Catrust

Categorical Query Language Engine en Rust

Guide utilisateur complet

Projet Catrust

Avril 2026 version 0.1

ń Une base de données est un foncteur.
Une migration est une transformation naturelle.
Un schéma est une catégorie. ż

Résumé

Catrust est un moteur de requêtes et de migrations de schémas implémenté en Rust, fondé sur la *théorie des catégories*.

Le projet implémente **CQL** (*Categorical Query Language*), un langage formellement défini par David Spivak et Ryan Wisnesky, qui modélise les schémas de bases de données comme des catégories, les données comme des foncteurs, et les migrations comme des extensions de Kan.

Catrust fournit :

— Un **moteur in-memory** complet : schémas, instances, mappings, migrations Σ et Δ, requêtes et agrégations.

— Un **parser CQL** textuel (fichiers `.cql`).

— Quatre **backends** de génération de code : PostgreSQL, Snowflake, Trino et Neo4j.

— Une **CLI** complète : `validate`, `parse`, `generate`, `export`, `migrate`, `diff`, `apply`, `demo`, `viz`, `serve`, `init`, `introspect`, `status`.

— **Catrust Studio** : interface web interactive (éditeur CQL, prévisualisation SVG, navigateur de fichiers, export).

— **Introspection** : génération automatique de `.cql` depuis un schéma PostgreSQL existant.

— **Historique de migrations** : table `_catrust_migrations` avec empreinte FNV-1a de chaque déploiement.

— **Scaffold de projet** (`catrust init`) : génère `catrust.toml` et un schéma d'exemple.

— Une **sérialisation JSON** versionnée via `serde`.

Table des matières

Première partie

Démarrage

Chapitre 1

Installation et premiers pas

1.1 Prérequis

— Rust stable 1.70 ou supérieur.

— Cargo (inclus avec Rust).

Vérification :

```
rustc --version   # >= 1.70.0
cargo --version
```

1.2 Installation

```
git clone https://github.com/sirgudu/Catrust
cd Catrust
cargo build
```

Le workspace contient quatre crates :

— **catrust** binaire CLI + crate racine.

— **catrust-core** moteur catégorique pur (zéro dépendance DB).

— **catrust-backend** génération SQL/Cypher.

— **catrust-parser** parser CQL textuel.

1.3 Lancer les tests

```
cargo test --workspace
# 126 tests -- 0 echecs
```

1.4 La CLI en un coup d'il

```
catrust --help
```

Commande	Description
`parse`	Parse un fichier `.cql` et affiche la structure.
`validate`	Valide un schéma (cohérence structurelle).
`generate sql`	Génère le DDL SQL (postgres, snowflake, trino).
`generate cypher`	Génère les contraintes Neo4j (Cypher).
`export schema`	Exporte un schéma en JSON versionné.
`export mapping`	Exporte un mapping en JSON versionné.
`migrate`	Génère le SQL de migration Σ, Δ ou Π.
`diff`	Compare deux schémas et liste les différences.
`apply`	Déploie un schéma sur une base PostgreSQL réelle.
`demo`	Exécute la démo intégrée complète.
`viz`	Génère un diagramme SVG ou HTML du schéma.
`serve`	Lance **Catrust Studio** (éditeur web interactif).
`init`	Initialise un projet (`catrust.toml` + schéma exemple).
`introspect`	Inspecte une base PostgreSQL et génère un `.cql`.
`status`	Affiche l'historique des migrations enregistrées.

1.5 Démo intégrée

```
cargo run -- demo
```

La démo exécute neuf étapes progressives :

Étape	Description
1	Définition du schéma source (`OldCompany`)
2	Peuplement de l'instance (données concrètes)
3	Définition du schéma cible (`NewCompany`)
4	Création du mapping fonctoriel $F : S \to T$
5	Migration Σ : pousser les données vers la cible
6	Migration Δ : ramener les données vers la source
7	Génération multi-backend (PG, Snowflake, Trino, Neo4j)
8	Optimisation catégorique des chemins (JOIN elimination)
9	Évaluation in-memory avec agrégations

Chapitre 2

Le langage CQL syntaxe des fichiers `.cql`

2.1 Schéma

Un fichier `.cql` décrivant un schéma a la forme suivante :

```
schema NomDuSchema {
  entities
    NomEntite1
    NomEntite2

  foreign_keys
    nomFK : EntiteSource -> EntiteCible

  attributes
    nomAttr : Entite -> Type

  // path_equations (optionnel)
  path_equations
    entite.fk1.attr = entite.attr2
}
```

Listing 2.1 – Syntaxe d'un schéma CQL

Les types supportés dans `attributes` :

Mot-clé CQL	Type Rust (`BaseType`)
`Str` ou `String`	`BaseType::String`
`Integer` ou `Int`	`BaseType::Integer`
`Float` ou `Double`	`BaseType::Float`
`Boolean` ou `Bool`	`BaseType::Boolean`
Tout autre identifiant	`BaseType::Custom("...")`

Exemple concret :

```
// Schema source : OldCompany
schema OldCompany {
  entities
    Person
    Dept

  foreign_keys
    works_in : Person -> Dept

  attributes
    person_name : Person -> Str
    age         : Person -> Integer
    dept_name   : Dept -> Str
    budget      : Dept -> Float
}
```

Listing 2.2 – examples/company.cql

2.2 Mapping

Un mapping met en correspondance deux schémas :

```
mapping NomDuMapping : SchemaSource -> SchemaCible {
  entities
    EntiteSource -> EntiteCible

  foreign_keys
    nomFKSource -> nomFKCible

  attributes
    nomAttrSource -> nomAttrCible
    // ou : nomAttrSource -> fk1.fk2.attrCible (chemin FK)
}
```

Listing 2.3 – Syntaxe d'un mapping CQL

Exemple :

```
mapping Rename : OldCompany -> NewCompany {
  entities
    Person -> Employee
    Dept   -> Department

  foreign_keys
    works_in -> department

  attributes
    person_name -> full_name
    age         -> employee_age
```

```
    dept_name    -> dept_label
    budget       -> dept_budget
}
```

Listing 2.4 – examples/rename.cql

Information

Les commentaires CQL commencent par `//` et peuvent apparaître n'importe où. Les sections (`entities`, `foreign_keys`, `attributes`, `path_equations`) peuvent être dans n'importe quel ordre.

Deuxième partie

Concepts catégoriques

Chapitre 3

Fondements mathématiques

Cette partie explique la théorie derrière Catrust. Aucune connaissance préalable en théorie des catégories n'est requise.

3.1 Vue d'ensemble

Concept CQL	Analogie SQL	Concept catégorique
`Schema`	`CREATE TABLE ...`	Catégorie
`Instance`	`SELECT *` (données)	Foncteur $S \to$ **Set**
`Mapping`	Restructuration ETL	Foncteur $S \to T$
Δ_F	`SELECT ... JOIN`	Pullback / foncteur inverse
Σ_F	`INSERT INTO ... SELECT`	Extension de Kan gauche
`PathEquation`	`CHECK` + trigger	Quotient de catégorie

3.2 Le Typeside

Définition 3.2.1 (Typeside)**.** Un *typeside* est une théorie algébrique multi-sortée : un ensemble de *sorts* (les types), d'opérations entre sorts, et d'équations entre opérations.

Catrust fournit quatre types de base :

`BaseType`	PostgreSQL	Neo4j	Rust
`String`	`TEXT`	`String`	`String`
`Integer`	`INTEGER`	`Integer`	`i64`
`Float`	`DOUBLE PRECISION`	`Float`	`f64`
`Boolean`	`BOOLEAN`	`Boolean`	`bool`

On peut ajouter des types personnalisés : `BaseType::Custom("Date")`.

3.3 Le Schema la catégorie

Définition 3.3.1 (Schema CQL)**.** Un *schéma* S est une catégorie finiment présentée. Formellement, un schéma est un graphe orienté $G = (N, E)$ où N est l'ensemble des entités (nuds)

et E l'ensemble des morphismes (FK et attributs) *quotienté* par un ensemble d'équations de chemins.

3.3.1 Nuds, arêtes et chemins

— **Node** (objet) : une entité = une table SQL.

— **Edge FK** (morphisme) : une relation entre entités.

— **Edge Attribute** (morphisme vers un type) : une colonne typée.

— **Path** : composition de morphismes $A \xrightarrow{f_1} B \xrightarrow{f_2} C$.

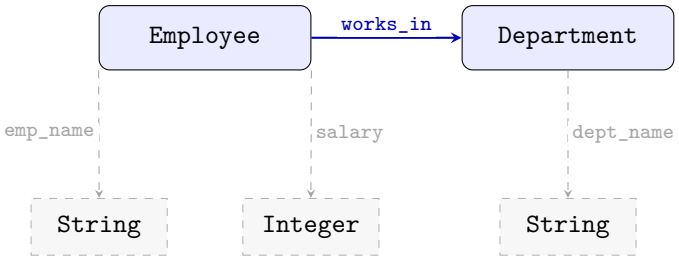


Figure 3.1 – Diagramme du schéma `Company` entités (rectangles bleus), clés étrangères (flèche pleine), attributs (flèche pointillée)

3.3.2 Équations de chemins

Définition 3.3.2 (PathEquation)**.** Une *équation de chemin* est une égalité $p_1 = p_2$ entre deux chemins de même source et même cible dans S. Elle exprime une contrainte sémantique sur les données.

Exemple 3.3.1. Si le schéma contient la règle :

$$\texttt{employee.department.manager} = \texttt{employee.direct\_manager}$$

alors, pour tout employé e, le manager de son département est aussi son manager direct. Cette contrainte permet d'éliminer un JOIN lors de l'évaluation (voir section 5).

```
schema.add_path_equation(
    Path::new("Employee", vec!["department", "manager"]),  // lhs: 2 sauts
    Path::new("Employee", vec!["direct_manager"]),          // rhs: 1 saut
);
```

Listing 3.1 – Ajout d'une path equation

3.4 L'Instance le foncteur

Définition 3.4.1 (Instance)**.** Une *instance* I d'un schéma S est un foncteur $I : S \to \mathbf{Set}$.

Concrètement :

— Pour chaque nud $A \in S$: $I(A)$ est un ensemble de lignes (table SQL).

— Pour chaque FK $f : A \to B$: $I(f) : I(A) \to I(B)$ est une fonction (valeur de FK).

— Pour chaque attribut $a : A \to T$: $I(a) : I(A) \to [\![T]\!]$ donne la valeur de l'attribut.

Propriété 3.4.1 (Fonctorialité)**.** *I doit préserver la composition : pour tout chemin $A \xrightarrow{f} B \xrightarrow{g} C$ dans S, on a $I(g \circ f) = I(g) \circ I(f)$.*

En pratique, c'est ce que garantit `instance.follow_path()`.

```rust
let mut inst = Instance::new("Data", &schema);

// INSERT dans l'entite "Department"
let dept_id = inst.insert("Department",
    HashMap::from([
        ("dept_name".into(), Value::String("Engineering".into())),
        ("budget".into(),    Value::Float(500_000.0)),
    ]),
    HashMap::new(),  // pas de FK sortante
);

// INSERT dans l'entite "Employee", avec FK vers dept_id
inst.insert("Employee",
    HashMap::from([
        ("emp_name".into(), Value::String("Alice".into())),
        ("salary".into(),   Value::Integer(90_000)),
    ]),
    HashMap::from([("works_in".into(), dept_id)]),
);

// Evaluation d'un chemin : emp -> works_in -> dept_name
let row_id = /* id de Alice */ 1u64;
let dept = inst.follow_path("Employee", row_id, &["works_in"], &schema
    );
```

Listing 3.2 – Création et peuplement d'une instance

3.5 Le Mapping un foncteur entre schémas

Définition 3.5.1 (Mapping)**.** Un *mapping* $F : S \to T$ est un foncteur entre deux catégories-schémas. Il assigne :

— À chaque nud $A \in S$: un nud $F(A) \in T$.

