions

Table functions

A callable table source becomes the SQL table-function form for supporting dialects.

BtrQL SQL
query.btrql

```
generate_series(1, 3)
  .select(value)
```

generated/query.sql

```
SELECT value FROM generate_series(1, 3);
```

JSON accessors

JSON postfix helpers read a JSON segment and coerce the result where supported.

BtrQL SQL
query.btrql

```
events
  .select(event_id, payload.json('user').asText -> user_id)
```

generated/query.sql

```
SELECT
  event_id,
  payload ->> 'user' AS user_id
FROM events;
```

`jsonTable`

A descriptor lists projected JSON table fields with `field -> path` entries.

BtrQL SQL
query.btrql

```
json_table(payload, [user_id -> $.user.id.asText, total -> $.total.asText])
  .select(user_id, total)
```

generated/query.sql

```
SELECT
    user_id,
    total
FROM JSON_TABLE(payload, '$' COLUMNS (...));
```

Syntax overview

BtrQL source is a file of headers, symbols definitions and top level queries and procedure calls.

Syntax failures and migration guidance are reported through Diagnostics and errors.

Grammar path

- file A source file contains headers followed by items.
  - headers Top-level context before declarations.
    - schema
    - using
    - import
  - items Definitions, objects, executable statements, and routines.
    - declarations
      - val
      - type
      - extension
      - macro
    - schema objects
      - table
      - view
      - materialized view
      - index
    - queries and DML
      - relation pipeline
      - insert / update / delete / merge
    - routines
      - function
      - table function
      - procedure
- relation A relation starts from a source and changes through pipeline steps.
  - source
    - table or view name
    - subquery
    - table function
    - literal rows
  - steps
    - select / addColumn / removeColumn / distinct
    - where / group / having
    - join / apply
    - with / over / qualify
    - union / intersect / except
    - orderBy / limit / offset
- expression Expressions cover values, names, calls, operators, branches, and dialect helpers.
  - identifiers
  - literals

Grammar

This grammar map is compact, not the full formal grammar. The extension-column surface below is currently implemented by the PostgreSQL and ClickHouse C# prototypes.

```
file              ::= header* item*
header            ::= using_header | schema_header | import_header
using_header      ::= "using" schema_name
schema_header     ::= "schema" schema_name
import_header     ::= "import" module_path [ "{" import_alias ("," import_alias)* "}" ]
import_alias      ::= ident [ "->" ident ]

item              ::= query | dml | call_stmt | object_decl | reusable_decl |
routine_decl | macro_decl
reusable_decl     ::= "val" ident "=" expr
                    | "type" ident "=" type_expr
                    | extension_decl
extension_decl    ::= "extension" table_type "{" extension_member* "}"
extension_member  ::= method_decl | column_decl
method_decl       ::= "method" ident [ table_param_group ] [ scalar_param_group ] [
"=>" table_type ] "=" relation
column_decl       ::= "column" ident [ ":" scalar_type ] "=" expr
table_param_group ::= "(" ident ":" table_type ("," ident ":" table_type)* ")"
scalar_param_group ::= "(" ident ":" scalar_type ("," ident ":" scalar_type)* ")"

object_decl       ::= "table" ident table_shape
                    | "view" ident "=" relation
                    | "materialized view" ident "=" relation
                    | [ "unique" ] "index" ident "on" ident "(" expr_list ")" [ "where"
expr ]

routine_decl      ::= "function" ident "(" param_list? ")" "=" expr
                    | "table function" ident "(" param_list? ")" "=>" table_shape "="
relation
                    | "procedure" ident "(" param_list? ")" "{" proc_stmt* "}"

query             ::= relation
relation          ::= relation_source relation_step*
relation_source   ::= ident | qualified_ident | "(" relation ")" | row_literal |
table_literal
relation_step     ::= ".as(" ident ")"
                    | ".select(" projection_list ")"
                    | ".where(" expr ")"
                    | ".addColumn(" projection_list ")"
                    | ".removeColumn(" column_list ")"
                    | ".orderBy(" sort_list ")"
                    | ".limit(" expr ")"
                    | ".offset(" expr ")"
                    | ".distinct"
                    | ".distinctOn(" expr_list ")"
                    | ".group { by(" expr_list ") aggregate(" projection_list ")" [ "
having(" expr ")" ] " }"
                    | join_step | apply_step | set_step | qualify_step
projection_list   ::= projection ("," projection)*
projection        ::= expr [ "->" ident ]
join_step         ::= "." join_name "(" relation " on " expr ")"
apply_step        ::= ".crossApply(" relation ")" | ".outerApply(" relation ")"
```

```
set_step          ::= "." set_name "(" relation ")"
qualify_step      ::= ".qualify(" expr ")"

dml               ::= relation ".insert(" ident ")"
                    | relation ".update { into " ident " set(" assignment_list ") }"
                    | relation ".delete"
                    | relation ".mergeInto(" ident ") { ... }"
                    | dml ".returning(" projection_list ")"

expr              ::= literal | ident | qualified_ident | function_call
                    | unary_expr | binary_expr | case_expr | window_expr | "(" expr ")"
```

