☕ ChaiSQL: the user manual.

Basic primer for users and developers.

This page is intentionally left blank.

Running DB schema

Here is the description of the running database schema used for this study.

1. Two entity tables

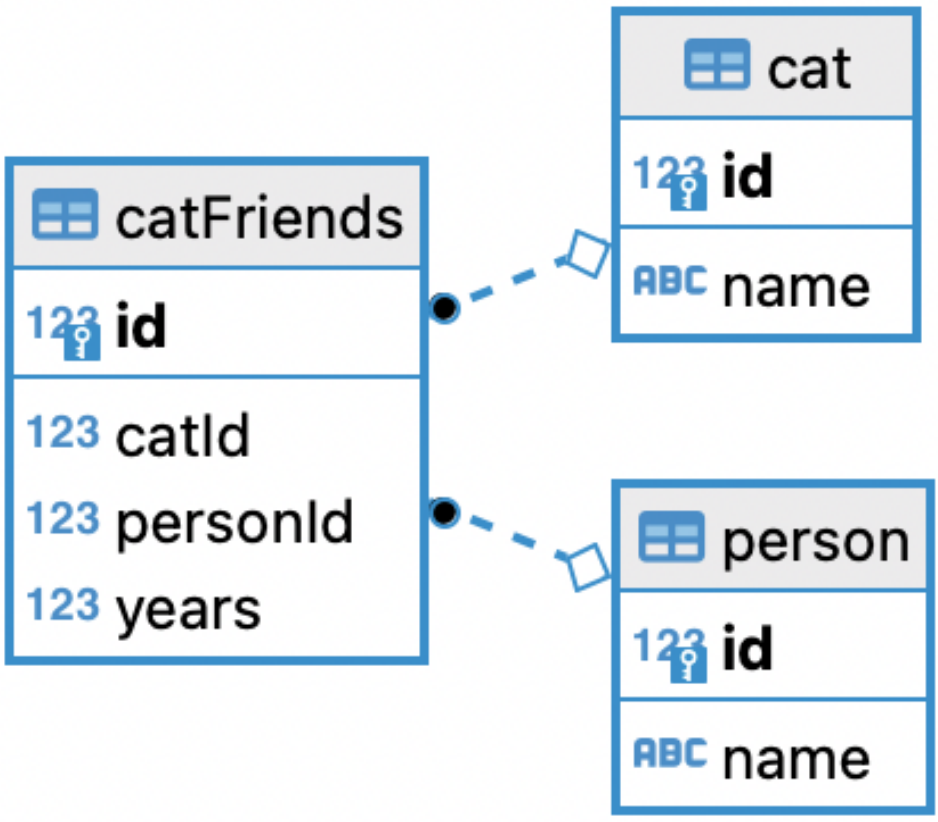
   `cat` and `person`

2. One intersection table

   `catFriends`

This database schema is purposefully simplistic to focus on illustrating the ChaiSQL concepts, but therefore, may demonstrate some impractical solutions.

~~~


~~~

What is ChaiSQL

☕ **ChaiSQL** is a **static type checker** for **SQL SELECT queries**.

In general, it is a tool, allowing to validate that the expected, or *hinted*, query result types correspond to its real behavior. It works only for SELECT queries at this point, but future versions may expand its capacities.

In more detail, the *static* side makes sure that the validation is executed before the query is submitted to any RDBMS. The *type checker* indicates the principle of how ChaiSQL works - by comparing the programmer-specified type hints with the internally computed types for the query. If these match, the type checking is considered successful. Otherwise, the query is considered not well typed.

It is important to notice, that not being well-typed, does not mean that the query can not be executed by a RDBMS, what it helps with instead, is knowing that the well-typed queries can be executed by a RDBMS with 100% type certainty.

ChaiSQL interfaces: files & CLI

ChaiSQL exposes two modes of user interaction: a comment-based type annotation syntax, and a command line (CLI) program.

The comment-based type annotations allow to specify type hints without modifying any existing query syntax, allowing, therefore, to execute the query in a RDBMS directly.

The CLI program, allows to validate the type hints specified in a standalone file containing the SQL queries.

ChaiSQL informal syntax

Below is the general structure of ChaiSQL annotations.

~~~

```
(-- @chaisql:<meta commands>)*

(-- @chaisql:<annotation command>)?

<SQL Query>;
```
~~~

ChaiSQL example: high-level overview

This is a simple example of using ChaiSQL annotations.

~~~

```
-- @chaisql:check
-- @chaisql:newtype Name = String
-- @chaisql:returns DbView <bag> {id: Number, name: Name}
SELECT id, name
FROM person;
```

ChaiSQL example discussed

~~~

>> This command instructs the ChaiSQL engine to perform the checking:

```
-- @chaisql:check
```

>> This command creates a type alias Name, equating it to a String type:

```
-- @chaisql:newtype Name = String
```

>> This command specifies the expected type of the query result:

```
-- @chaisql:returns DbView <bag> {id: Number, name: Name}
```

>> Since we select, id, and name from the person table, we know the types:

```
SELECT id, name
```

>> The person table is the same as discussed in "Running DB schema":

```
FROM person;
```