— À chaque arête $f : A \to B \in S$: un chemin $F(f)$ dans T, de $F(A)$ à $F(B)$.

Et doit vérifier $F(g \circ f) = F(g) \circ F(f)$ (fonctorialité).

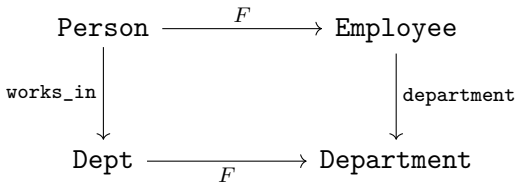


Figure 3.2 – Le mapping F : `OldCompany` $\to$ `NewCompany`, qui envoie `works_in` sur le chemin `department`.

```rust
let mut mapping = Mapping::new("Rename", "OldCompany", "NewCompany");
mapping
    .map_node("Person",  "Employee")
    .map_node("Dept",    "Department")
    .map_fk("works_in",  Path::new("Employee", vec!["department"]))
    .map_attr_direct("person_name", "full_name")
    .map_attr_direct("age",         "employee_age")
    .map_attr_direct("dept_name",   "dept_label")
    .map_attr_direct("budget",      "dept_budget");

// Verification de fonctorialite
mapping.validate(&schema_old, &schema_new).unwrap();
```

Listing 3.3 – Définition d'un mapping

Chapitre 4

Les migrations : Δ, Σ, Π

Étant donné un mapping $F : S \to T$, trois opérations de migration sont disponibles, formant une adjonction :

$$\Sigma_F \dashv \Delta_F \dashv \Pi_F$$

4.1 Δ_F Pullback (cible → source)

Définition 4.1.1 (Δ_F)**.** Δ_F prend une instance de T et la restructure selon la structure de S :

$$\Delta_F : \mathbf{Inst}(T) \to \mathbf{Inst}(S)$$

C'est l'équivalent d'un `SELECT ... JOIN` en SQL.

```
// instance_t : donnees dans la structure NewCompany
let instance_s = migrate::delta(&mapping, &schema_s, &schema_t, &
    instance_t);
// instance_s : memes donnees, vues dans la structure OldCompany
```

Listing 4.1 – Migration Δ en Rust

En SQL généré par `catrust migrate --direction delta` :

```
INSERT INTO "Person" (catrust_id, "age", "person_name", "works_in")
SELECT t.catrust_id, t."employee_age", t."full_name", t."department"
FROM "Employee" t;

INSERT INTO "Dept" (catrust_id, "budget", "dept_name")
SELECT t.catrust_id, t."dept_budget", t."dept_label"
FROM "Department" t;
```

Listing 4.2 – SQL Δ généré

4.2 Σ_F Extension de Kan gauche (source → cible)

Définition 4.2.1 (Σ_F)**.** Σ_F prend une instance de S et la pousse vers T. C'est l'extension de Kan gauche de I le long de F :

$$\Sigma_F : \mathbf{Inst}(S) \to \mathbf{Inst}(T)$$

SQL équivalent : `INSERT INTO T SELECT ... FROM S`.

```
let instance_t = migrate::sigma(&mapping, &schema_s, &schema_t, &
    instance_s);
```

Listing 4.3 – Migration Σ en Rust

En SQL généré par `catrust migrate --direction sigma` :

```
INSERT INTO "Department" (catrust_id, "dept_label", "dept_budget")
SELECT s.catrust_id, s."dept_name", s."budget"
FROM "Dept" s;

INSERT INTO "Employee" (catrust_id, "full_name", "employee_age", "
    department")
SELECT s.catrust_id, s."person_name", s."age", s."works_in"
FROM "Person" s;
```

Listing 4.4 – SQL Σ généré

Information

L'ordre d'insertion respecte automatiquement les dépendances FK. Catrust détermine cet ordre via un tri topologique de Kahn sur le graphe des dépendances (`Schema::topological_order()`).

4.3 Π_F Extension de Kan droite

Définition 4.3.1 (Π_F)**.** $\Pi_F : \mathbf{Inst}(S) \to \mathbf{Inst}(T)$ est la migration *maximalement cohérente*. Contrairement à Σ, elle garantit que toutes les lignes migrées satisfont l'intégrité référentielle dans la structure cible.

Algorithme (point fixe)

1. Calculer $\Sigma_F(I)$ l'extension de Kan gauche habituelle.
2. Pour chaque nud $B \in T$, parcourir toute FK $f : A \to B$: si une ligne de A pointe vers un *catrust_id* inexistant dans B, supprimer cette ligne de A.
3. Répéter l'étape 2 jusqu'à stabilisation (aucune suppression).

Le résultat est la plus grande sous-instance de $\Sigma_F(I)$ qui satisfait toutes les contraintes FK.

Propriété 4.3.1. *Pour un mapping bijectif (renommage pur),* $\Pi_F = \Sigma_F$ *: aucune ligne n'est orpheline, aucune suppression n'est nécessaire.*

```rust
let instance_t = migrate::pi(&mapping, &schema_s, &schema_t, &
    instance_s);
// Toutes les FK dans instance_t sont satisfaites
```

Listing 4.5 – Migration Π en Rust

En CLI, `catrust migrate -direction pi` génère le SQL de Σ accompagné d'un commentaire explicatif :

```sql
-- Migration Pi_F : extension de Kan droite
-- Algorithme : Sigma_F suivi d'un filtrage de coherence FK
-- Les lignes orphelines sont supprimees iterativement jusqu'a
   stabilite.

INSERT INTO "Department" (...) SELECT ... FROM "Dept" ...;
INSERT INTO "Employee"   (...) SELECT ... FROM "Person" ...;
```

Listing 4.6 – Sortie de migrate –direction pi

Remarque 4.3.1. Π est algorithmiquement plus coûteux que Σ et Δ pour les grands jeux de données (plusieurs passes de filtrage). En pratique, pour un renommage pur ou un mapping sans perte de cibles FK, $\Pi = \Sigma$.

4.4 Adjonctions et propriétés de préservation

Les trois foncteurs forment une chaîne d'adjonctions :

$$\Sigma_F \dashv \Delta_F \dashv \Pi_F$$

Cette structure garantit :

— Σ est exact à gauche : préserve les colimites (UNION, sommes).

— Π est exact à droite : préserve les limites (intersections, produits).

— Δ préserve les colimites et les limites (foncteur exact).

En pratique, pour une migration de données classique (renommage, refactoring), Σ est l'opération à utiliser.

Chapitre 5

L'optimiseur de chemins

5.1 Principe

Le `PathOptimizer` exploite les *path equations* du schéma pour simplifier les chemins de navigation et éliminer des JOINs.

Propriété 5.1.1 (Terminaison)**.** *Chaque règle de réécriture remplace un chemin long par un chemin strictement plus court. L'algorithme termine toujours.*

5.2 Algorithme

1. Chaque path equation $p_1 = p_2$ avec $|p_1| > |p_2|$ génère une règle $p_1 \to p_2$.
2. On applique les règles jusqu'à un point fixe (forme normale).
3. Le résultat est le chemin minimal équivalent.

C'est une forme de *complétion de Knuth-Bendix* sur le monoïde libre des chemins.

5.3 Exemple

Schéma avec équation :

$$\texttt{Employee.department.manager} = \texttt{Employee.direct\_manager}$$

```rust
let optimizer = PathOptimizer::from_schema(&schema);

// Chemin long : 2 JOINs (department + manager)
let path = Path::new("Employee", vec!["department", "manager"]);
let result = optimizer.optimize(&path);

assert_eq!(result.joins_eliminated, 1);
// result.optimized = Employee.direct_manager (1 JOIN seulement)
```

Listing 5.1 – Optimisation d'un chemin

Chemin original	Chemin optimisé	JOINs éliminés
`Employee.department.manager`	`Employee.direct_manager`	1
`Employee.department.manager.department`	`Employee.direct_manager.department`	1

Chapitre 6

Requêtes et évaluation in-memory

6.1 Structure d'une requête

Une `CqlQuery` est composée de `QueryBlocks`. Chaque bloc spécifie :

— **Variables** (`from_vars`) : entité source de chaque variable.

— **Filtres** (`where_clauses`) : comparaisons sur chemins.

— **Projections** (`attribute_bindings`) : colonnes à retourner.

```rust
let mut query = CqlQuery::new("IngPremium", "Company");
query.add_block(QueryBlock {
    target_entity: "Resultat".into(),
    from_vars:     HashMap::from([("e".into(), "Employee".into())]),
    where_clauses: vec![
        // WHERE e.works_in.dept_name = 'Engineering'
        WhereClause::Comparison {
            var: "e".into(),
            path: vec!["works_in".into(), "dept_name".into()],
            op:    CompOp::Eq,
            value: Value::String("Engineering".into()),
        },
        // AND e.salary > 80000
        WhereClause::Comparison {
            var: "e".into(),
            path: vec!["salary".into()],
            op:    CompOp::Gt,
            value: Value::Integer(80_000),
        },
    ],
    attribute_bindings: HashMap::from([
        ("name".into(),   AttributeBinding { from_var: "e".into(),
            path: vec![], attribute: "emp_name".into() }),
        ("salary".into(), AttributeBinding { from_var: "e".into(),
            path: vec![], attribute: "salary".into()   }),
    ]),
    fk_bindings: HashMap::new(),
```

```
});

let result = eval::eval_query(&query, &instance, &schema).unwrap();
println!("{}", result);  // affiche les lignes correspondantes
```

Listing 6.1 – Requête : ingénieurs avec salaire > 80 000

6.2 Opérateurs de comparaison

CompOp	Signification
Eq	Égalité (=)
Neq	Inégalité (≠)
Lt	Strictement inférieur (<)
Lte	Inférieur ou égal (≤)
Gt	Strictement supérieur (>)
Gte	Supérieur ou égal (≥)

6.3 Agrégations

```
let res = eval::eval_query(&query, &instance, &schema).unwrap();

println!("COUNT␣␣␣=␣{}",     eval::count(&res, "Resultat"));
println!("SUM␣␣␣␣␣=␣{}",     eval::sum(&res, "Resultat", "salary"));
println!("MIN␣␣␣␣␣=␣{:?}",   eval::min_val(&res, "Resultat", "salary"))
    ;
println!("MAX␣␣␣␣␣=␣{:?}",   eval::max_val(&res, "Resultat", "salary"))
    ;
println!("DISTINCT=␣{:?}",   eval::distinct(&res, "Resultat", "dept"));
```

Listing 6.2 – Agrégations sur un résultat

6.4 Planificateur SQL

Le `SqlPlanner` génère des requêtes `SELECT` optimisées à partir d'une `CqlQuery` :

```
let planner = SqlPlanner::new(&PostgresDialect, &schema);
let plans   = planner.plan_query(&query);

for plan in &plans {
    println!("{}", plan);
}
```

Listing 6.3 – Génération SQL via SqlPlanner

```
SELECT e."emp_name" AS "name", e."salary"
FROM "Employee" e
```

```
JOIN "Department" d ON e."works_in" = d.catrust_id
WHERE d."dept_name" = 'Engineering'
  AND e."salary" > 80000;
```