`column recipe` and `addColumn[Type]` are removed from the C# prototype grammar and receive targeted migration diagnostics. A bare extension-column request is an ordinary scalar expression whose output alias is the extension member name. Earlier aliases in the same `select` or `addColumn` are visible only to later projections.

Top-level statements

Statements that set context, import modules, or call procedures.

Top-level statements

Loading Top-level statements...

Reusable abstractions

Names, relation types, extension methods, extension columns, and macros for reuse.

Reusable abstractions

Loading Reusable abstractions...

Schema objects

Tables, views, materialized views, and indexes.

Schema objects

Loading Schema objects...

Query pipeline

Relation-first queries, filters, ordering, limits, and expressions.

Query pipeline

Loading Query pipeline...

Projection shaping

Aliases, distinct rows, and column-shape changes.

Projection shaping

Loading Projection shaping...

Set operations

Union, intersect, and except composition.

Set operations

Loading Set operations...

Grouping and HAVING

Grouping columns, aggregate expressions, and HAVING.

Grouping and HAVING

Loading Grouping and HAVING...

Joins and APPLY

Joins and dependent relation steps.

Joins and APPLY

Loading Joins and APPLY...

CTEs and analytics

CTEs, window expressions, and post-window filters.

CTEs and analytics

Loading CTEs and analytics...

JSON and table functions

JSON helpers, table-function sources, and search/planner helpers.

JSON and table functions

Loading JSON and table functions...

Mutations and RETURNING

Insert, update, delete, merge, and returned row images.

Mutations and RETURNING

Loading Mutations and RETURNING...

Types and constraints

Declared types, table shapes, and constraints.

Types and constraints

Loading Types and constraints...

Routine definitions

Functions, table functions, procedures, and procedure bodies.

Routine definitions

Loading Routine definitions...

Builtin functions

Recognized function families for dialect-aware SQL generation.

Builtin functions

Loading Builtin functions...

Walkthroughs

Applied BtrQL-to-SQL examples that combine multiple syntax elements.

Walkthroughs

Loading Walkthroughs...

Errors

Parse, name, type, and dialect errors before SQL is emitted.

Errors

Loading Errors...

PostgreSQL Dialect

PostgreSQL is the broadest current target for BtrQL examples, generated SQL checks, and real database execution.

PostgreSQL Dialect

Loading PostgreSQL Dialect...

ClickHouse syntax

ClickHouse-specific BtrQL forms.

ClickHouse syntax

Loading ClickHouse syntax...

Projection shaping

->

`expr -> alias` names a computed projection or assignment output.

BtrQL SQL
query.btrql

```
users
  .select(id, upper(name) -> display_name)
  .orderBy(display_name.asc)
```

generated/query.sql

```
SELECT
  id,
  UPPER(name) AS display_name
FROM users
ORDER BY display_name ASC;
```

Pivot aliases

Pivot values can use `value -> alias` to control generated output names.

