immigrant

01 Data Modelling 2

02 Integrations 30

01

Data Modelling

Immigrant Schema Language 3

Tables 8

Views 13

Foreign Keys 14

SQL Expression Language 16

PostgreSQL 17

Roles, Grants and RLS 20

Extensions 24

Modules 25

Migrations 29

Immigrant Schema Language

Quick Start

Immigrant schema describes your database in a single file, which can be split into modules. Instead of writing raw DDL, you declare types, tables, and constraints. Immigrant figures out what SQL to generate, and what migrations to run when you change things.

Here is a minimal schema:

```
scalar user_id = sql"INTEGER";

enum role {
    admin;
    member;
};

table User {
    user_id @primary_key;
    role;
    name: sql"TEXT";
};
```

This produces a PostgreSQL domain `user_id`, an enum type `role`, and a table with three columns. Notice that `user_id` and `role` columns don't need explicit types -- immigrant resolves them by name.

Core Concepts

Naming

Every item has a **code name** (used in schema references) and an optional **database name** (actual SQL identifier).

```
table UserProfile "user_profiles" {};
scalar user_id "uid" = sql"INTEGER";
enum status "account_status" { active; inactive; };
```

If database name is omitted, naming convention preprocessor transforms code names automatically (e.g. PostgreSQL convention turns `UserProfile` into `user_profiles`).

Renaming the database name in schema triggers `ALTER ... RENAME` in migration. Immigrant tracks identity by code name.

Comments

// comments before items become SQL `COMMENT ON` statements (see PostgreSQL). /// comments produce no SQL output.

```
// Doc comment, becomes COMMENT ON
/// Ignored by parser, for schema authors only
```

Items

Scalars

Scalars define reusable column types. Non-inlined scalars become PostgreSQL `DOMAIN`s with constraints baked in:

```
scalar user_id = sql"INTEGER";
scalar positive_int = sql"INTEGER" @check(_ > 0);
scalar created_at = sql"TIMESTAMPTZ" @default(NOW());
```

@check / @default

CHECK constraints and defaults on the domain. `_` is placeholder for the value. Both use the expression language, not raw SQL:

```
scalar positive_int = sql"INTEGER" @check(_ > 0);
scalar bounded = sql"INTEGER" @check "named" (_ > 0 && _ < 1000);
scalar created_at = sql"TIMESTAMPTZ" @default(NOW());
```

@primary_key / @unique / @index

Propagate to every column using this scalar:

```
scalar user_id = sql"INTEGER" @index;
scalar email = sql"TEXT" @unique;
```

Every column of type `user_id` gets an index, every `email` column gets a unique constraint.

@inline

Skips domain creation. Raw SQL type and constraints expand at point of use:

```
scalar non_zero_int = sql"INTEGER" @check(_ != 0) @inline;
```

Columns using `non_zero_int` get type `INTEGER` with CHECK directly on the table. No domain.

@external

Prevents immigrant from generating CREATE/DROP DOMAIN. For types managed outside immigrant:

```
scalar existing_type = sql"TEXT" @external;
```

Propagation

Scalar attributes propagate to columns using the scalar:

- `@primary_key`, `@unique`, `@index` — always propagate, regardless of inline.
- `@default`, `@check` — propagate only when scalar is `@inline`. Otherwise stay on the domain.

Enums

Enums map to PostgreSQL enum types.

```
enum user_group {
    admin;
    member;
    guest;
```

```
};
```

Variants can have custom SQL values (useful for renaming without migration pain):

```
enum user_group {
    // Stored as 'superuser' in database, referenced as 'admin' in schema
    admin "superuser";
    member;
};
```

Adding a variant appends it with `ALTER TYPE ... ADD VALUE`. Removing a variant recreates the type (PostgreSQL limitation).

Structs

Structs become PostgreSQL composite types. When used as a column type, immigrant automatically generates nullability checks for their fields.

```
struct address {
    street: sql"TEXT";
    city: sql"TEXT";
    zip: sql"TEXT" @check(char_length(_) == 5);
};

table Customer {
    id: sql"INTEGER" @primary_key;
    /// column type is 'address' composite
    address;
};
```

Struct-level checks can reference fields via `_.fieldname`:

```
struct range {
    lo: sql"INTEGER";
    hi: sql"INTEGER";
    @check "lo_le_hi" (_.lo <= _.hi);
};
```

Annotations

Generic annotations (`#...`) attach metadata for code generators. They do not affect SQL output, but are preserved for tools like diesel code generation:

```
#diesel(type="dt::Text",copy,custom)
scalar user_id = sql"TEXT";

#diesel(derive = "async_graphql::Enum")
enum client_kind { ... };
```

Annotations can be placed before any item (scalar, enum, struct, table) or before individual columns and enum variants.