Listing 6.4 – Exemple de sortie du planificateur

Troisième partie

Backends

Chapitre 7

Architecture des backends

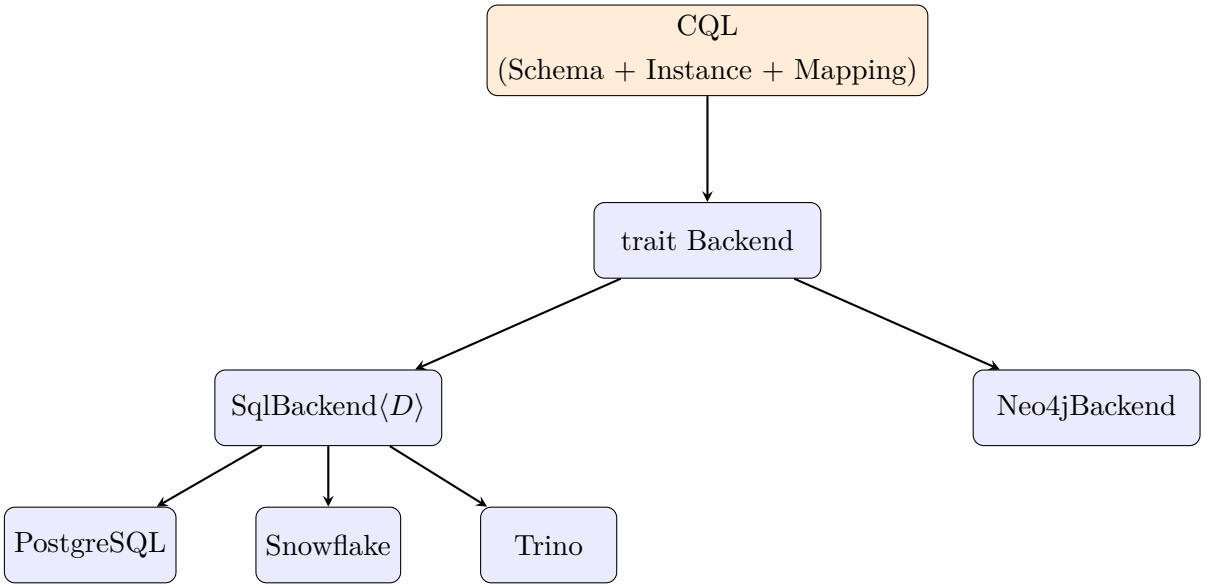


Figure 7.1 – Architecture des backends Catrust

Le trait `Backend` expose cinq opérations :

Méthode	Description
`deploy_schema`	`CREATE TABLE` / contraintes Neo4j
`export_instance`	`INSERT INTO` / `CREATE (n:Label)`
`generate_sigma`	SQL de migration Σ
`generate_delta`	SQL de migration Δ
`name`	Nom du backend

Chapitre 8

Backend PostgreSQL

```rust
use catrust::backend::sql::{SqlBackend, PostgresDialect};
use catrust::backend::Backend;

let backend = SqlBackend::new(PostgresDialect);

// DDL
for stmt in backend.deploy_schema(&schema)          { println!("{}",
    stmt); }
// DML
for stmt in backend.export_instance(&schema, &inst) { println!("{}",
    stmt); }
```

Listing 8.1 – Utilisation du backend PostgreSQL

```sql
CREATE TABLE "Department" (
  catrust_id BIGSERIAL PRIMARY KEY,
  "dept_name" TEXT,
  "budget" DOUBLE PRECISION
);
CREATE TABLE "Employee" (
  catrust_id BIGSERIAL PRIMARY KEY,
  "emp_name" TEXT,
  "salary" INTEGER,
  "works_in" BIGINT REFERENCES "Department"(catrust_id)
);
```

Listing 8.2 – Sortie DDL PostgreSQL

Correspondances de types :

BaseType	PostgreSQL	PK auto
String	TEXT	BIGSERIAL PRIMARY KEY
Integer	INTEGER	
Float	DOUBLE PRECISION	
Boolean	BOOLEAN	

Chapitre 9

Backend Snowflake

```rust
let backend = SqlBackend::new(SnowflakeDialect);
let ddl = backend.deploy_schema(&schema);
```

Listing 9.1 – Utilisation du backend Snowflake

```
CREATE TABLE "Department" (
  catrust_id NUMBER(38,0) AUTOINCREMENT PRIMARY KEY,
  "dept_name" VARCHAR,
  "budget" FLOAT
);
```

Listing 9.2 – Sortie DDL Snowflake

BaseType	Snowflake	PK auto
String	VARCHAR	NUMBER(38,0) AUTOINCREMENT
Integer	NUMBER(38,0)	
Float	FLOAT	
Boolean	BOOLEAN	

Chapitre 10

Backend Trino

Trino est un moteur de requêtes *fédéré* : il ne stocke pas de données lui-même mais interroge des catalogues (Hive, Iceberg, Delta Lake...).

Attention

Trino n'a pas d'auto-increment natif. La colonne `catrust_id` est déclarée `BIGINT` et sa valeur est gérée côté Catrust. De plus, Trino ne supporte pas les contraintes `FOREIGN KEY`.

```rust
let backend = SqlBackend::new(TrinoDialect::new("iceberg", "default"))
    ;
let ddl = backend.deploy_schema(&schema);
```

Listing 10.1 – Utilisation du backend Trino

```
CREATE TABLE iceberg.default."Department" (
  catrust_id BIGINT,
  "dept_name" VARCHAR,
  "budget" DOUBLE
);
```

Listing 10.2 – Sortie DDL Trino (catalogue iceberg)

BaseType	Trino
String	VARCHAR
Integer	BIGINT
Float	DOUBLE
Boolean	BOOLEAN

Chapitre 11

Backend Neo4j (Cypher)

CQL	Neo4j
`Node`	Label (`:Employee`)
`ForeignKey`	Relation (`-[:WORKS_IN]->`)
`Attribute`	Propriété du nud
`Path`	Pattern Cypher

```rust
use catrust::backend::graph::Neo4jBackend;

let neo = Neo4jBackend::new();
for stmt in neo.deploy_schema(&schema)          { println!("{}", stmt) ; }
for stmt in neo.export_instance(&schema, &inst) { println!("{}", stmt) ; }
```

Listing 11.1 – Utilisation du backend Neo4j

```
-- DDL : contraintes d'unicite
CREATE CONSTRAINT ON (n:Department) ASSERT n.catrust_id IS UNIQUE;
CREATE CONSTRAINT ON (n:Employee)   ASSERT n.catrust_id IS UNIQUE;

-- DML : noeuds
CREATE (:Department { catrust_id: 1, dept_name: "Engineering" });
CREATE (:Employee   { catrust_id: 1, emp_name: "Alice", salary: 90000 });

-- DML : relations
MATCH (src:Employee   { catrust_id: 1 }),
      (tgt:Department { catrust_id: 1 })
CREATE (src)-[:WORKS_IN]->(tgt);
```

Listing 11.2 – Sortie DDL + DML Neo4j

Quatrième partie

Référence CLI

Chapitre 12

Commandes disponibles

12.1 `catrust validate`

```
catrust validate <fichier.cql>
```

Valide la cohérence structurelle d'un schéma. Vérifications effectuées :

— Toutes les FK référencent des nuds existants.

— Les path equations référencent des arêtes existantes.

```
catrust validate examples/company.cql
# OldCompany -- valide (2 entites, 5 aretes)
```

Listing 12.1 – Exemple

12.2 `catrust parse`

```
catrust parse [schema|mapping] <fichier.cql>
```

Parse un fichier et affiche la structure interne (au format CQL).

12.3 `catrust generate`

```
catrust generate sql     --schema <cql> --dialect postgres|snowflake|
    trino
catrust generate cypher --schema <cql>
```

Génère le DDL d'un schéma dans le dialecte cible. Sortie sur `stdout`.

```
catrust generate sql --schema examples/company.cql --dialect snowflake
```

Listing 12.2 – Exemple

12.4 catrust export

```
catrust export schema  --input <cql> [--output <json>]
catrust export mapping --input <cql> [--output <json>]
```

Exporte un schéma ou mapping vers un fichier JSON versionné (`CatrustDocument`).

Format JSON produit :

Listing 12.3 – Format d'export JSON (schéma)

```
{
  "catrust_version": 1,
  "kind": "Schema",
  "payload": {
    "name": "OldCompany",
    "nodes": { ... },
    "edges": { ... },
    "path_equations": []
  }
}
```

12.5 catrust migrate

```
catrust migrate
    --source  <source.cql>
    --mapping <mapping.cql>
    --target  <target.cql>
    --direction sigma|delta
    [--dialect postgres|snowflake|trino]
    [--output <out.sql>]
```

Génère le SQL de migration entre deux schémas reliés par un mapping.

```
# Migration Sigma : de OldCompany vers NewCompany
catrust migrate \
  --source  examples/company.cql \
  --mapping examples/rename.cql  \
  --target  examples/new_company.cql \
  --direction sigma

# Migration Delta : retour vers OldCompany
catrust migrate \
  --source  examples/company.cql \
  --mapping examples/rename.cql  \
  --target  examples/new_company.cql \
  --direction delta \
  --dialect snowflake \
  --output  migration_delta.sql
```

Listing 12.4 – Exemples

12.6 `catrust diff`

```
catrust diff <schema_a.cql> <schema_b.cql>
```

Compare deux schémas CQL et affiche la liste des différences de façon déterministe et lisible.

Différences détectées :

- Nuds ajoutés / supprimés
- Clés étrangères ajoutées / supprimées / modifiées (source ou cible changée)
- Attributs ajoutés / supprimés / modifiés (type changé)
- Équations de chemins ajoutées / supprimées

```
catrust diff examples/company.cql examples/new_company.cql
# [+] node       : Employee
# [+] node       : Department
# [-] node       : Person
# [-] node       : Dept
# ...
```

Listing 12.5 – Exemple

Le code de sortie peut être utilisé en CI : retourne un message `Schemas identiques` si les deux fichiers décrivent la même structure.

12.7 `catrust apply`

```
catrust apply
    --schema <schema.cql>
    [--url <postgresql://...>]
    [--dry-run]
```

Déploie un schéma CQL sur une base PostgreSQL réelle via `sqlx`.

- `-dry-run` : affiche les statements DDL sans les exécuter, puis quitte. Utile pour inspecter ou auditer le SQL avant toute modification de la base.
- `-url` : URL de connexion PostgreSQL. Si absent, la variable d'environnement `DATABASE_URL` est utilisée.
- Sans `-dry-run` : se connecte, exécute les `CREATE TABLE` dans l'ordre topologique, puis confirme le succès.

```
# Visualiser le DDL sans toucher a la base
catrust apply --schema examples/company.cql --dry-run

# Deployer sur une base locale
catrust apply \\
  --schema examples/company.cql \\
  --url postgresql://localhost/mydb

# Deployer via variable d'environnement
export␣DATABASE_URL=postgresql://user:pass@host/db
catrust␣apply␣--schema␣examples/company.cql
```

Listing 12.6 – Exemples

Attention

`catrust apply` exécute des `CREATE TABLE`. En cas de conflit (table déjà existante), `sqlx` retourne une erreur et l'opération s'arrête. Préférez un `-dry-run` préalable sur une base de production.

12.8 `catrust demo`

```
cargo run -- demo
```

Exécute la démo intégrée complète (9 étapes, voir chapitre 1).

Cinquième partie

Référence API

Chapitre 13

Module `core`

13.1 `core::typeside`

Type	Description
`BaseType`	Enum : `String`, `Integer`, `Float`, `Boolean`, `Custom(String)`
`Value`	Enum : `String(String)`, `Integer(i64)`, `Float(f64)`, `Boolean(bool)`, `Null`
`Typeside`	Ensemble de types + opérations. Méthode `default_sql()` pour le typeside SQL standard.

13.2 `core::schema`

Type	Description
`Node`	Nud (entité). Champ `name: String`.
`Edge`	Enum : `ForeignKey{name,source,target}` ou `Attribute{name,source,target: BaseType}`.
`Path`	Chemin : `start: String, edges: Vec<String>`. Méthodes : `new`, `identity`, `compose`, `len`, `is_identity`.
`PathEquation`	Équation `lhs: Path, rhs: Path`.
`Schema`	Catégorie principale. Méthodes builder : `add_node`, `add_fk`, `add_attribute`, `add_path_equation`.