BtrQL SQL
query.btrql

```
sales
  .pivot {
    by(region)
    on(status)
    using(sum(amount) -> total)
    values('open' -> open_total, 'closed' -> closed_total)
  }
```

generated/query.sql

```
SELECT
  region,
  SUM(CASE WHEN status = 'open' THEN amount END) AS open_total,
  SUM(CASE WHEN status = 'closed' THEN amount END) AS closed_total
FROM sales
GROUP BY region
;
```

distinct

`.distinct` deduplicates the current projected row shape.

BtrQL | SQL

query.btrql

```
users
  .select(region)
  .distinct
```

generated/query.sql

```
SELECT DISTINCT region FROM users;
```

distinctOn

`distinctOn(...)` requests first-row-per-key semantics where the dialect supports it.

BtrQL | SQL

query.btrql

```
users
  .select(region, name)
  .distinctOn(region)
  .orderBy(region.asc, name.asc)
```

generated/query.sql

```
SELECT DISTINCT ON (region)
  region,
  name
FROM users
ORDER BY region ASC, name ASC;
```

`removeColumn`

`removeColumn(...)` drops columns from the current row shape.

BtrQL SQL
query.btrql

```
users
  .removeColumn(password_hash)
  .select(id, name)
```

generated/query.sql

```
SELECT
  id,
  name
FROM users;
```

`addColumn`

`addColumn(...)` appends computed columns to the row shape.

BtrQL SQL
query.btrql

```
users
  .addColumn(case when active then 'active' else 'inactive' end -> active_label)
  .select(id, active_label)
```

generated/query.sql

```
SELECT
  id,
  CASE WHEN active THEN 'active' ELSE 'inactive' END AS active_label
FROM users;
```

`rowShape`

`rowShape(...)` declares the expected row columns after reshaping.

BtrQL SQL
query.btrql

```
users
  .select(id, name)
  .rowShape([id: BIGINT, name: TEXT])
```

generated/query.sql

```
SELECT
  id,
  name
FROM users;
```

PostgreSQL DISTINCT ON lowering

PostgreSQL now exposes distinctOn(...) as a public deduplication surface with exact parse/print, reverse reconstruction, generated parity, and editor/runtime coverage for the supported subset.

Reusable abstractions

Typed column lists

A table-shaped type can stand in for a repeated column list.

BtrQL | SQL

query.btrql

```
type PublicUser = [id: BIGINT, name: TEXT]

users
  .select[PublicUser]

archived_users
  .select[PublicUser]
```

generated/query.sql

```
SELECT
  id,
  name
FROM users;
SELECT
  id,
  name
FROM archived_users;
```

Symbolic relation results in Outline

Extension-method details bind open relation variables explicitly. `T0` is always the receiver, while `T1`, `T2`, and later variables follow table-kind argument order. `Tn` is never a scalar type. Documentation uses `ScalarType0`, `ScalarType1`, and so on only when a scalar is intentionally schematic; when inference knows the type, Outline prints the concrete name such as `INT`, `DECIMAL`, or `TEXT`. Both table-kind and scalar-kind argument groups are always shown, even when one group is empty.

```
T0=[id: INT, amount: DECIMAL].()() => T0
T0=[id: INT].(other: T1=[id: INT])() => T0 ++ T1
T0=[id: INT].(left: T1=[id: INT], right: T2=[id: INT])() => T0 ++ T1 ++ T2
```

Set operations keep their left symbolic operand: `self.union(other)` returns `T0`, while `other.union(self)` returns `T1`. `++ T1` means joined row shapes, never SQL `UNION`. A join whose sides are reshaped independently can render as `T0 -- [bla: ScalarType1] ++ T1 -- [foo: ScalarType2] ++ [bar: ScalarType3]`.

Projection and grouping changes remain attached to the originating variable. Omitted fields use `--`, computed fields and aggregate aliases use `++`, and a rename is always paired: `T0 -- [old_name: TEXT] ++ [new_name: TEXT]`. A type replacement is likewise `T0 -- [value: INT] ++ [value: DECIMAL]`. Simple grouping keys keep their original identity; non-grouped fields are removed and aggregate outputs are added. Extra call-site columns remain part of the instantiated `Tn` until an operation explicitly omits them.