Full Example

```
scalar user_id = sql"TEXT" @check(LEFT(_, 1) == "U" && char_length(_) == 33);
scalar client_id = sql"TEXT" @check(LEFT(_, 1) == "C" && char_length(_) == 33);
scalar scope_id = sql"TEXT" @check(!(_ ~~ "% %"));
scalar created_at = sql"TIMESTAMPTZ" @default(NOW());
scalar updated_at = sql"TIMESTAMPTZ" @default(NOW());
```

```
scalar expires_in = sql"INTEGER" @check(_ > 0);
scalar name = sql"TEXT";
scalar bool = sql"BOOLEAN" @inline;

enum challenge_kind {
    password;
    gpg;
    totp;
};

enum client_kind {
    confidential;
    public;
};

struct pkce_challenge_data {
    method: sql"TEXT";
    challenge: sql"TEXT";
};

@mixin Timestamps {
    created_at;
    updated_at;
};

table User {
    user_id @primary_key;
    created_at;
};

table Profile {
    // Profile data for a user
    user_id @primary_key ~.cascade User;
    name?;
    picture: sql"TEXT"?;
    updated_at;
};

table UserChallenge {
    user_id ~.cascade User;
    challenge_kind;
    check_data: sql"BYTEA";
    search_key: sql"BYTEA"?;

    @primary_key(user_id, challenge_kind, check_data);
    @unique(challenge_kind, search_key);
};

table Client {
    client_id @primary_key;
    name;
    client_kind;
    uses_client_secret: bool;
    uses_pkce: bool;
    created_at;
};

table ClientRedirectUrl {
    client_id @index ~.cascade Client;
    redirect_url: sql"TEXT";
    public: bool @initialize_as(false);

    @primary_key(client_id, redirect_url);
};
```

```
table Audience {
    scope_id @primary_key(audience_id);
    audience_id: sql"TEXT" @index;
    name;
};

table AudienceScope {
    audience_id ~.cascade Audience;
    scope_id;
    @primary_key(audience_id, scope_id);
};

table ClientAllowedScope {
    client_id ~.cascade Client;
    audience_id ~.cascade Audience;
    scope_id;

    @primary_key(client_id, audience_id, scope_id);
    (audience_id, scope_id) ~.cascade AudienceScope;
};

view ActiveUsers = sql"""
    SELECT * FROM {User}
    JOIN {Profile} ON {Profile.user_id} = {User.user_id}
    WHERE {Profile.name} IS NOT NULL
""";
```

Tables

```
table User "users" {
    user_id @primary_key;
    name: sql"TEXT";
    created_at;
};
```

Inside a table block:

1. Columns (with optional type, nullability, attributes, foreign keys)
2. Table-level constraints (`@primary_key`, `@unique`, `@check`, `@index`)
3. Standalone foreign keys
4. Mixin inclusions

Columns

```
table Example {
    column_name: type;
};
```

Type can be omitted. Immigrant resolves it from a scalar/enum/struct with the same name:

```
scalar user_id = sql"TEXT";
enum challenge_kind { password; gpg; totp; };

table UserChallenge {
    /// resolved to scalar user_id
    user_id;
    /// resolved to enum challenge_kind
    challenge_kind;
    /// explicit inline type
    name: sql"TEXT";
    /// explicit inline type
    check_data: sql"BYTEA";
};
```

Nullability

Append `?` for nullable. Without it, columns are `NOT NULL`:

```
table Profile {
    user_id @primary_key;
    /// nullable, type resolved by name
    name?;
    /// nullable, explicit type
    picture: sql"TEXT"?;
};
```

Inline SQL types

Write SQL type directly instead of referencing a scalar:

```
table Example {
    data: sql"BYTEA";
    tags: sql"TEXT[]";
    /// reference another type inside inline SQL
```

```
    scores: sql"{score_type}[]";
};
```