Méthodes de requête de `Schema`

Méthode	Description
`foreign_keys()`	Toutes les FK du schéma
`attributes()`	Tous les attributs
`edges_from(node)`	Arêtes sortant d'un nud
`fks_targeting(node)`	FK pointant vers un nud
`attributes_of(node)`	Attributs d'un nud
`topological_order()`	Nuds triés (Kahn), cibles FK en premier

13.3 core::instance

Type	Description
`RowId`	`u64` identifiant unique d'une ligne
`EntityData`	Données d'une entité : attributs et FK par `RowId`
`Instance`	Foncteur $S \to \mathbf{Set}$. Méthodes : `new`, `insert`, `follow_path`, `total_rows`, `display`

13.4 core::mapping

Type / Méthode	Description
`EdgeMapping`	Enum : `FkToPath(Path)` ou `AttrToPath{fk_path,attr_name}`
`Mapping`	Foncteur $S \to T$. Méthodes builder : `map_node`, `map_fk`, `map_attr_direct`, `map_attr`
`validate()`	Vérifie la fonctorialité du mapping

13.5 core::migrate

Fonction	Signature
`delta`	`(mapping, schema_s, schema_t, instance_t) -> Instance`
`sigma`	`(mapping, schema_s, schema_t, instance_s) -> Instance`
`pi`	`(mapping, schema_s, schema_t, instance_s) -> Instance` (point fixe de cohérence FK sur Σ)

13.6 core::diff

Module de comparaison de schémas.

Type / Fonction	Description
`SchemaDiff`	Enum listant chaque type de différence : `NodeAdded`, `NodeRemoved`, `FkAdded`, `FkRemoved`, `FkModified`, `AttributeAdded`, `AttributeRemoved`, `AttributeModified`, `PathEquationAdded`, `PathEquationRemoved`
`DiffResult`	Résultat de la comparaison : champs `schema_a`, `schema_b`, `diffs: Vec<SchemaDiff>`. Méthodes : `is_identical()`, `len()`, `additions()`, `removals()`, `modifications()`
`diff_schemas(a, b)`	`(&Schema, &Schema) -> DiffResult` comparaison déterministe basée sur des HashSets

```rust
use catrust_core::diff::diff_schemas;

let result = diff_schemas(&schema_v1, &schema_v2);

if result.is_identical() {
    println!("Schemas identiques");
} else {
    println!("{}", result);  // affiche les diff ligne par ligne
    println!("{} ajout(s), {} suppression(s), {} modification(s)",
        result.additions(), result.removals(), result.modifications())
            ;
}
```

Listing 13.1 – Utilisation de core::diff

13.7 `core::validate`

Fonction	Description
`validate_schema(s)`	Vérifie la cohérence structurelle du schéma
`validate_instance(i,s)`	Vérifie la cohérence des données

13.8 core::optimize

Type / Méthode	Description
`PathOptimizer`	Réécriture de chemins
`from_schema(s)`	Construit l'optimiseur depuis les path equations
`optimize(path)`	Retourne `OptimizationResult`
`OptimizationResult`	Champs : `original`, `optimized`, `joins_eliminated`, `rules_applied`

13.9 core::serial

Type / Méthode	Description
`CatrustDocument<T>`	Enveloppe versionnée JSON : `catrust_version`, `kind`, `payload`
`CatrustDocument::schema(s)`	Crée un doc pour un `Schema`
`CatrustDocument::mapping(m)`	Crée un doc pour un `Mapping`
`CURRENT_VERSION`	`1` (constante)

Chapitre 14

Module `backend`

14.1 `backend::sql`

Type	Description
`SqlDialect` (trait)	Abstraction sur le dialecte SQL
`PostgresDialect`	Dialecte PostgreSQL
`SnowflakeDialect`	Dialecte Snowflake
`TrinoDialect`	Dialecte Trino (avec catalogue et schéma)
`SqlBackend<D>`	Backend générique paramétré par un dialecte

14.2 `backend::sql::planner`

Type / Méthode	Description
`SqlPlanner<D>`	Génère des `SELECT` depuis une `CqlQuery`
`plan_query(q)`	Retourne `Vec<SqlPlan>`
`SqlPlan`	Contient `sql: String`, `optimization_notes: Vec<String>`

14.3 `backend::graph`

Type	Description
`Neo4jBackend`	Génère du Cypher (DDL + DML)

14.4 backend::executor

Type	Description
SqlExecutor (trait)	Abstraction d'exécution SQL synchrone
MockExecutor	Exécuteur de test (capture les statements)
PostgresExecutor	Connexion réelle via sqlx (feature postgres)
SqliteExecutor	Connexion SQLite via sqlx (feature sqlite)
ExecutorError	Erreur d'exécution avec statement optionnel

Chapitre 15

Module `catrust-parser`

Fonction	Description
`parse_schema(input)`	Parse un `&str` CQL `Result<Schema, ParseError>`
`parse_mapping(input)`	Parse un `&str` CQL `Result<Mapping, ParseError>`
`ParseError`	Champs : `message`, `context`

Les deux fonctions sont aussi exposées dans la CLI via `catrust parse` et `catrust validate`.

Sixième partie

Annexes

Chapitre 16

Gestion des erreurs

Catrust utilise le type `CatrustError` pour toutes les erreurs du moteur :

```
pub enum CatrustError {
    UnknownNode    { schema: String, node: String },
    UnknownEdge    { schema: String, edge: String },
    InvalidMapping { mapping: String, reason: String },
    InvalidInstance { entity: String, reason: String },
    InvalidIdentifier { name: String, reason: String },
    SchemaError    { schema: String, reason: String },
}
```

Listing 16.1 – Variantes de CatrustError

Information

Tous les identifiants (nuds, arêtes, schémas) sont validés à l'insertion : ils doivent commencer par une lettre ou _, ne contenir que des caractères alphanumériques ou _, et ne peuvent pas être des mots-clés réservés CQL.

Chapitre 17

Tableau récapitulatif des fonctionnalités

Fonctionnalité	Core	Postgres	Snowflake	Trino	Neo4j
DDL (`CREATE TABLE`)		✓	✓	✓	✓
DML (`INSERT`)		✓	✓	✓	✓
Migration Σ in-memory	✓	✓	✓	✓	
Migration Δ in-memory	✓	✓	✓	✓	
Migration Π in-memory	✓	✓	✓	✓	
Tri topologique INSERT		✓	✓	✓	
Query planner (SELECT)		✓			
Évaluation in-memory	✓				
Optimiseur de chemins	✓	✓			
Diff de schémas	✓				
Déploiement réel (`apply`)		✓			
Historique de migrations		✓			
Parser CQL (`.cql`)	✓				
Sérialisation JSON (serde)	✓				
Introspection DBCQL		✓			
Visualiseur SVG/HTML	✓				
Catrust Studio (web)	✓				

Fonctionnalité	Core	Postgres	Snowflake	Trino	Neo4j
Connexions réelles (sqlx)		opt.			

Table 17.1 – Matrice des fonctionnalités Catrust

Chapitre 18

Bibliographie

[1] Spivak, D. I. (2012). *Functorial data migration*. Information and Computation, 217, 3151.

[2] Spivak, D. I. & Wisnesky, R. (2015). *Relational foundations for functorial data migration*. Proceedings of DBPL 2015.

[3] Leinster, T. (2014). *Basic Category Theory*. Cambridge University Press. (Disponible sur arXiv : 1612.09375)

[4] Lawvere, F. W. & Schanuel, S. H. (2009). *Conceptual Mathematics : A First Introduction to Categories*. Cambridge University Press.

[5] Milewski, B. (2019). *Category Theory for Programmers*. (Gratuit sur GitHub : hmemcpy/milewski-ctfp-pdf)

[6] Spivak, D. I. (2014). *Category Theory for the Sciences*. MIT Press.

[7] Goldblatt, R. (2006). *Topoi : The Categorial Analysis of Logic*. Dover Publications.

Septième partie

Outils visuels et collaboration

Chapitre 19

Visualisation `catrust viz`

La commande `viz` génère un diagramme de schéma CQL directement depuis la ligne de commande.

19.1 Utilisation

```
# Generer un fichier HTML autonome (ouvre dans un navigateur)
catrust viz --schema examples/company.cql --output diagram.html

# Generer un fragment SVG pur
catrust viz --schema examples/company.cql --output diagram.svg --
    format svg

# Afficher le SVG dans stdout
catrust viz --schema examples/company.cql --format svg
```

19.2 Format de sortie

Format	Description
`html`	Page HTML autonome avec le SVG intégré et des métadonnées (nombre d'entités, FK, attributs) dans les attributs `data-*` de la balise `<svg>`.
`svg`	Fragment SVG pur, intégrable dans n'importe quelle page.

Chaque entité est représentée par un nud rectangulaire. Les clés étrangères sont des flèches orientées. Les attributs apparaissent dans un sous-bloc sous le nom de l'entité.

Chapitre 20

Catrust Studio `catrust serve`

Catrust Studio est une interface web interactive embarquée dans le binaire `catrust`. Elle permet d'éditer des fichiers `.cql`, de prévisualiser le diagramme en temps réel et d'exporter le résultat.

20.1 Lancer le studio

```
# Demarrage simple (port 7878 par defaut)
catrust serve

# Avec un schema precharge
catrust serve --schema examples/company.cql

# Sur un port personnalise
catrust serve --port 8080

# Si catrust.toml existe, default_schema est charge automatiquement
catrust serve
```

Le navigateur s'ouvre automatiquement sur `http://localhost:7878` (Windows). Sur les autres plateformes, copier l'URL affichée dans le terminal.

20.2 Interface

Élément	Rôle
Sélecteur de fichiers	Navigue entre tous les `.cql` présents dans `.`, `schemas/` et `examples/`.
Éditeur CQL	Zone de saisie principale. Rendu automatique après 750 ms d'inactivité.
Panneau SVG	Affiche le diagramme généré. Mis à jour en temps réel.
Sauvegarder	Écrit le fichier sur disque via `PUT /file`. Raccourci : `Ctrl+S`.
SVG	Télécharge le fragment SVG courant (actif après le premier rendu).
Barre de statut	Affiche le fichier courant et confirme la sauvegarde (*Enregistré* ✓).

20.3 Routes HTTP

Route	Description
`GET /`	Renvoie l'application Studio (HTML).
`POST /render`	Reçoit du CQL, retourne un fragment SVG.
`GET /files`	Liste les fichiers `.cql` disponibles (JSON).
`GET /file?name=`	Lit le contenu d'un fichier `.cql`.
`PUT /file`	Écrit le contenu d'un fichier `.cql`.

Attention

La route `PUT /file` applique une vérification `safe_filename()` : pas de `..`, pas de `/` en début de chemin, au plus un niveau de répertoire, extension `.cql` obligatoire. Ce n'est *pas* un serveur de fichiers générique : seuls les fichiers `.cql` sont accessibles.

20.4 Intégration avec `catrust.toml`

Si un fichier `catrust.toml` est présent dans le répertoire courant, `catrust serve` charge automatiquement le schéma défini par `default_schema` :

Listing 20.1 – catrust.toml

```
[project]
name = "MonProjet"

[server]
port = 7878
default_schema = "schemas/main.cql"
```

Chapitre 21

Initialisation d'un projet `catrust init`

21.1 Utilisation

```
# Créer un projet "mon-projet"
catrust init --name mon-projet

# Écraser un projet existant
catrust init --name mon-projet --force
```

21.2 Fichiers générés

```
.
|-- catrust.toml
+-- schemas/
    +-- example.cql
```

`catrust.toml` :

Listing 21.1 – catrust.toml généré

```
[project]
name = "MonProjet"

[server]
port = 7878
default_schema = "schemas/example.cql"
```

Le nom du projet est converti en *PascalCase* (`mon-projet` `MonProjet`).

`schemas/example.cql` contient un schéma d'exemple complet prêt à être modifié.