Extension-column symbols use the restricted form `[receiver] ++ [columnName: ConcreteScalarType]`; their scalar dependencies are substituted into expressions but are not listed as output columns. Optional written method and column annotations are checked against normalized inference and mismatches are errors. This symbolic Outline surface is currently implemented in the PostgreSQL and ClickHouse C# prototypes.

Extension columns

`column` members are receiver-bound scalar expressions. Any relation satisfying the receiver contract can use them in projections, filters, joins, grouping, ordering, windows, and DML expressions. Requesting a bare extension column from `select` or `addColumn` gives it the member name. Dependencies are substituted into the requested expression but are not added as output columns automatically. The PostgreSQL and ClickHouse C# compilers implement this surface.

Relation methods and scalar columns share one receiver contract; one extension column can depend on another without projecting the dependency.

query.btrql

```
using analytics

type EngagementInput = [
  posts_viewed: INT,
  comments_written: INT
]

extension EngagementInput {
  method activeOnly =
    self
      .where(is_engaged)

  method withMinimumPosts()(minPosts: INT) =
    self
      .where(posts_viewed >= minPosts)

  column engagement_score: INT =
    posts_viewed + comments_written * 5

  column is_engaged: BOOLEAN =
    posts_viewed > 10 and comments_written > 0

  column engagement_label =
    case when is_engaged then 'active' else 'inactive' end
}

view engaged_users =
  users
    .activeOnly()
    .withMinimumPosts()(5)
    .addColumn(engagement_score, engagement_label)
```

generated/postgresql.sql

```
CREATE VIEW "engaged_users" AS
SELECT
  *,
  "_q3"."posts_viewed" + "_q3"."comments_written" * 5 AS "engagement_score",
  CASE
    WHEN "_q3"."posts_viewed" > 10 AND "_q3"."comments_written" > 0
    THEN 'active'
    ELSE 'inactive'
  END AS "engagement_label"
FROM (
  SELECT *
  FROM (
    SELECT
      "users"."user_id",
      "users"."posts_viewed",
      "users"."comments_written"
    FROM "analytics"."users" AS "users"
  ) AS "_q1"
  WHERE "_q1"."posts_viewed" > 10 AND "_q1"."comments_written" > 0
) AS "_q3"
WHERE "posts_viewed" >= 5;
```

generated/clickhouse.sql

```
CREATE VIEW `engaged_users` AS
SELECT
  *,
  `_q3`.`posts_viewed` + `_q3`.`comments_written` * 5 AS `engagement_score`,
  CASE
    WHEN `_q3`.`posts_viewed` > 10 AND `_q3`.`comments_written` > 0
    THEN 'active'
    ELSE 'inactive'
  END AS `engagement_label`
FROM (
  SELECT *
  FROM (
    SELECT
      `users`.`user_id`,
      `users`.`posts_viewed`,
      `users`.`comments_written`
    FROM `analytics`.`users` AS `users`
  ) AS `_q1`
  WHERE `_q1`.`posts_viewed` > 10 AND `_q1`.`comments_written` > 0
) AS `_q3`
WHERE `posts_viewed` >= 5;
```

Macros

A macro expands a checked source template before normal compilation.

BtrQL SQL
query.btrql

```
macro activeView(src: UserShape)(out: identifier) =
  <{ view $out = $src.where(active == TRUE) }>

activeView(users)(active_users)
```

generated/query.sql

```
CREATE VIEW active_users AS
SELECT *
FROM users
WHERE active = TRUE;
```

`val`

`val` names a scalar expression for reuse in later BtrQL source.

BtrQL SQL
query.btrql

```
val activeOnly = active == TRUE

users
  .where(activeOnly)
  .select(id)

archived_users
  .where(activeOnly)
  .select(id)
```

generated/query.sql

```
SELECT id
FROM users
WHERE active = TRUE;

SELECT id
FROM archived_users
WHERE active = TRUE;
```

`type`

A relation type names a reusable row contract. The same row shape can drive extension-method receivers and repeated projections.