ChaiSQL built-in types: Primitive, Compound: views, notations

ChaiSQL generalizes the popular types available in different SQL engines. Below is the list of the included concepts so far:

~~~ Primitive types

```
-- ... String
```

> The `String` *type represents textual values. This captures popular SQL engine datatypes like* `CHAR(size)`, `VARCHAR(size)`, `TEXT(size)` *and so forth.*

```
-- ... Number
```

> The `Number` *type represents numeric values. This captures popular SQL engine datatypes like* `INT(size)`, `INTEGER(size)`, `FLOAT(size, d)` *and so forth.*

```
-- ... Boolean
```

> The `Boolean` *type represents logical values. This captures popular SQL engine datatypes like* `BOOL`, and `BOOLEAN(size)`, but also extends to represent the types of the condition statements in the `WHERE` clauses, acting in this case as the three-valued logic of SQL.

~~~ Compound types

```
-- ... DbView <[notation: bag|set]> {[key]: [type], ...}
```

> The `DbView <[notation: bag|set]> {[key]: [type]}` *type represents the type of the returned SQL results.*

> The first type parameter - `<[notation: bag|set]`*, specifies whether the result collection is in the bag or set notation. For instance, all results from a query* `SELECT DISTINCT ...`*, would be interpreted with* `set` *notation, whereas* `SELECT ...`*, without a* `DISTINCT` *part, will be treated as a* `bag`*. In that sense, a* `<set>` *is a special case of the* `<bag>` *notation with all duplicates removed.*

> *The second parameter* {[key]: [type], ...} *specifies the column structure of the returned results. In that sense,* [key] *corresponds to the name of the returned column, and* [type] *can be one of the Primitive types. For example, a query* SELECT id FROM person, will return the result of type: DbView <bag> {id: Number}.

In the future versions, the users of ChaiSQL will be able to provide schema definitions with custom types, extending the built-in types.

ChaiSQL commands: Trigger, Alias, Type Hint

The ChaiSQL syntax has only three predefined commands, acting as *validation trigger, type alias declaration,* and the *query result type hint*.

ChaiSQL trigger

To inform the ChaiSQL that the SQL queries provided in the file need to be checked, it requires an explicit command, before any other type-related information is provided. The command is specified as follows:

~~~

```
-- @chaisql:check
```

> *This informs the ChaiSQL process to check the provided SQL file.*

ChaiSQL type alias

Motivated by the need to, e.g., reuse a complex type in many places, or give a more descriptive identity to the underlying type, it may be convenient to declare a custom alias for a primitive or compound type. This is done as follows:

~~~

```
-- @chaisql:newtype [Alias] = [Primitive or compound type]
```

> *This informs the ChaiSQL to associate a new type* [Alias] *with* [Primitive or compound type]. *A number of restrictions apply, requiring unique*

[Alias] *declarations - same* [Alias] *cannot be declared more than once. And that* [Primitive or compound type] *may reuse other aliases, but should resolve to a primitive or compound type without any aliases. Furthermore,* [Alias] *should consist of alphanumeric values, including '_' and '-'.*

> To illustrate this, consider the example:

```
-- @chaisql:newtype Key = Number
```

> this declares a new type alias Key.

```
-- @chaisql:newtype IdView = DbView <bag> {id: Key}
```

> this declares a new type alias IdView, linked to a compound type, reusing the Key type alias.

ChaiSQL type hint

This command specifies the expected type of the result of an SQL SELECT query. It can be used to annotate both nested and top-level queries. ChaiSQL will use these type hints to check the type-safety of the provided query. It should be done as follows:

~~~

```
-- @chaisql:returns [Primitive, compound, or alias type]
<SQL SELECT to annotate>
```

> This command specifies a type hint for the SQL SELECT below. It must always appear directly above the SQL query that it annotates. The [Primitive, compound, or alias type] part may contain any well-written type expression, be it a primitive type, a compound, or a type alias.

> Consider the following example:

```
-- @chaisql:returns DbView <bag> {name: String}
SELECT name FROM person;
```
~~~

ChaiSQL syntax details

Enabling the type checker

Before we submit the query to the ChaiSQL type checker, we need to enable it in comments with the following command:

~~~

```
-- @chaisql:check
```

Handling basic SELECT queries

ChaiSQL is able to check the specified *expected* types against the *inferred* types.

The expected types are provided by the SQL programmer, in the form of a comment. This specification acts like an assertion of the resulting type. ChaiSQL, upon receiving a check request, evaluates the query, computing all the types internally and checking whether the specified expectations satisfy the computed values.

Generally, we distinguish four types of SELECT queries: *asterisk*, *direct*, *aliased,* and *fully-qualified*.

Asterisk SELECT

The asterisk, SELECT *, selects *all* columns from the "*from*" clause of the query.

This should be represented in the returned *DbView* type construct, by specifying all columns and their types in the *body* of the view.

~~~

```
-- @chaisql:check

-- @chaisql:returns DbView <bag> {id: Number, name: String}

SELECT *

FROM person;
```

Direct column access in SELECT

The SELECT queries with direct access to the column names are the bread and butter of any SQL programmer. This type of queries, specifies precisely *which columns* are projected from the "*from*" clause of the query.