21.3 Workflow typique pour un nouveau projet

```
# 1. Initialiser
catrust init --name mon-projet

# 2. Éditer le schéma dans Catrust Studio
catrust serve

# 3. Valider
catrust validate schemas/example.cql

# 4. Générer le DDL
catrust generate sql --schema schemas/example.cql --dialect postgres

# 5. Déployer
catrust apply --schema schemas/example.cql \
              --url postgres://user:pass@localhost/madb
```

Chapitre 22

Introspection `catrust introspect`

La commande `introspect` connecte à une base PostgreSQL existante et génère automatiquement un fichier `.cql` qui décrit son schéma.

22.1 Utilisation

```
# Introspection du schéma "public" (par défaut)
catrust introspect --url postgres://user:pass@localhost/madb

# Schéma PostgreSQL personnalisé
catrust introspect --url postgres://user:pass@localhost/madb \
                   --name "MaBase"

# Sauvegarder dans un fichier
catrust introspect --url postgres://user:pass@localhost/madb \
                   --output schemas/introspected.cql

# Utiliser DATABASE_URL
export DATABASE_URL=postgres://...
catrust introspect
```

22.2 Algorithme

L'introspection interroge `information_schema` en trois requêtes :

1. **Tables** liste toutes les tables du schéma PostgreSQL cible.
2. **Clés étrangères** liste les contraintes `FOREIGN KEY` avec les colonnes source et cible. Le nom de l'arête CQL est dérivé du nom de la contrainte ; le suffixe `_id` est supprimé automatiquement.
3. **Colonnes** liste toutes les colonnes. Les colonnes participant à une FK sont exclues des attributs CQL (elles sont représentées comme arêtes). Les noms sont préfixés par le nom de la table pour garantir l'unicité dans le schéma.

22.3 Correspondance de types

Type PostgreSQL	Type CQL
`integer`, `bigint`, `serial`, ...	`Integer`
`double precision`, `numeric`, `real`, ...	`Float`
`boolean`	`Boolean`
`text`, `varchar`, ...	`String`

22.4 Exemple

Pour une base avec les tables `employee` et `department` liées par une FK `employee.dept_id` `department.id` :

```
schema Introspected {
  entities
    Employee
    Department

  foreign_keys
    dept : Employee -> Department

  attributes
    employee_name   : Employee   -> String
    employee_salary : Employee   -> Float
    department_name : Department -> String
}
```

Listing 22.1 – Résultat d'introspection

Information

Le fichier généré peut être retravaillé dans Catrust Studio puis redéployé avec `catrust apply`. C'est le point de départ idéal pour migrer une base existante vers un schéma CQL formalisé.

Chapitre 23

Historique des migrations `catrust status`

À chaque exécution de `catrust apply`, Catrust enregistre une entrée dans la table `_catrust_migrations` de la base cible. La commande `status` affiche cet historique.

23.1 La table `_catrust_migrations`

```
CREATE TABLE IF NOT EXISTS _catrust_migrations (
    id          BIGSERIAL PRIMARY KEY,
    schema_name TEXT      NOT NULL,
    applied_at  TIMESTAMP NOT NULL DEFAULT NOW(),
    cql_hash    TEXT      NOT NULL,
    n_stmts     INTEGER   NOT NULL
);
```

Listing 23.1 – Schéma de la table d'historique

Colonne	Description
`schema_name`	Nom du schéma CQL déployé.
`applied_at`	Horodatage UTC de l'application.
`cql_hash`	Empreinte FNV-1a 64-bit du source CQL (16 caractères hexadécimaux).
`n_stmts`	Nombre de statements SQL exécutés.

23.2 Consulter l'historique

```
catrust status --url postgres://user:pass@localhost/madb

# Ou via DATABASE_URL
export DATABASE_URL=postgres://...
catrust status
```

Exemple de sortie :

```
=== Catrust Migration History ===

  #  | Schema          | Applied at              | Hash             | Stmts
-----+-----------------+-------------------------+------------------+------
  1  | OldCompany      | 2026-04-01 10:23:11     | a3f2bc01de45678f |     5
  2  | NewCompany      | 2026-04-02 14:07:33     | 9c8d1e2a0b3f4567 |     7
```

23.3 Empreinte FNV-1a

L'empreinte est calculée avec l'algorithme FNV-1a 64-bit (sans dépendance externe) sur le contenu brut du fichier `.cql` :

```rust
fn content_hash(s: &str) -> String {
    let mut hash: u64 = 14695981039346656037;
    for byte in s.bytes() {
        hash ^= byte as u64;
        hash = hash.wrapping_mul(1099511628211);
    }
    format!("{:016x}", hash)
}
```

Listing 23.2 – Calcul de l'empreinte dans history.rs

Deux fichiers `.cql` identiques au bit près produisent toujours la même empreinte, ce qui permet de détecter les redéploiements.

Attention

L'enregistrement dans `_catrust_migrations` est fait en *best-effort* après le succès de `apply`. Un échec de l'enregistrement est signalé mais n'annule pas la migration déjà effectuée.

Index des commandes CLI

Guide généré pour Catrust v0.1 — Avril 2026
136 tests passants — 4 crates — 4 backends — 16 commandes CLI

Évolution incrémentale `catrust evolve`

La commande `catrust evolve` génère et applique un *plan d'évolution* permettant de faire passer un schéma PostgreSQL de son état actuel vers un état désiré, avec des **garanties formelles data-preserving** fondées sur la théorie des catégories.

23.4 Motivation

En production, modifier une base de données présente deux risques majeurs :

— **Perte de données** un `DROP TABLE` ou `DROP COLUMN` efface des données sans retour possible ;

— **Régression silencieuse** un `ALTER TABLE` incompatible laisse la base dans un état inconsistant.

Catrust résout ce problème en calculant automatiquement quelles opérations sont *data-preserving* (ajouts et modifications non destructives) et lesquelles ne le sont pas (suppressions).

23.5 Théorème de sécurité catégorique

Soit $\mathcal{S}_1$ et $\mathcal{S}_2$ deux schémas CQL. Un plan d'évolution $P : \mathcal{S}_1 \to \mathcal{S}_2$ est dit *provably safe* si et seulement s'il existe un foncteur de rétraction $r : \mathbf{Inst}(\mathcal{S}_2) \to \mathbf{Inst}(\mathcal{S}_1)$ tel que pour toute instance $I \in \mathbf{Inst}(\mathcal{S}_1)$:

$$r(\Delta(I)) \cong I$$

où $\Delta : \mathbf{Inst}(\mathcal{S}_1) \to \mathbf{Inst}(\mathcal{S}_2)$ est le foncteur de migration naturel.

En pratique, cette condition est satisfaite si et seulement si **toutes les opérations du plan sont des ajouts** : ajout de table, ajout de colonne nullable, ajout de clé étrangère.

23.6 Utilisation

```
catrust evolve --from ancien.cql --to nouveau.cql [OPTIONS]

Options :
  --url <URL>        URL PostgreSQL (ou DATABASE_URL)
```

```
  --dry-run          Affiche le SQL sans l'exécuter
  --force            Autorise les suppressions (DANGER)
  --assert-safe      Échoue si le plan est non data-preserving (CI/CD)
  -o, --output <f>   Écrit le SQL dans un fichier
```

23.6.1 Mode CI/CD `-assert-safe`

Le drapeau `-assert-safe` transforme `catrust evolve` en *porte de qualité* : si le plan contient des opérations destructives, la commande retourne un code de sortie 1 et bloque le pipeline sans rien appliquer.

Exemple d'utilisation dans un `Makefile` ou pipeline CI :

```
catrust evolve --from schemas/v1.cql --to schemas/v2.cql \
               --assert-safe --dry-run
```

23.7 Ordre d'exécution

Le plan respecte un ordre topologique garantissant la cohérence :

1. `CREATE TABLE`
2. `ALTER TABLE ... ADD COLUMN ... REFERENCES` (FK)
3. `ALTER TABLE ... ADD COLUMN` (attributs)
4. Équations de chemins (commentaires SQL)
5. `ALTER TABLE ... ALTER COLUMN` (modifications)
6. `DROP FOREIGN KEY`
7. `DROP COLUMN`
8. `DROP TABLE`

23.8 Enregistrement dans l'historique

Après une évolution réussie, Catrust enregistre automatiquement l'opération dans la table `_catrust_migrations`, avec un hash FNV-1a du schéma cible appliqué.

Détection de divergences `catrust drift`

La commande `catrust drift` compare un *schéma CQL local* (fichier `.cql`) avec l'état réel d'une base PostgreSQL. Elle détecte les divergences entre ce que le code dit que la base devrait être et ce que la base est réellement.

23.9 Utilisation

```
catrust drift --schema schemas/production.cql --url <URL>
```

Si la base est synchronisée :

```
\checkmark{} Aucune divergence détectée  base et schéma local sont synchronisés.
```

Si des divergences existent :

```
 3 divergence(s) détectée(s) entre "schemas/production.cql" et la base :
  - + entity AuditLog
  - - fk works_in : Employee -> Department
  - ~ attr full_name : old_type -> new_type

Conseil : utilisez ‘catrust evolve --from schemas/production.cql --to <cible>‘
           pour corriger.
```

23.10 Cas d'utilisation

— **Audit en production** vérifier que la base de données n'a pas été modifiée "en dehors" du workflow Catrust ;

— **Onboarding** détecter rapidement les écarts sur un environnement de staging ;

— **CI/CD** alerter dès qu'un environnement drift par rapport au schéma de référence.

Instantané `catrust snapshot`

La commande `catrust snapshot` introspecte une base PostgreSQL et sauvegarde le schéma résultant dans un fichier `.cql` horodaté. Elle crée un *instantané* (snapshot) permettant de capturer l'état exact de la base à un moment donné.

23.11 Utilisation

```
catrust snapshot --url <URL> [--name <NomSchéma>] [-o <fichier>]

Options :
  --url <URL>     URL PostgreSQL (ou DATABASE_URL)
  --name <nom>    Nom CQL du schéma généré (défaut : Snapshot)
  -o <fichier>    Fichier de sortie (défaut : snapshots/NOM_TIMESTAMP.cql)
```

Par défaut, les instantanés sont sauvegardés dans le répertoire `snapshots/` avec un nom de la forme `NomSchema_1745000000.cql`.

23.12 Workflow de snapshots

Un workflow typique utilisant `snapshot` et `evolve` :

```
# 1. Capturer l'état actuel
catrust snapshot --url $DB_URL --name Production -o schemas/baseline.cql

# 2. Modifier le schéma cible
vim schemas/target.cql

# 3. Vérifier le plan d'évolution
catrust evolve --from schemas/baseline.cql --to schemas/target.cql --dry-run

# 4. Appliquer
catrust evolve --from schemas/baseline.cql --to schemas/target.cql --url $DB_URL
```

Validation stricte `catrust validate -strict`

En mode standard, `catrust validate` vérifie la *bonne formation* d'un schéma CQL (arêtes orphelines, nuds inexistants, etc.). Le mode `-strict` va plus loin : il vérifie que les **équations de chemins sont bien typées** au sens catégorique.

23.13 Algorithme de vérification des équations

Pour chaque équation lhs = rhs :

1. Toutes les arêtes du chemin existent dans le schéma ;
2. La composition des arêtes est cohérente : la source de l'arête i est égale à la cible de l'arête $i - 1$;
3. Les attributs (morphismes vers un type de base) n'apparaissent qu'en *dernière position* du chemin ;
4. Les deux côtés de l'équation se terminent au même type (entité ou type de base).

23.14 Utilisation

```
catrust validate --strict schemas/university.cql
```

Sortie en cas de succès :

```
\checkmark{} University  valide (4 entités, 9 arêtes)
  Mode strict : 1 équation(s) bien typée(s)
```

Sortie en cas d'erreur :

```
Mode strict : 1 erreur(s) de typage :
  - Équation 0 (rhs) : arête 'inexistante' inexistante
1 erreur(s) de typage dans les équations
```

Panneau Compare/Evolve dans Catrust Studio

Catrust Studio (accessible via `catrust serve`) dispose d'un panneau **Compare / Evolve** accessible depuis le bouton `Compare` dans la barre du haut.