BtrQL | SQL

query.btrql

```
type CustomerCard = [id: BIGINT, name: TEXT, region: VARCHAR]

extension CustomerCard {
  method inRegion()(targetRegion: VARCHAR) =
    self
      .where(region == targetRegion)
}

users
  .select[CustomerCard]

archived_users
  .where(active == TRUE)
  .select[CustomerCard]

customers
  .inRegion()('EMEA')
  .select[CustomerCard]
```

generated/query.sql

```
SELECT
  id,
  name,
  region
FROM users;

SELECT
  id,
  name,
  region
FROM archived_users
WHERE active = TRUE;

SELECT
  id,
  name,
  region
FROM customers
WHERE region = 'EMEA';
```

`extension`

An extension adds checked relation methods to a row shape. Extension methods and extension columns use a receiver contract: a compatible relation can reuse the member, and a method expands inline before SQL is emitted.

BtrQL SQL
query.btrql

```
type LeaderboardRow = [
  customer_id: BIGINT,
  customer_name: TEXT,
  region: VARCHAR,
  sales_rep_name: TEXT,
  order_id: BIGINT,
  amount: DECIMAL,
  regional_paid_amount: DECIMAL,
  regional_rank: INT
]

extension OrderShape {
  method regionalLeaderboard()(minAmount: DECIMAL) =
    self
      .as(o)
      .innerJoin(customers.as(c) on o.customer_id == c.id)
      .leftJoin(sales_reps.as(rep) on c.sales_rep_id == rep.id)
      .where(o.status == 'paid' and o.amount >= minAmount)
      .select[LeaderboardRow](
        c.id -> customer_id,
        c.name -> customer_name,
        c.region,
        rep.name -> sales_rep_name,
        o.order_id,
        o.amount,
        sum(o.amount)
          .over {
          partitionBy(c.region)
        } -> regional_paid_amount,
        row_number()
          .over {
          partitionBy(c.region)
          orderBy(o.amount.desc, o.ordered_at.desc)
        } -> regional_rank
      )
}

val enterpriseOrders =
  orders
    .where(order_kind == 'enterprise')
    .regionalLeaderboard()(500)
    .where(regional_rank <= 3)

val renewalOrders =
  orders
    .where(order_kind == 'renewal')
    .regionalLeaderboard()(250)
    .where(regional_paid_amount >= 2500)

val expansionOrders =
  orders
    .where(order_kind == 'expansion')
```

```
    .regionalLeaderboard()(100)
    .where(regional_rank <= 5)

enterpriseOrders
  .unionAll(renewalOrders)
  .unionAll(expansionOrders)
  .orderBy(region.asc, regional_rank.asc)
```

generated/query.sql

```
SELECT
  leaderboard.customer_id,
  leaderboard.customer_name,
  leaderboard.region,
  leaderboard.sales_rep_name,
  leaderboard.order_id,
  leaderboard.amount,
  leaderboard.regional_paid_amount,
  leaderboard.regional_rank
FROM (
  SELECT *
  FROM (
    SELECT
      c.id AS customer_id,
      c.name AS customer_name,
      c.region,
      rep.name AS sales_rep_name,
      o.order_id,
      o.amount,
      SUM(o.amount) OVER (
        PARTITION BY c.region
      ) AS regional_paid_amount,
      ROW_NUMBER() OVER (
        PARTITION BY c.region
        ORDER BY o.amount DESC, o.ordered_at DESC
      ) AS regional_rank
    FROM orders AS o
    INNER JOIN customers AS c
      ON o.customer_id = c.id
    LEFT JOIN sales_reps AS rep
      ON c.sales_rep_id = rep.id
    WHERE
      o.order_kind = 'enterprise'
      AND o.status = 'paid'
      AND o.amount >= 500
  ) AS enterprise_ranked
  WHERE enterprise_ranked.regional_rank <= 3

  UNION ALL

  SELECT *
  FROM (
    SELECT
      c.id AS customer_id,
      c.name AS customer_name,
      c.region,
      rep.name AS sales_rep_name,
      o.order_id,
      o.amount,
      SUM(o.amount) OVER (
        PARTITION BY c.region
      ) AS regional_paid_amount,
```

```
      ROW_NUMBER() OVER (
        PARTITION BY c.region
        ORDER BY o.amount DESC, o.ordered_at DESC
      ) AS regional_rank
    FROM orders AS o
    INNER JOIN customers AS c
      ON o.customer_id = c.id
    LEFT JOIN sales_reps AS rep
      ON c.sales_rep_id = rep.id
    WHERE
      o.order_kind = 'renewal'
      AND o.status = 'paid'
      AND o.amount >= 250
  ) AS renewal_ranked
  WHERE renewal_ranked.regional_paid_amount >= 2500

  UNION ALL

  SELECT *
  FROM (
    SELECT
      c.id AS customer_id,
      c.name AS customer_name,
      c.region,
      rep.name AS sales_rep_name,
      o.order_id,
      o.amount,
      SUM(o.amount) OVER (
        PARTITION BY c.region
      ) AS regional_paid_amount,
      ROW_NUMBER() OVER (
        PARTITION BY c.region
        ORDER BY o.amount DESC, o.ordered_at DESC
      ) AS regional_rank
    FROM orders AS o
    INNER JOIN customers AS c
      ON o.customer_id = c.id
    LEFT JOIN sales_reps AS rep
      ON c.sales_rep_id = rep.id
    WHERE
      o.order_kind = 'expansion'
      AND o.status = 'paid'
      AND o.amount >= 100
  ) AS expansion_ranked
  WHERE expansion_ranked.regional_rank <= 5
) AS leaderboard
ORDER BY leaderboard.region ASC, leaderboard.regional_rank ASC;
```