Column Attributes

@primary_key

```
table User {
    user_id @primary_key;
};
```

Multiple columns with `@primary_key` merge into one composite primary key:

```
table SessionIdentity {
    session_id @primary_key;
    user_id @primary_key;
    /// produces PRIMARY KEY(session_id, user_id)
};
```

A column can also list additional columns to include in the primary key:

```
table AudienceScope {
    scope_id @primary_key(audience_id);
    audience_id @index;
    /// produces PRIMARY KEY(scope_id, audience_id)
};
```

Named primary key:

```
table SessionIdentity {
    session_id @primary_key "session_identity_pkey";
};
```

@unique

```
table RefreshToken {
    refresh_token_base @primary_key;
    access_token @unique;
};
```

@check

CHECK constraint using the expression language. `_` is placeholder for column value:

```
table Example {
    age: sql"INTEGER" @check(_ >= 0);
    name: sql"TEXT" @check(char_length(_) > 0);
    code: sql"TEXT" @check "valid_code" (LEFT(_, 1) == "U" && char_length(_) == 33);
};
```

@default

Sets column default using the expression language:

```
table Client {
    created_at: sql"TIMESTAMPTZ" @default(NOW());
    active: sql"BOOLEAN" @default(true);
    expires_in: sql"INTEGER" @default(7200);
```

```
};
```

@initialize_as

Like `@default`, but for populating columns at creation time. Can reference other columns and use expression language casts:

```
table UserBuiltinConsent {
    scope_id: builtin_scope @initialize_as(scope_id::builtin_scope);
    public: bool @initialize_as(false);
};
```

@index

```
table UsernameHistory {
    user_id @index;
    user_id @index "custom_index_name";
};
```

Index options for specialized indexes:

```
table Example {
    embedding @index.opclass(vector_cosine_ops);
    embedding @index.using(ivfflat);
    embedding @index.unique;
    embedding @index.with("lists = 100");

    /// all options combined
    embedding @index.opclass(vector_cosine_ops).using(ivfflat).with("options = \"..settings..\"");
};
```

Named indexes merge columns into one index:

```
table Example {
    a: sql"INTEGER" @index "composite_idx";
    b: sql"INTEGER" @index "composite_idx";
    /// produces CREATE INDEX composite_idx ON example(a, b)
};
```

Table-Level Constraints

For composite constraints or referencing multiple columns:

```
table UserChallenge {
    user_id;
    challenge_kind;
    check_data: sql"BYTEA";
    search_key: sql"BYTEA"?;
    keybase_material_sort_key: sql"INTEGER"?;

    @primary_key(user_id, challenge_kind, check_data);
    @unique(challenge_kind, search_key);
    @unique(user_id, keybase_material_sort_key);
};
```

Named constraints:

```
table AuthorizationCodeAudience {
    authorization_code;
    audience_id;
```

```
    @primary_key "oauth_authorization_code_audiences_pk" (audience_id, authorization_code);
};
```

Table-level @check references columns by name (no _ placeholder):

```
table Example {
    lo: sql"INTEGER";
    hi: sql"INTEGER";
    @check "valid_range" (lo <= hi);
};
```

Table-level @index:

```
table Example {
    user_id: sql"TEXT";
    email: sql"TEXT";
    @index(user_id, email);
    @index.unique(user_id, email);
};
```

Constraint Merging

All primary keys merge into one. Checks with the same name merge with AND:

```
scalar test = sql"TEXT" @inline @check "testcheck" (_ != "1");

table Foo {
    test @check "testcheck" (_ != "2");
    /// produces CHECK (test <> '1' AND test <> '2')
};
```

Named @unique and @index also merge their columns.

@external

Prevents immigrant from generating CREATE/DROP TABLE. For tables managed outside immigrant:

```
table External {
    id: sql"INTEGER";
    @external;
};
```

@rls, @role, @policy

Row level security, privileges and policies are described on the Roles, Grants and RLS page.

Mixins

Reusable groups of columns, attributes, and foreign keys:

```
@mixin Timestamps {
    created_at;
    updated_at;
};

table User {
    @mixin Timestamps;
    user_id @primary_key;
};
```

```
table Post {
    @mixin Timestamps;
    id: sql"INTEGER" @primary_key;
};
```

Mixins can include other mixins.