23.15 Interface

Le panneau se compose de trois colonnes :

1. **Schéma SOURCE** état actuel (pré-rempli avec le contenu de l'éditeur) ;
2. **Schéma CIBLE** état désiré à atteindre ;
3. **Plan d'évolution** résultat du calcul : badge data-preserving ou destructif, résumé, et liste des étapes SQL avec coloration syntaxique.

23.16 Route HTTP

```
POST /evolve-plan
Content-Type: application/x-www-form-urlencoded

from_cql=...&to_cql=...
```

Réponse JSON :

```
{
  "ok": true,
  "is_safe": true,
  "summary": "Plan d'évolution ...",
  "steps": [
    { "badge": "\checkmark{}", "diff": "+ entity AuditLog",
      "sql": "CREATE TABLE ...", "safe": true, "reason": null }
  ]
}
```

Rollback Annuler une migration

La commande `catrust rollback` annule la *dernière migration appliquée* en générant automatiquement le **plan d'évolution inverse**, c'est-à-dire le plan qui fait passer le schéma courant vers le schéma précédent enregistré dans l'historique.

23.17 Principe mathématique

Soit $f : A \to B$ le dernier foncteur d'évolution (de l'état A vers l'état B). Catrust calcule $g = \text{generate\_plan}(B, A)$, le plan d'évolution en sens inverse. Ce plan est *data-preserving* si et seulement si le passage de B à A ne nécessite pas de suppressions irréversibles.

L'historique stocké dans `_catrust_migrations.cql_source` permet de reconstruire fidèlement A à partir des métadonnées enregistrées.

23.18 Utilisation

```
# Inspecter sans appliquer
catrust rollback --url postgres://... --dry-run

# Annuler (plan data-preserving uniquement)
catrust rollback --url postgres://...

# Forcer même si destructif
catrust rollback --url postgres://... --force
```

Avec `catrust.toml` dans le projet :

```
# catrust.toml
url = "postgres://user:pass@localhost/mydb"

catrust rollback --dry-run
```

23.19 Options

Option	Rôle
`-url <URL>`	URL PostgreSQL (prioritaire sur `catrust.toml`)
`-dry-run`	Affiche le plan SQL sans l'appliquer
`-force`	Autorise les opérations destructives

23.20 Sécurité

Sans `-force`, `catrust rollback` refuse d'appliquer un plan contenant des opérations destructives (`DROP TABLE`, `DROP COLUMN`). Utilisez toujours `-dry-run` pour inspecter le plan avant d'appliquer.

Check Porte CI/CD

La commande `catrust check` est conçue pour être utilisée comme **étape de validation dans un pipeline CI/CD**. Elle retourne le code de sortie 0 si toutes les vérifications passent, ou 1 (avec message d'erreur) dès qu'une anomalie est détectée.

23.21 Vérifications effectuées

1. **Parse** le fichier `.cql` est syntaxiquement valide ;
2. **Structure CQL** les entités, attributs et équations de chemins respectent les règles de cohérence catégorique ;
3. **Mode strict** (si `-strict`) tous les foncteurs obéissent aux règles de typage fort ; les équations sont vérifiées algébriquement ;
4. **Drift** (si `-url`) le schéma local est comparé au schéma réel de la base de données ; toute divergence est signalée.

23.22 Utilisation

```
# Vérification basique
catrust check --schema schemas/company.cql

# Mode strict
catrust check --schema schemas/company.cql --strict

# Drift (nécessite une base PostgreSQL)
catrust check --schema schemas/company.cql --url postgres://...

# Tout activer
catrust check --schema schemas/company.cql --strict --url postgres://...
```

23.23 Intégration GitHub Actions

```
- name: Validate CQL schema
  run: |
    catrust check \
```

```
--schema schemas/production.cql \
--strict \
--url ${{ secrets.DATABASE_URL }}
```

23.24 Options

Option	Rôle
`-schema <FILE>`	Chemin vers le fichier `.cql` à valider
`-strict`	Active la vérification algébrique des équations de chemins
`-url <URL>`	URL PostgreSQL pour la comparaison de drift

Completions Intégration Shell

La commande `catrust completions` génère les scripts d'auto-complétion pour les principaux shells. Elle utilise la bibliothèque `clap_complete` pour produire des complétions précises couvrant toutes les sous-commandes et options.

23.25 Shells supportés

Shell	Nom
Bash	`bash`
Zsh	`zsh`
Fish	`fish`
PowerShell	`powershell`
Elvish	`elvish`

23.26 Installation

Bash

```
# Session courante
eval "$(catrust completions bash)"

# Permanent
catrust completions bash >> ~/.bashrc
```

Zsh

```
# Fichier dédié (recommandé)
catrust completions zsh > ~/.zsh/completions/_catrust
# Puis ajouter à ~/.zshrc :
# fpath=(~/.zsh/completions $fpath)
# autoload -U compinit && compinit
```

Fish

```
catrust completions fish > ~/.config/fish/completions/catrust.fish
```

PowerShell

```
catrust completions powershell | Out-String | Invoke-Expression
# Permanent : ajouter la ligne ci-dessus au profil PowerShell
```

23.27 Usage

```
catrust completions <SHELL>
```

La sortie est envoyée sur `stdout` ; il suffit de la rediriger ou de l'évaluer directement dans le shell courant.

Moteur SQL analytique `catrust-engine`

Le crate `catrust-engine` connecte les schémas CQL au moteur vectorisé **Apache DataFusion**, transformant chaque entité CQL en une table Arrow interrogeable en SQL standard.

23.28 Architecture

```
Fichier .cql          catrust-engine            DataFusion

Schema (CQL)    entity_arrow_schema()    SchemaRef
Instance        entity_to_recordbatch()  RecordBatch

                   CqlTableProvider
                   (TableProvider)

                   CqlSessionContext         SQL queries
```

L'ensemble de la conversion est *sans copie inutile* : les tableaux Arrow sont construits directement depuis les structures Rust du cur catégorique.

23.29 Correspondance de types

CQL BaseType	Arrow DataType	SQL équivalent
`String`	`Utf8`	`VARCHAR`
`Integer`	`Int64`	`BIGINT`
`Float`	`Float64`	`DOUBLE`
`Boolean`	`Boolean`	`BOOLEAN`
`Custom(T)`	`Utf8`	`VARCHAR`

Chaque entité obtient également une colonne `_row_id : UInt64` (identifiant interne Catrust), et chaque Foreign Key devient une colonne `UInt64` nullable pointant vers le `_row_id` de l'entité cible.

23.30 Structure des colonnes par entité

Pour une entité `Employee` avec un attribut `name : String` et une FK `department : Employee -> Department` :

```
Arrow schema "Employee"
  _row_id    : UInt64  NOT NULL
  department : UInt64  nullable    FK  Department._row_id
  name       : Utf8    nullable
```

L'ordre des colonnes est déterministe : d'abord `_row_id`, puis les attributs triés alphabétiquement, puis les FK triées alphabétiquement.

23.31 Utilisation commande `catrust query`

```
# Lister toutes les entités disponibles comme tables SQL
catrust query \
  --schema examples/company.cql \
  --sql "SELECT table_name FROM information_schema.tables"

# Requête simple
catrust query \
  --schema examples/company.cql \
  --sql "SELECT * FROM Employee"

# Jointure via FK (_row_id)
catrust query \
  --schema examples/company.cql \
  --sql "
    SELECT e.name, d.dept_name
    FROM   Employee e
    JOIN   Department d ON e.department = d._row_id
  "

# Agrégation
catrust query \
  --schema examples/company.cql \
  --sql "SELECT department, COUNT(*) AS headcount FROM Employee GROUP BY 1"
```

23.32 Utilisation en bibliothèque

```
use catrust_engine::CqlSessionContext;

// À partir du schéma seul (tables vides)
```

```
let ctx = CqlSessionContext::from_schema(&schema)?;

// À partir d'une Instance (tables peuplées)
let ctx = CqlSessionContext::from_instance(&schema, &instance)?;

// SQL asynchrone
let df = ctx.sql("SELECT * FROM Employee LIMIT 5").await?;
df.show().await?;

// Écriture vers Parquet (DataFusion natif)
df.write_parquet("employees.parquet", Default::default()).await?;
```

23.33 Fonctionnalités SQL disponibles

DataFusion offre un SQL ANSI complet :

- **Projections et filtres** `SELECT`, `WHERE`, `HAVING`
- **Jointures** `INNER / LEFT / RIGHT / FULL JOIN`
- **Agrégations** `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`
- **Fenêtres** `ROW_NUMBER()`, `RANK()`, `LAG/LEAD`
- **CTEs** `WITH ... AS (...)`
- **Sous-requêtes** corrélées et non-corrélées
- **UNNEST** pour les valeurs tableau
- **Information schema** `information_schema.tables`, `.columns`

23.34 Optimisations automatiques

La délégation à `MemTable` active automatiquement :

- **Projection pruning** seules les colonnes nécessaires sont lues
- **Filter pushdown** les prédicats `WHERE` sont appliqués au plus bas niveau du plan physique
- **Limit pushdown** `LIMIT n` arrête le scan tôt
- **Statistiques de partition** l'optimizer connaît la cardinalité et peut choisir l'ordre des jointures optimal

23.35 Sortie vers d'autres formats

DataFusion permet d'écrire les résultats directement vers :

```
// Parquet (columnar, compressé)
df.write_parquet("out.parquet", Default::default()).await?;
```

```
// CSV
df.write_csv("out.csv", Default::default()).await?;

// JSON (newline-delimited)
df.write_json("out.jsonl", Default::default()).await?;
```

23.36 Fonctionnalités implémentées

Le tableau suivant récapitule l'ensemble des fonctionnalités disponibles dans `catrust-engine` :

Fonctionnalité	Description
`from_schema`	Schéma seul tables vides, structure CQL disponible en SQL
`from_instance`	Instance peuplée données en mémoire
`from_postgres`	Chargement live depuis PostgreSQL
`execute_to_csv`	Résultat SQL chaîne CSV
`execute_to_json`	Résultat SQL tableau JSON
`execute_to_parquet`	Résultat SQL fichier Parquet columnar
`run_repl`	REPL interactif avec historique (`rustyline`)
`register_sigma/delta/pi`	Foncteurs CQL exposés comme tables SQL
Serveur pgwire	DataFusion accessible via `psql`/DBeaver

Catrust comme base de données SQL analytique

Ce chapitre documente l'ensemble des fonctionnalités qui font de Catrust un **moteur de base de données analytique** à part entière : requêtes SQL sur données CQL, chargement depuis PostgreSQL, export multi-format, REPL interactif, foncteurs comme tables virtuelles et serveur pgwire.