Routine definitions

`function`

`function` declares a scalar callable body.

BtrQL | SQL

query.btrql

```
function activeLabel(flag: BOOLEAN) = case when flag then 'active' else 'inactive' end
```

generated/query.sql

```
CREATE FUNCTION activeLabel(flag BOOLEAN) RETURNS TEXT AS ...;
```

`table function`

`table function` declares a callable that returns a table-shaped row set.

BtrQL | SQL

query.btrql

```
table function activeUsers(flag: BOOLEAN) => [id: BIGINT, name: TEXT] =
  users
    .where(active == flag)
    .select(id, name)
```

generated/query.sql

```
CREATE FUNCTION activeUsers(flag BOOLEAN) RETURNS TABLE (id BIGINT, name TEXT) AS ...;
```

`procedure`

`procedure` declares a callable statement block.

BtrQL | SQL

query.btrql

```
procedure refreshUsers {
  call refresh_user_totals();
}
```

generated/query.sql

```
CREATE PROCEDURE refreshUsers() AS BEGIN CALL refresh_user_totals(); END;
```

Parameters

Routine parameters use `name: TYPE` and are checked inside the body.

BtrQL | SQL
query.btrql

```
function scoreLabel(score: INTEGER) = case when score > 10 then 'high' else 'low' end
```

generated/query.sql

```
CREATE FUNCTION scoreLabel(score INTEGER) RETURNS TEXT AS ...;
```

`var`

`var name: TYPE = expr` declares a local procedure value.

BtrQL | SQL
query.btrql

```
var total: DECIMAL = 0;



procedure p() { var total: DECIMAL = 0; return; }
```

generated/query.sql

```
DECLARE total DECIMAL = 0;
```

Assignment

`name = expr` updates a procedure-local value.

BtrQL | SQL
query.btrql

```
total = total + amount;



procedure p() { var total: DECIMAL = 0; total = total + amount; return; }
```

generated/query.sql

```
SET total = total + amount;
```

`if`

`if (condition) { ... } else { ... }` branches a procedure block.

BtrQL | SQL
query.btrql

```
procedure p(amount: DECIMAL) {
  if (amount > 0) {
    call mark_positive();
  }
  else {
    call mark_empty();
  }
}
```

generated/query.sql

```
IF amount > 0 THEN CALL mark_positive(); ELSE CALL mark_empty(); END IF;
```

`case`

A procedure `case` block branches over conditions or a scrutinee.

BtrQL | SQL
query.btrql

```
procedure p(amount: DECIMAL) {
  case {
    when amount > 0 {
      call positive();
    }
    else {
      call empty();
    }
  }
}
```

generated/query.sql

```
CASE WHEN amount > 0 THEN CALL positive(); ELSE CALL empty(); END CASE;
```

`while`

`while (condition) { ... }` repeats a procedure block while a condition holds.