Views

Views wrap a SQL query. Use {TableName} to reference tables and {TableName.column} to reference columns. Immigrant substitutes actual database names:

```
view ActiveUsers = sql"""
    SELECT * FROM {User}
    WHERE {User.active} = true
""";
```

Materialized views use `view.materialized`:

```
view.materialized UserStats = sql"""
    SELECT {User.user_id}, count(*) as post_count
    FROM {User}
    JOIN {Post} ON {Post.user_id} = {User.user_id}
    GROUP BY {User.user_id}
""";
```

When referenced tables or columns are renamed, immigrant automatically recreates the view with updated names.

Views may carry an attribute block before `=`, holding role grants and the `@security_definer` opt-out:

```
view ActiveUsers {
    @role Reader {
        select;
    };
} = sql"SELECT * FROM {User}";
```

Views are security invoker by default, see the Roles, Grants and RLS page.

Foreign Keys

Inline (column-level)

Append ~ after column definition to declare a foreign key:

```
table Post {
    /// target column resolved by name
    user_id ~ User;
    /// explicit target column
    user_id ~ User(id);
    /// with ON DELETE action
    user_id ~.cascade User;
    /// named foreign key
    user_id "fk_author" ~.cascade User;
};
```

Foreign keys combine with other column attributes:

```
table Profile {
    // Primary key that also references User with cascade delete
    user_id @primary_key ~.cascade User;
};

table ClientRedirectUrl {
    // Indexed foreign key
    client_id @index ~.cascade Client;
    redirect_url: sql"TEXT";
    @primary_key(client_id, redirect_url);
};
```

Standalone (table-level)

For composite foreign keys or when source/target columns differ:

```
table AccessTokenScope {
    access_token ~.cascade AccessToken;
    audience_id ~.cascade Audience;
    scope_id;

    @primary_key(access_token, audience_id, scope_id);
    (audience_id, scope_id) ~.cascade AudienceScope;
    (authorization_code, audience_id) "aud_fk" ~.cascade AuthorizationCodeAudience;
};
```

ON DELETE actions

Specified with `.action` after ~:

Action	SQL
.cascade	ON DELETE CASCADE
.restrict	ON DELETE RESTRICT
.set_null	ON DELETE SET NULL
.set_default	ON DELETE SET DEFAULT

`.noop`	`ON DELETE NO ACTION` (default if omitted)

SQL Expression Language

Used in `@check`, `@default`, and `@initialize_as`. This is not raw SQL. Immigrant has its own expression syntax that compiles to SQL.

Operators

Precedence	Operator	SQL equivalent
Highest	`-x`, `+x`, `!x`	`-`, `+`, `NOT`
	`*`, `/`, `%`	`*`, `/`, `%`
	`+`, `-`	`+`, `-`
	`<`, `>`, `<=`, `>=`	same
	`~~`	`LIKE`
	`==`, `!=`	`=`, `<>`
	`===`, `!==`	`IS NOT DISTINCT FROM`, `IS DISTINCT FROM`
	`&&`	`AND`
Lowest	`||`	`OR`

Other expressions

```
table Example {
    a: sql"INTEGER"
        /// placeholder for current value
        @check(_ > 0)
        /// null literal
        @check(_ != null)
        /// boolean literal
        @check(_ == true)
        /// string literal
        @check(_ != "forbidden")
        /// numeric literal
        @check(_ != 42)
        /// function call
        @check(char_length(_) > 0)
        /// cast
        @check(_ :: BIGINT > 0)
        /// conditional (CASE WHEN ... THEN ... ELSE ... END)
        @check(if _ > 10 then true else false)
    ;
};

/// field access on composites
struct example {
    a: sql"INTEGER";
    @check(_.a > 0);
};
```

PostgreSQL

Naming Conventions

By default, immigrant transforms code names for PostgreSQL:

- **Tables and views:** snake_case + pluralize. `UserProfile` becomes `user_profiles`, `User` becomes `users`.
- **Columns:** used as-is from schema.
- **Scalars/domains:** used as-is from schema.

Override with explicit database name:

```
table UserProfile "user_profiles" { ... };
```

Or disable transformation entirely with `#pgnc(as_is)`:

```
#pgnc(as_is)
table my_table { ... };
/// Database name is 'my_table', not 'my_tables'
```

Auto-generated Constraint Names

When no explicit name is given, immigrant generates names following PostgreSQL conventions:

Constraint	Pattern
Primary key	`{table}_pkey`
Unique	`{table}_{columns}_key`
Index (non-unique)	`{table}_{columns}_idx`
Index (unique)	`{table}_{columns}_key`
Check	`{table}_{columns}_check`

Names longer than 63 characters are truncated with a hash suffix.