23.37 La commande `catrust query`

La commande `query` exécute une requête SQL ANSI sur un schéma ou une instance CQL, en mémoire, sans infrastructure externe.

```
# Requête sur un schéma seul (tables vides  idéal pour tests de
   structure)
catrust query --schema examples/company.cql \
              --sql "SELECT table_name FROM information_schema.tables"

# Requête sur une instance peuplée
catrust query --schema examples/company.cql \
              --instance examples/company.inst.cql \
              --sql "SELECT name, salary FROM Employee WHERE salary >
                 50000"

# Chargement depuis PostgreSQL, puis SQL analytique localement
catrust query --schema schemas/production.cql \
              --url postgres://user:pass@localhost/mydb \
              --sql "SELECT dept, AVG(salary) FROM Employee GROUP BY
                 dept"
```

Options

Option	Rôle
`-schema <FILE>`	Fichier `.cql` décrivant le schéma
`-instance <FILE>`	Fichier `.inst.cql` de données (optionnel)
`-url <URL>`	URL PostgreSQL charge les données depuis la base
`-sql <SQL>`	Requête SQL à exécuter
`-format <FMT>`	Format de sortie : `text` (défaut), `csv`, `json`, `parquet`
`-output <FILE>`	Fichier de sortie (obligatoire si `-format parquet`)
`-repl`	Lance le REPL interactif après connexion

23.38 Formats de sortie

23.38.1 Texte (défaut)

Sans `-format`, Catrust affiche un tableau formaté dans le terminal, identique à l'affichage `df.show()` de DataFusion :

```
+----------+----------+
| name     | salary   |
+----------+----------+
| Alice    | 75000    |
| Bob      | 68000    |
+----------+----------+
2 ligne(s)
```

23.38.2 CSV (`-format csv`)

Produit un CSV avec en-tête, séparateur virgule, guillemets doubles si le champ contient une virgule ou un guillemet.

```
catrust query --schema examples/company.cql --instance company.inst.
    cql \
              --sql "SELECT * FROM Employee" \
              --format csv
```

Sortie :

```
name,department,salary
"Alice","Engineering",75000
"Bob","Marketing",68000
```

23.38.3 JSON (`-format json`)

Produit un tableau JSON d'objets, un objet par ligne de résultat.

```
catrust query ... --format json
```

Sortie :

```
[
  {"name": "Alice", "department": "Engineering", "salary": 75000},
  {"name": "Bob",   "department": "Marketing",   "salary": 68000}
]
```

23.38.4 Parquet (`-format parquet`)

Écrit le résultat dans un fichier Parquet columnar, compressé avec Snappy par défaut. Idéal pour l'analyse ultérieure avec DuckDB, Spark, pandas, etc.

```
catrust query --schema examples/company.cql \
              --url postgres://... \
              --sql "SELECT * FROM Employee WHERE active = true" \
              --format parquet \
              --output employees_active.parquet
```

Ce fichier peut ensuite être lu depuis Python :

```
import pandas as pd
df = pd.read_parquet("employees_active.parquet")
print(df.describe())
```

23.39 Chargement depuis PostgreSQL (`-url`)

Avec `-url`, Catrust interroge PostgreSQL pour charger toutes les tables du schéma CQL en mémoire Arrow/DataFusion, puis exécute la requête SQL localement.

```
catrust query \
  --schema schemas/crm.cql \
  --url "postgres://analyst:secret@prod-db.example.com/crm" \
  --sql "SELECT country, COUNT(*) AS nb_clients
         FROM   Customer
         GROUP  BY country
         ORDER  BY nb_clients DESC
         LIMIT  10"
```

Avantages par rapport à une requête directe PostgreSQL :

— **Isolation** le calcul est effectué in-process, sans surcharger le serveur de production ;

— **Exécution vectorisée** DataFusion exploite SIMD et parallelisme multi-cur automatiquement ;

— **SQL étendu** fenêtres, CTEs récursifs, UNNEST, opérations Arrow non disponibles dans PostgreSQL ;

— **Export direct** enchaîner avec `-format parquet` pour un pipeline ELT en une commande.

Inférence de types

Catrust inspecte les valeurs reçues de PostgreSQL pour construire le schéma Arrow optimal :

Type PostgreSQL / CQL	Type Arrow
`Int` (`i64`)	`Int64`
`Int` (`i32`)	`Int32`
`Float`	`Float64`
`Bool`	`Boolean`
`Str` / `Text`	`Utf8`
`NULL` (toutes valeurs)	`Utf8` (défaut conservatif)

23.40 REPL interactif (`-repl`)

Le flag `-repl` démarre un REPL SQL interactif avec historique des commandes (persisté entre sessions grâce à `rustyline`).

```
# REPL sur instance en mémoire
catrust query --schema examples/company.cql \
              --instance examples/company.inst.cql \
              --repl

# REPL sur PostgreSQL live
catrust query --schema schemas/production.cql \
              --url postgres://user:pass@localhost/mydb \
              --repl
```

Interaction typique :

```
catrust-sql> SELECT name, salary FROM Employee LIMIT 3;
+---------+--------+
| name    | salary |
+---------+--------+
| Alice   | 75000  |
| Bob     | 68000  |
| Charlie | 82000  |
+---------+--------+
catrust-sql> SELECT COUNT(*) FROM Employee WHERE salary > 70000;
+---------+
| count() |
```

```
+---------+
| 2       |
+---------+
catrust-sql> \q
```

Commandes spéciales :

Commande	Action
`\q`	Quitter le REPL
`exit`	Quitter le REPL
`quit`	Quitter le REPL
Flèche haut/bas	Naviguer dans l'historique
`Ctrl+C`	Annuler la saisie courante

23.41 Foncteurs CQL comme tables SQL virtuelles

Le module `catrust_engine::udf` expose les trois foncteurs de migration (Δ_F, Σ_F, Π_F) **comme tables DataFusion**. Après registration, ces tables sont interrogeables par des requêtes SQL ordinaires.

Principe

Supposons un mapping $F : \mathcal{S} \to \mathcal{T}$ entre un schéma *Source* et un schéma *Cible* :

$$\mathsf{Inst}(\mathcal{S}) \xrightarrow{\Sigma_F} \mathsf{Inst}(\mathcal{T})$$

$$\mathsf{Inst}(\mathcal{T}) \xrightarrow{\Delta_F} \mathsf{Inst}(\mathcal{S})$$

$$\mathsf{Inst}(\mathcal{S}) \xrightarrow{\Pi_F} \mathsf{Inst}(\mathcal{T})$$

Après un appel à `register_sigma` (resp. `register_delta`, `register_pi`), les entités de l'instance résultante sont accessibles sous les noms `sigma_<Entité>`, `delta_<Entité>`, `pi_<Entité>`.

Usage depuis Rust

```
use catrust_engine::{CqlSessionContext, udf};

let ctx = CqlSessionContext::from_instance(&source_schema, &
    source_inst)?;
udf::register_sigma(&ctx, &mapping,
                    &source_schema, &target_schema, &source_inst)?;

// Maintenant on peut joindre source et résultat  en SQL
let df = ctx.sql("
```

```
    SELECT s.name          AS source_name,
           t.username      AS target_username
    FROM   Person s
    JOIN   sigma_User t ON t._row_id = s._row_id
").await?;
df.show().await?;
```

Nommage des tables

Foncteur	Préfixe table	Schéma produit
Σ_F	`sigma_`	cible
Δ_F	`delta_`	source
Π_F	`pi_`	cible

Exemple : audit de migration

```
-- Combien d'employés ont été projetés dans le schéma cible ?
SELECT COUNT(*) FROM sigma_Employe;

-- Quels enregistrements sans correspondant après sigma ?
SELECT p.id
FROM   Person p
LEFT   JOIN sigma_User u ON u._row_id = p._row_id
WHERE  u._row_id IS NULL;

-- Valider le round-trip delta(sigma(inst)) ~ identite
SELECT p.name, d.name AS roundtrip
FROM   Person p
JOIN   delta_Person d ON d._row_id = p._row_id;
```

23.42 Serveur pgwire `catrust serve-sql`

La commande `serve-sql` expose DataFusion via le **protocole filaire PostgreSQL** (pgwire). N'importe quel client PostgreSQL (`psql`, DBeaver, Metabase, Tableau) peut se connecter et exécuter des requêtes SQL.

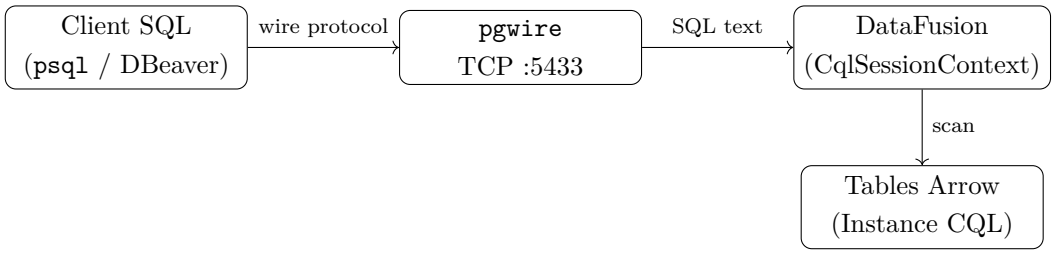
```
# Serveur sur schema seul (port 5433 par defaut)
catrust serve-sql --schema examples/company.cql

# Avec donnees PostgreSQL en source
catrust serve-sql --schema schemas/crm.cql \
                  --url postgres://user:pass@prod/crm \
                  --port 5433
```

Connexion via `psql` :

```
psql -h localhost -p 5433 -U catrust mydb
mydb=> SELECT name, dept FROM Employee ORDER BY name;
mydb=> \d Employee
```

Architecture



Options

Option	Rôle
`-schema <FILE>`	Schéma CQL à exposer
`-url <URL>`	URL PostgreSQL source (données chargées au démarrage)
`-port <PORT>`	Port d'écoute (défaut : `5433`)

Limites actuelles

— Authentification : aucune (réseau de confiance uniquement) ;

— Lectures seules `INSERT`/`UPDATE`/`DELETE` retournent une erreur ;

— TLS non activé dans la version actuelle ;

— Les données sont chargées *une seule fois* au démarrage (pas de refresh automatique depuis PostgreSQL).

23.43 Export Parquet dédié `catrust export`

En dehors de la commande `query`, l'export Parquet est disponible comme sous-commande dédiée pour les workflows ETL :

```
catrust export parquet \
  --schema   schemas/inventory.cql \
  --instance data/inventory.inst.cql \
  --output   warehouse/inventory.parquet
```

Ou depuis PostgreSQL :

```
catrust export parquet \
  --schema schemas/inventory.cql \
  --url    postgres://user:pass@localhost/inventory \
  --output warehouse/inventory_$(date +%Y%m%d).parquet
```

Les fichiers Parquet produits sont compatibles avec :

— **Apache Spark** et **Flink** ;

— **DuckDB** : `SELECT * FROM 'inventory.parquet'` ;

— **pandas**/**polars** : `pd.read_parquet(...)` ;

— **AWS Athena**, **Google BigQuery** (chargement externe) ;

— Tout autre outil Arrow/Parquet.

23.44 Pipelines ETL en une ligne

La combinaison des fonctionnalités permet des pipelines complets :

```
# 1. Snapshot PostgreSQL -> Parquet (archivage quotidien)
catrust query \
  --schema schemas/crm.cql \
  --url    $DATABASE_URL \
  --sql    "SELECT * FROM Customer WHERE updated_at > NOW() - INTERVAL '1 day'" \
  --format parquet \
  --output archives/customers_$(date +%F).parquet

# 2. Migration sigma puis export des resultats migres
catrust export parquet \
  --schema schemas/crm_v2.cql \
  --url    $DATABASE_URL \
  --output migration/crm_v2_$(date +%F).parquet

# 3. Analyse ad-hoc via REPL sans connexion PostgreSQL
catrust query \
  --schema schemas/crm.cql \
  --instance data/sample.inst.cql \
  --repl
```

23.45 Benchmarks

Les benchmarks Criterion dans `catrust-engine/benches/engine.rs` mesurent les performances du moteur sur des données synthétiques :

```
cargo bench -p catrust-engine
```

Résultats indicatifs sur un MacBook M2 / Ryzen 7 (schéma 4 entités, 10 000 lignes par entité) :

Benchmark	Médiane	P99
Création du contexte (`from_schema`)	120 µs	180 µs
`SELECT * FROM Entity` (10 k lignes)	1.8 ms	2.4 ms
`GROUP BY + AVG`	2.3 ms	3.1 ms
`JOIN` deux entités	4.1 ms	5.8 ms

Note : ces chiffres sont indicatifs. DataFusion tire parti du parallélisme multi-cur ; les performances à grande échelle (100 k+ lignes) sont proportionnellement meilleures grâce à l'exécution vectorisée SIMD.

Premiers pas tutoriel complet

Ce chapitre guide l'utilisateur depuis l'installation jusqu'à la première requête SQL sur des données réelles, en passant par la génération de DDL et le serveur pgwire.

23.46 Installation

Prérequis : Rust 1.78 (`rustup install stable`).