BtrQL SQL
query.btrql

```
procedure countdown(remaining: INTEGER) {
  while (remaining > 0) {
    remaining = remaining - 1;
  }
}
```

generated/query.sql

```
WHILE remaining > 0 LOOP remaining := remaining - 1; END LOOP;
```

`for`

`for (row in query) { ... }` iterates over a checked query fragment.

BtrQL SQL
query.btrql

```
procedure touchActive {
  for (u in
    users
      .where(active == TRUE)
  ) {
    call touch_user(u.id);
  }
}
```

generated/query.sql

```
FOR u IN
SELECT *
FROM users
WHERE active = TRUE LOOP CALL touch_user(u.id); END LOOP;
```

`loop`

`loop { ... }` creates an unconditional procedure loop.

BtrQL | SQL

query.btrql

```
procedure p {
  loop {
    call tick();
    return;
  }
}
```

generated/query.sql

```
LOOP CALL tick(); RETURN; END LOOP;
```

`return`

`return` exits a procedure, optionally with a value in supported contexts.

BtrQL | SQL

query.btrql

```
return total;



procedure p() { var total: INTEGER = 1; return total; }
```

generated/query.sql

```
RETURN total;
```

`call`

`call name(args...)` invokes a procedure from top-level or inside a routine.

BtrQL | SQL
query.btrql

```
call refresh_user_totals(2024)

procedure p {
  call refresh_user_totals(2024);
}
```

generated/query.sql

```
CALL refresh_user_totals(2024);
```

Embedded statement

A procedure body can contain a BtrQL query, DML, view, or table declaration fragment.

BtrQL | SQL
query.btrql

```
procedure purgeInactive {
  users
    .where(active == FALSE)
    .deleteFrom(users);
}
```

generated/query.sql

```
DELETE FROM users
WHERE active = FALSE;
```

Schema objects

`table`

`table` declares a checked table shape for generated DDL and later queries.

BtrQL SQL
query.btrql

```
table users [id: BIGINT, name: TEXT, active: BOOLEAN]
```

generated/query.sql

```
CREATE TABLE users (id BIGINT, name TEXT, active BOOLEAN);
```

`view`

`view` names a reusable checked query object.

BtrQL SQL
query.btrql

```
view active_users =
  users
    .where(active == TRUE)
    .select(id, name)
```

generated/query.sql

```
CREATE VIEW active_users AS SELECT
  id,
  name
FROM users
WHERE active = TRUE;
```

`materialized view`

`materialized view` emits a persisted query object on supported dialects.

BtrQL SQL
query.btrql

```
materialized view active_snapshot =
  users
    .where(active == TRUE)
    .select(id, name)
```

generated/query.sql

```
CREATE MATERIALIZED VIEW active_snapshot AS SELECT
  id,
  name
FROM users
WHERE active = TRUE;
```

index

`index` and `unique index` declare checked key expressions.

BtrQL | SQL

query.btrql

```
unique index users_email_idx on users (email) where active == TRUE
```

generated/query.sql

```
CREATE UNIQUE INDEX users_email_idx ON users (email) WHERE active = TRUE;
```

Set operations

`union`

`union(...)` and `unionAll(...)` combine compatible query branches.

BtrQL SQL
query.btrql

```
active_users
  .union(recent_users)
  .orderBy(id.asc)
```

generated/query.sql

```
SELECT * FROM active_users
UNION
SELECT * FROM recent_users
ORDER BY id ASC;
```

`unionByName`

`unionByName` aligns columns by name where supported.

BtrQL SQL
query.btrql

```
north_sales
  .unionByName(south_sales)
  .select(order_id, amount)
```

generated/query.sql

```
SELECT
  order_id,
  amount
FROM north_sales
UNION BY NAME
SELECT
  order_id,
  amount
FROM south_sales;
```

`intersect` / `except`

`intersect(...)` keeps shared rows and `except(...)` subtracts rows.

BtrQL SQL

query.btrql

```
active_users
  .intersect(recent_users)
  .select(id)
```

generated/query.sql

```
SELECT id FROM active_users
INTERSECT
SELECT id FROM recent_users;
```