Comments

`//` doc comments before items produce `COMMENT ON` statements:

```
// User accounts
table User {
    // Unique identifier
    user_id @primary_key;
};
```

Generates:

```
COMMENT ON TABLE users IS 'User accounts';
COMMENT ON COLUMN users.user_id IS 'Unique identifier';
```

`///` comments are ignored by immigrant entirely.

Type Mapping

Schema item	PostgreSQL object
Scalar (non-inline)	`CREATE DOMAIN`
Scalar (`@inline`)	No object — type expanded at point of use
Enum	`CREATE TYPE ... AS ENUM`
Struct	`CREATE TYPE ... AS` (composite type)
Table	`CREATE TABLE`
View	`CREATE VIEW`
Materialized view	`CREATE MATERIALIZED VIEW`

View Management

PostgreSQL does not support in-place modification of view definitions. When a view needs to change, immigrant DROPs and reCREATEs it.

A view needs rebuilding when any of its dependencies change. PostgreSQL expands `SELECT *` at view creation time, so renaming a column in a referenced table leaves the view producing stale column names. Immigrant detects this by fingerprinting each view's SQL, referenced column names/types, and referenced view definitions (recursively).

When rebuilding is needed:

1. Affected views are renamed to temporary names
2. Table/column changes are applied
3. Temporary views are dropped
4. Views are recreated with updated references

Views are topologically sorted to respect dependencies — a view referencing another view is dropped first and created last. Materialized views are handled the same way.

Migration Stages

Immigrant generates migrations in ordered stages to maintain referential integrity:

1. Drop old foreign keys, rename columns
2. Drop indexes, add/alter columns, add/update non-FK constraints, recreate indexes
3. Drop removed columns, update comments
4. Drop affected views (dependency order)
5. Create/recreate views (dependency order)
6. Create new foreign keys

Column type changes use `ALTER COLUMN SET DATA TYPE ... USING` with the `@initialize_as` expression when provided.

Enum variant additions use `ALTER TYPE ... ADD VALUE`. Removing a variant requires recreating the type (PostgreSQL limitation).

Roles, Grants and RLS

Immigrant manages PostgreSQL roles, table/view privileges and row level security as part of the schema, so they are diffed and migrated like any other object.

Roles

```
role User {
    default {
        select;
    };
};
```

This declares a role that immigrant owns: it emits `CREATE ROLE` when the role appears, `DROP ROLE` when it is removed, and `ALTER ROLE ... RENAME TO` when it is renamed.

Roles follow the same naming rules as other items -- the code name is snake_cased, an explicit database name may be given, and `#pgnc(as_is)` disables the transformation:

```
role ReadOnly "reporting_ro" {};
```

Roles are created before every other object and dropped after them, so grants and policies always have a role to refer to.

External roles

`@external` marks a role that exists outside of the schema (created by the cloud provider, an operator, or another application):

```
role Admin {
    @external;
    default {
        select;
        insert;
        update;
        delete;
    };
};
```

Immigrant never creates, drops or renames an external role, but it still manages the privileges granted to it.

Default permissions

The `default` block lists permissions granted on **every** table and view in the schema:

```
role User {
    default {
        select;
    };
};
```

Available permissions are `select`, `insert`, `update` and `delete`.

A role without a `default` block gets no privileges unless a table or view grants them explicitly.

Grants

`@role` inside a table or view body overrides the default for that role, on that object only:

```
table Aaa {
    user_id;

    @role User {
        insert;
    };
};
```

Here `User` gets `INSERT` on `aaas` and nothing else -- the `select` from the `default` block does not apply, because the object opts out of the default by naming the role.

An empty block revokes everything for that role on that object:

```
table Secrets {
    @role User {};
};
```

Views take the same block, in an attribute list placed before `=`:

```
view AaaView {
    @role User {
        select;
    };
} = sql"SELECT * FROM {Aaa}";
```

Changing permissions produces the minimal delta -- only the added permissions are granted and only the removed ones are revoked:

```
REVOKE INSERT ON aaas FROM "user";
GRANT SELECT ON aaas TO "user";
```

Privileges of `@external` tables are not managed.