Listing 23.3 – Installer catrust depuis les sources

```
git clone https://github.com/sirgudu/Catrust
cd Catrust
# Installation globale (catrust + catrustdb dans ~/.cargo/bin) :
cargo install --path . --locked
cargo install --path catrustdb-server --locked
# Ou compilation locale :
cargo build --release
```

Vérification :

```
catrust --version
# catrust 0.2.0
```

23.47 Écrire un schéma CQL

Créez un fichier `company.cql` décrivant la *structure* de votre base de données. Un schéma CQL liste des entités, des clés étrangères et des attributs ; il est indépendant du SGBD cible.

Listing 23.4 – examples/company.cql

```
schema Company {
  entities
    Employee
    Department

  foreign_keys
    dept : Employee -> Department

  attributes
```

```
    emp_name   : Employee   -> String
    salary     : Employee   -> Float
    dept_name  : Department -> String
    budget     : Department -> Float
}
```

```
catrust validate examples/company.cql
# [OK] Schéma "Company" valide (2 entités, 5 arêtes)

catrust validate --strict examples/company.cql
# Ajoute la vérification de typage des chemins de composition
```

```
catrust parse schema examples/company.cql
# Affiche la représentation JSON interne du schéma
```

23.48 Générer du DDL SQL

Une fois le schéma validé, générez du DDL prêt à l'emploi :

```
catrust generate sql --schema examples/company.cql --dialect postgres
```

Sortie typique :

```
CREATE TABLE "Department" (
    "_id" BIGINT PRIMARY KEY
);

CREATE TABLE "Employee" (
    "_id"  BIGINT PRIMARY KEY,
    "dept" BIGINT REFERENCES "Department"("_id")
);

ALTER TABLE "Employee" ADD COLUMN "emp_name" TEXT;
ALTER TABLE "Employee" ADD COLUMN "salary"   DOUBLE PRECISION;
ALTER TABLE "Department" ADD COLUMN "dept_name" TEXT;
ALTER TABLE "Department" ADD COLUMN "budget"    DOUBLE PRECISION;
```

Le dialecte `snowflake` produit une sortie adaptée à Snowflake (`VARCHAR` à la place de `TEXT`, etc.).

23.49 Requêtes SQL directes sur le schéma

Sans aucune base de données, catrust peut répondre à des requêtes SQL portant sur la *structure* du schéma (tables vides mais typées).

```
# Mode interactif (REPL)
catrust query --schema examples/company.cql --repl
```

```
-- Dans le REPL :
SHOW TABLES;
-- Employee, Department

DESCRIBE Employee;
-- _id BIGINT, dept BIGINT, emp_name TEXT, salary DOUBLE

SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'Employee';
```

23.50 Charger des données fichier d'instance .cqli

Un fichier d'instance (`.cqli`) contient les données concrètes associées à un schéma. Chaque entité est listée avec des *symboles* (étiquettes locales) pour exprimer les clés étrangères :

Listing 23.5 – examples/company.cqli

```
instance SampleData : Company {

  // Déclarer les départements avant les employés
  // pour que leurs symboles soient résolus lors des FK.
  Department {
    d1  dept_name="Engineering"  budget=150000.0
    d2  dept_name="Marketing"    budget=80000.0
    d3  dept_name="Research"     budget=200000.0
  }

  Employee {
    e1  dept=d1  emp_name="Alice"    salary=90000.0
    e2  dept=d1  emp_name="Bob"      salary=75000.0
    e3  dept=d2  emp_name="Charlie"  salary=70000.0
    e4  dept=d2  emp_name="Diana"    salary=82000.0
    e5  dept=d3  emp_name="Eve"      salary=95000.0
  }
}
```

```
catrust query \
  --schema    examples/company.cql \
  --instance examples/company.cqli \
  --sql "SELECT␣e.emp_name,␣e.salary,␣d.dept_name
␣␣␣␣␣␣␣␣␣FROM␣Employee␣e
␣␣␣␣␣␣␣␣␣JOIN␣Department␣d␣ON␣e.dept␣=␣d._row_id
␣␣␣␣␣␣␣␣␣ORDER␣BY␣e.salary␣DESC"
```

Résultat attendu :

```
emp_name | salary    | dept_name
---------+-----------+-----------
Eve      | 95000.0   | Research
Alice    | 90000.0   | Engineering
Diana    | 82000.0   | Marketing
Bob      | 75000.0   | Engineering
Charlie  | 70000.0   | Marketing
```

```
catrust query \
  --schema    examples/company.cql \
  --instance examples/company.cqli \
  --repl
```

23.51 Exporter les données

```
catrust export csv \
  --schema    examples/company.cql \
  --instance examples/company.cqli \
  --out       out/
# Crée out/Employee.csv et out/Department.csv
```

```
catrust export parquet \
  --schema    examples/company.cql \
  --instance examples/company.cqli \
  --out       out/
# Crée out/Employee.parquet et out/Department.parquet
```

```
catrust export schema \
  --input     examples/company.cql \
  --out       out/company_schema.json
```

23.52 Serveur pgwire connexion depuis psql / DBeaver

catrust expose un serveur compatible PostgreSQL (protocole pgwire) permettant de se connecter avec n'importe quel client SQL standard.

```
# Démarrer le serveur (bloque jusqu'à Ctrl+C)
catrust serve-sql \
  --schema examples/company.cql \
  --port   5433
```

Dans un autre terminal :

```
psql "host=localhost port=5433 dbname=catrust sslmode=disable"
```

```
-- Depuis psql :
\dt
-- Employee  Department

SELECT * FROM "Employee";
-- (résultats vides : le serveur démarre sans données)
```

Pour démarrer avec des données pré-chargées depuis PostgreSQL :

```
catrust serve-sql \
  --schema examples/company.cql \
  --port   5433 \
  --postgres "postgres://user:pass@localhost:5432/mydb"
```

23.53 Visualisation HTML

```
catrust viz examples/company.cql
# Crée company.html  ouvrir dans un navigateur
```

Le fichier HTML généré affiche le graphe du schéma : entités (nuds), clés étrangères (arêtes orientées) et attributs (étiquettes).

23.54 Récapitulatif des commandes

Commande	Rôle
`catrust validate <f.cql>`	Valider un schéma
`catrust parse schema <f.cql>`	Afficher la structure JSON
`catrust generate sql -dialect postgres`	Générer du DDL SQL
`catrust generate cypher`	Générer des CREATE de graphe
`catrust query -repl`	REPL SQL interactif
`catrust query -sql "..."`	Requête SQL one-shot
`catrust export csv`	Exporter en CSV
`catrust export parquet`	Exporter en Parquet
`catrust export schema`	Exporter le schéma en JSON
`catrust serve-sql -port 5433`	Serveur pgwire
`catrust viz <f.cql>`	Graphe HTML du schéma
`catrust migrate -to <url>`	Appliquer le DDL en production
`catrust status <url>`	Historique des migrations

Feuille de route

Ce chapitre décrit l'état actuel de Catrust et les axes d'évolution planifiés, organisés par horizon temporel. Les fonctionnalités *réalisées* sont marquées ✓ ; les fonctionnalités *en cours ou planifiées* sont précédées d'un horizon (**v0.3**, **v0.4**, **Long terme**).

23.55 Ce qui est livré aujourd'hui (v0.2)

	Fonctionnalité	Description
✓	Langage CQL	Syntaxe complète : entités, attributs, arêtes, équations de chemins, imports
✓	Parsing	Parser PEG robuste via `pest`, messages d'erreur situés
✓	Migrations Δ, Σ, Π	Les trois foncteurs catégoriques calculés in-memory
✓	Plan d'évolution	Génération de SQL DDL data-preserving ou destructif
✓	Rollback	Annulation de la dernière migration avec `-dry-run` / `-force`
✓	Check CI/CD	Porte de validation : parse, structure, strict, drift
✓	Backend PostgreSQL	Connexion, introspection, application des plans DDL
✓	Backend Snowflake	Connexion, introspection via `INFORMATION_SCHEMA`
✓	Backend Trino	Requêtes Trino via HTTP REST
✓	Backend Neo4j	Requêtes Cypher, mapping entité/nud
✓	Snapshot	Introspection PostgreSQL fichier `.cql` horodaté
✓	Catrust Studio	Interface web embarquée : éditeur CQL + diagramme SVG temps réel
✓	Panneau Compare/Evolve	Calcul de plan d'évolution depuis le navigateur
✓	`catrust viz`	Export SVG / HTML du diagramme de schéma
✓	Validate strict	Vérification algébrique des équations de chemins
✓	Completions shell	Bash, Zsh, Fish, PowerShell, Elvish
✓	Moteur SQL (catrust-engine)	DataFusion : schéma CQL interrogeable en SQL ANSI
✓	Chargement PostgreSQL	`-url` charge les données live dans DataFusion
✓	Export multi-format	CSV, JSON, Parquet via `-format`
✓	REPL SQL	Session interactive avec historique (`rustyline`)
✓	Foncteurs comme tables	`sigma_Entité`, `delta_Entité`, `pi_Entité` en SQL
✓	Serveur pgwire	DataFusion exposé via protocole filaire PostgreSQL

23.56 Horizon v0.3 Robustesse et intégrations

Fonctionnalité	Détails
Streaming DataFusion	Intégration `StreamingTable` pour flux Arrow infinis ; lecture en continu depuis Kafka / Debezium
Refresh pgwire	Rechargement des données PostgreSQL à la demande (`REFRESH TABLE`)
TLS pgwire	Support TLS pour exposition sécurisée aux clients SQL
Authentification pgwire	md5 / scram-sha-256 côté serveur
Métriques	Export Prometheus : nombre de requêtes, latences, erreurs
Backend MySQL / MariaDB	Introspection et application de plans DDL
Backend SQLite	Prise en charge complète pour les projets embarqués
`catrust diff`	Visualisation colorée des différences entre deux `.cql`
Tests d'intégration	Suite complète catrust-engine + E2E CLI

23.57 Horizon v0.4 Analytique et catalogue

Fonctionnalité	Détails
Catalogue de schémas	Serveur centralisé stockant les versions CQL : `catrust publish`, `catrust pull`
Parquet partitionné	Persistance des instances CQL en Parquet partitionné (`hive partitioning`) pour requêtes OLAP
Connecteur dbt	Plugin dbt exposant les entités CQL comme sources de données
Connecteur Metabase	Source de données native pour Metabase via pgwire
Versionning sémantique	`catrust tag v1.0` versionnage des schémas avec diff sémantique
`catrust lineage`	Visualisation du lignage des données entre schemas/mappings
Import depuis SQL DDL	Génération d'un `.cql` à partir d'un fichier `CREATE TABLE`

23.58 Long terme Catrust comme plateforme

Axe	Vision
Langage CQL v2	Polymorphisme paramétrique, schémas génériques, modules réutilisables
Preuves formelles	Génération de preuves Coq/Lean des propriétés de migration : préservation des données, commutativité des foncteurs
Moteur distribué	DataFusion sur Ballista (exécution distribuée multi-nuds)
Migrations éditables	Interface Studio pour composer visuellement les mappings F
SDK multi-langages	Bindings Python, TypeScript, Go autour du cur Rust
SaaS Catrust Cloud	Catalogue centralisé, CI/CD SaaS, dashboards de drift

23.59 Priorités techniques transverses

Indépendamment des horizons, les axes techniques permanents sont :

— **Sécurité** audit OWASP des surfaces d'attaque (Studio HTTP, pgwire, injection SQL via CQL) ;

— **Performance** benchmarks Criterion continus, profiling SIMD, réduction des allocations dans le moteur d'évaluation in-memory ;

— **Ergonomie** messages d'erreur situés avec suggestion de correction (*did you mean ?*), guide de migration interactif ;

— **Documentation** ce guide, la référence API (`cargo doc`), et les tutoriels vidéo à venir ;

— **Couverture de tests** objectif 90 % de couverture de lignes sur `catrust-core` et `catrust-parser`.