Row level security

`@rls` enables row level security on a table, `@rls.owner` additionally forces it for the table owner:

```
table Aaa {
    user_id;
    @rls;
    @rls.owner;
};
```

Attribute	SQL
`@rls`	`ENABLE ROW LEVEL SECURITY`
`@rls.owner`	`FORCE ROW LEVEL SECURITY`

Removing an attribute emits `DISABLE ROW LEVEL SECURITY` / `NO FORCE ROW LEVEL SECURITY`.

Note that RLS without policies denies everything except to the table owner -- add `@rls` together with the policies that should allow access.

Policies

`@policy.for(Role) "optional-name" (expression)` declares a policy. As a column attribute, `_` is the placeholder for that column:

```
table Aaa {
    user_id @policy.for(User) "per-user" (_ == current_setting("app.current_user_id")::user_id);
    @rls;
};
```

```
CREATE POLICY "per-user" ON aaas FOR ALL TO "user"
    USING (user_id = (current_setting('app.current_user_id'))::user_id);
```

The check uses the same expression language as `@check` and `@default`, see the SQL Expression Language page.

A policy may also be written as a table attribute, in which case there is no `_` placeholder and columns are named directly:

```
table Post {
    author_id;
    published: sql"BOOLEAN";
    @policy.for(User) (author_id == current_setting("app.current_user_id")::user_id || published);
    @rls;
};
```

Policies always generate `FOR ALL ... USING (...)`.

When the database name is omitted, it is generated as `{table}_{columns}_{role}_policy`, truncated with a hash suffix if it exceeds the identifier limit.

Two policies are considered the same object when they have the same name, or when their role and rendered check are identical -- so renaming a policy emits `ALTER POLICY ... RENAME TO` instead of a drop/create pair.

Declaring a policy on a table without `@rls` is an error, as is referencing a role that is not declared in the schema.

Views and `security_invoker`

Views are created with `WITH (security_invoker = true)`, so RLS of the underlying tables is evaluated against the role executing the query rather than the view owner:

```
CREATE VIEW aaa_views WITH (security_invoker = true) AS SELECT * FROM aaas;
```

`@security_definer` opts a view out, restoring the PostgreSQL default:

```
view V {
    @security_definer;
} = sql"SELECT * FROM {A}";
```

Toggling the attribute emits `ALTER VIEW ... SET (security_invoker = ...)` without recreating the view.

Materialized views are always security definer -- PostgreSQL does not apply RLS to them.

Schema versions

`security_invoker` changes the meaning of an existing schema, so it is gated behind a schema version. Old migrations are replayed with the version they were written under, which keeps migration history reproducible.

Version	Behaviour
0	`"INTEGER"` accepted in place of `sql"INTEGER"`
1	Views are security definer, RLS of underlying tables is bypassed on `SELECT`
2	Views are security invoker unless `@security_definer` is set

The current schema is always parsed at the latest version (2). A schema file may declare the version it was written for, and immigrant errors out if it does not match the version this binary supports:

```
@!schema_version 2;
```

Migrations record the version in their schema section header, and `immigrant-migrate` stores it in the `schema_version` column of `__immigrant_migrations`:

```
## Schema diff (v2)
```

Upgrading is a normal migration -- an existing schema on v1 produces `ALTER VIEW ... SET (security_invoker = true)` for every view the first time a migration is generated with a v2-aware immigrant.

Full example

```
scalar user_id = sql"INTEGER";

role User {
    default {
        select;
    };
};

table Name {
    user_id;
};

table Aaa {
    user_id @policy.for(User) "per-user" (_ == current_setting("app.current_user_id")::user_id);

    @role User {
        insert;
    };
    @rls;
};
```

```
CREATE ROLE "user";
CREATE TABLE names (
    user_id user_id NOT NULL
);
GRANT SELECT ON names TO "user";
CREATE TABLE aaas (
    user_id user_id NOT NULL
);
ALTER TABLE aaas ENABLE ROW LEVEL SECURITY;
GRANT INSERT ON aaas TO "user";
CREATE POLICY "per-user" ON aaas FOR ALL TO "user"
    USING (user_id = (current_setting('app.current_user_id'))::user_id);
```

Extensions

```
extension vector;
extension uuid_ossp "uuid-ossp";
extension vchord { cascade; };
extension postgis { version "3.4.2"; };
```

Generated SQL

Change	SQL
extension added	`CREATE EXTENSION "name" [VERSION 'v'] [CASCADE];`
`version` changed from one value to another	`ALTER EXTENSION "name" UPDATE TO 'v';`
`version` added or removed	nothing; the pin only affects future `CREATE EXTENSION`
`cascade` changed	nothing; only affects `CREATE EXTENSION`
extension removed	`DROP EXTENSION "name";`
database name changed	error, PostgreSQL has no rename for extensions

Extensions are created before roles are granted and before any type, table or view, and dropped after all of them, in reverse declaration order. `DROP EXTENSION` is emitted without `CASCADE`: if a column or index still depends on the extension, PostgreSQL refuses, which is the safe default.

Attributes

version

Pins the extension version. Passed as `VERSION` on create; a change between two pinned versions becomes `ALTER EXTENSION ... UPDATE TO`.

cascade

Adds `CASCADE` to `CREATE EXTENSION`, installing extensions the declared one depends on (e.g. `vchord` requires `vector`). Prefer declaring dependencies explicitly when the schema also uses their types directly.

@external

Marks an extension that exists outside of the schema (preinstalled by the provider or managed by an operator). Immigrant never creates, updates or drops it, but the declaration documents the dependency:

```
extension vector { @external; };

scalar embedding = sql"VECTOR(1024)";
```

Modules

A schema can be split into several files. The root file is `migrations/db.schema`, every other file is reached through `import` statements.

```
// migrations/db.schema
import { User, user_id } from "./auth/users.schema";

table Post {
   author: user_id ~ User(id);
};
```

```
// migrations/auth/users.schema
scalar user_id = sql"INTEGER";

table User {
   id: user_id @primary_key;
};
```

Imports

Import paths are relative to the importing file and must start with `./` or `../`. Paths may point outside of the `migrations` directory, e.g. to a schema of another service.

`import "./other.schema";` imports a file without binding any names. The file is still managed, which is useful to include a file that nothing references.

Packages

Paths starting with `@` point into a package: `@logseq/db.schema` is the file `db.schema` of the package `@logseq`. Relative imports inside of a package stay inside of it, a path climbing out of the package root is an error.

Packages are virtual, they only exist in the files given to the parser (e.g. by `immigrant-web`), keyed by their full path:

```
{
   "@main/db.schema": "...",
   "@logseq/db.schema": "import { User } from \"@main/db.schema\"; ...",
}
```

The `immigrant` CLI reads schemas from disk and doesn't support packages.

Multiple roots

A project may have several root files, e.g. one per plugin. A file is managed if it is reachable from any root through normal imports, and all roots share a single migration history, so cross references between roots, including cycles, are handled in a single migration. Removing a root drops its items, unless they are still imported by another root.

Every file has its own scope. A name is visible in a file only if the file declares it or imports it, including names used implicitly, like the column type of `user_id;`. Every declaration is importable, but imported names are not re-exported: import a name from the file that declares it.

Two files may declare items with the same name, as long as their database names differ:

```
import { Audit as AuthAudit } from "./auth.schema";

table Audit "post_audits" {
   ...
};
```

Import cycles are allowed.

Renaming on import

`import { A as B }` makes `A` available under the local name `B`. The rename only affects the current file; generated SQL always uses the definition.

Aliases

`alias B = A;` gives a local name to anything visible in the file, including imported names. Aliases are file-local and can't be imported.

```
import { user_id } from "./auth.schema";
alias uid = user_id;

table Session {
   uid;
};
```

External imports

`import @external { ... }` makes names usable for references, but immigrant never creates, alters or drops the imported items. Use it for objects owned by another service or migration tool.

```
import @external { Invoice, invoice_id } from "../../billing/migrations/db.schema";

table Payment {
   invoice: invoice_id ~ Invoice(id);
};
```

Which files are managed is decided from the root files: a file is managed if it is reachable from a root through normal imports. Everything that is only reachable through `@external` imports is external. If `a.schema` imports `b.schema`, and `b.schema` imports `a.schema` as `@external`, then with root `a.schema` both files are managed, and with root `b.schema` only `b.schema` is.

External items are imported as stubs: only the imported names and what they need are kept.

- Tables keep their name, database name, annotations, columns, primary key and unique constraints. Foreign keys, indexes, checks, policies, grants and defaults are ignored.
- Scalars, enums and structs are kept whole. Attributes of an external scalar still propagate to columns that use it, e.g. an `@index` on an external scalar creates an index on your table.
- Mixins are kept whole, applying an external mixin copies its contents into your table.
- Roles keep their name; their `default` permissions are not applied to your tables.

When an external item is renamed by its owner, references to it (e.g. foreign keys) generate no SQL, as PostgreSQL follows the rename.

Switching a file between managed and external generates no SQL for its items: immigrant assumes the objects already exist as declared when adopting a file, and leaves them in place when releasing it. Items

which were never referenced while the file was external are an exception: adopting them creates them.

Rename hints

By default immigrant matches old and new items by code name, and then by database name. Changing both at once looks like a removed item and a new one. `@renamed` tells immigrant what an item was called before:

```
table Account {
   email: sql"TEXT";
   @renamed mail to email;
};
@renamed User to Account;

enum Status {
   active;
   inactive;
   @renamed disabled to inactive;
};
```

Hint	SQL
table	`ALTER TABLE users RENAME TO accounts;`
column	`ALTER TABLE accounts RENAME COLUMN mail TO email;`
enum variant	`ALTER TYPE "Status" RENAME VALUE 'disabled' TO 'inactive';`
struct field	`ALTER TYPE "Point" RENAME ATTRIBUTE x TO lat;`

The new name is resolved like any reference, the old name is taken from the current file. This allows moving an item to another file with a new name, by leaving the hint in the old file:

```
// migrations/db.schema, which used to declare Account
import { Customer } from "./crm.schema";
@renamed Account to Customer;
```

If the old file is deleted, put the hint in any managed file and point to the old file with `from`:

```
@renamed Customer from "./crm.schema" to Client;
```

A hint is only used if the old schema has the old name, and doesn't have the new one (unless the new name is itself renamed away, so swaps work). Hints which were already applied are ignored, so they can be removed after the migration is committed.

Migrations

Every file reachable from the root is stored in migrations, keyed by its path relative to the `migrations` directory. External files are stored as slices: only the stub items and the imports they use are kept, so unrelated changes in external files don't produce migrations.

````
## Schema diff (v2)

### ../billing/db.schema (new)
```immigrant
scalar invoice_id = sql"INTEGER";
...
```

### auth/users.schema
````

```
@@ -2,4 +2,5 @@
...

### old.schema (removed)
```

When the list of roots differs from the previous migration, it is stored in a `## Roots` section, placed before the schema section:

```
## Roots
- @logseq/db.schema
- @main/db.schema
```

Projects with a single `db.schema` root file keep the original migration format.

Directories in `migrations` that don't start with a digit are not treated as migrations, so schema files can be organized into subdirectories.

Migrations

Non-destructive migrations

Both `immigrant commit-unchecked` and `immigrant-migrate commit` accept `--non-destructive`, which controls what happens with data, when a table or a column is removed from the schema. Indexes, constraints and other objects which don't hold data are dropped as usual.

Mode	Behavior
`fail`	Generating the migration fails, if it would drop a table or a column.
`keep`	Dropped tables and columns are renamed to `obsolete_<id>_<name>`, where `<id>` is the id of the new migration.
any other value	Same as `keep`, but the value is used in place of the migration id.

Kept objects are detached from the rest of the schema, so they don't prevent future migrations:

- foreign keys, constraints, indexes, policies and grants of kept tables are dropped;
- kept columns lose `NOT NULL` and defaults;
- columns of enum and struct types are converted to `TEXT`, columns of domain types to the underlying type.

The id used for kept objects is stored in the migration, so `immigrant regenerate-sql` produces the same SQL:

```
## Non-destructive
keep 12
```

Obsolete objects are not tracked by immigrant, drop them manually once the data is no longer needed.

02

Integrations

Diesel DRAFT 31

DMDL DRAFT 32

Diesel

DRAFT

Immigrant can generate Diesel schema definitions and derive macros for your Rust projects.

Setup

TODO

Annotations

Use `#diesel(...)` annotations to control code generation:

```
#diesel(type="dt::Text",copy,custom)
scalar user_id = sql"TEXT";

#diesel(derive = "async_graphql::Enum")
enum client_kind { confidential; public; };
```

Generated Code

TODO

DMDL

DRAFT

DMDL (Data Model Definition Language) integration allows immigrant schemas to be exported in DMDL format for documentation and visualization tools.

Overview

TODO

Usage

TODO