This should be accordingly reflected in the type expectations specified for ChaiSQL.

~~~

```
-- @chaisql:check

-- @chaisql:returns DbView <bag> {name: String}

SELECT name

FROM person;
```

Aliased column access in SELECT

This select variation allows the programmer to override the name of the selected column with custom aliases.

This can (and should) be accordingly represented in the ChaiSQL type annotations.

~~~

```
-- @chaisql:check

-- @chaisql:returns DbView <bag> {personName: String}

SELECT name AS personName

FROM person;
```

Fully-qualified column access in SELECT

Besides specifying aliases, SQL allows the fully-qualified column access, where the column is preceded by the table name. In such cases, only the column name needs to be specified in the type hint.

~~~

```
-- @chaisql:check

-- @chaisql:returns DbView <bag> {name: String}

SELECT p.name

FROM person AS p;
```

Set/bag notations elaborated

ChaiSQL's compound type `DbView <[notation: bag|set]> {[key]: [type], ...}` expects the *notation*, either `<set>` or `<bag>`, as its first type argument.

This information is added to make the result type of the SELECT queries more detailed. In particular, it illustrates the use of SQL's native *bag* or *set* semantics of the query results.

In short, *bag*s allow duplicates in the result, meanwhile *sets* ensure that all duplicates are removed. This becomes relevant, when we attempt to distinguish between SQL's `SELECT ...` and `SELECT DISTINCT ...`, or the SQL's query operators `UNION/INTERSECT/EXCEPT ...` and `UNION/INTERSECT/EXCEPT ALL ...`.
~~~

SELECT without DISTINCT

Every regular SELECT query results in a DbView <bag> {[key]: [type], ...}.

~~~

```
-- @chaisql:check
-- @chaisql:returns DbView <bag> {name: String}
SELECT name
FROM person;
```

>> In this case, the result of the query may contain duplicate names.

SELECT with DISTINCT

Every regular SELECT query, results in a DbView <bag> {[key]: [type], ...}.

~~~

```
-- @chaisql:check
-- @chaisql:returns DbView <set> {name: String}
SELECT DISTINCT name
FROM person;
```

>> In this case, the result of the query does not contain any duplicate names.

UNION/INTERSECT/EXCEPT without ALL

Note: although here we consider UNION as the main example for the coming two cases, the same principle applies if we replace it with INTERSECT or EXCEPT SQL query operators.

By the semantics of UNION/INTERSECT/EXCEPT (accepted by most RDBMS), unless the SQL author specifies ALL modifiers to this operator, the result of the query expression follows *set* semantics.

Following the SQL language specification (accepted by most RDBMS), to use this operator, both the left hand side and the right hand side query must have equal number and order of the returned columns, with compatible data types. This, therefore, should also be captured at the type hint level.

Let's illustrate this:

~~~

```
-- @chaisql:check
-- @chaisql:returns DbView <set> {name: String}
(
    -- @chaisql:returns DbView <bag> {name: String}
    SELECT name FROM person
    >> Here we select all names from the person table.
)
UNION
(
    -- @chaisql:returns DbView <bag> {name: String}
    SELECT name FROM cat
    >> Here we select all names from the cat table.
);
>> In the end, we select the set of all names from the person, or cat table.
```
~~~

UNION/INTERSECT/EXCEPT with ALL

In contrast to the previous example, where we use the SQL query operators without the ALL modifier, here we explicitly mention it, to allow duplicate results.

Consider the following example:

~~~

```
-- @chaisql:check
-- @chaisql:returns DbView <bag> {name: String}
(
    -- @chaisql:returns DbView <bag> {name: String}
    SELECT name FROM person
    >> Here we select all names from the person table.
)
UNION ALL
(
    -- @chaisql:returns DbView <bag> {name: String}
    SELECT name FROM cat
    >> Here we select all names from the cat table.
);
>> In the end, we select the bag of all names from the person, or cat table.
>> These names may be duplicates, without being filtered out.
```
~~~

Annotating sub-queries

In the previous example, we have already seen that type hints may appear inside of the sub-query brackets. The type hints are always scoped to the nearest query below them. Thus, this technique can be used in any cases, where a subquery is relevant and a type hint is beneficial.

Let us consider the following example:

~~~

```
-- @chaisql:check

-- @chaisql:returns DbView <bag> {name: String}

SELECT name

FROM person

WHERE id IN (

    -- @chaisql:returns DbView <set> {personId: Number}

    SELECT DISTINCT personId

    FROM catFriends

);
```

>> Here, we provide a type hint on the nested query, capturing that the condition `id IN (...)` checks whether an `id` of a `person` record appears in the selected set of `personId`'s from the `catFriends` table.

>> As both of these columns are of type `Number`, the condition is well-typed.

Note: future versions of ChaiSQL may extend the type hint syntax, allowing to specify annotations on the column access, from, and condition levels.
~~~

Putting all things together

To provide a final example, we illustrate a large query featuring all the elements discussed above, including the *type aliases*, *bag and set* notations and the *type hints for sub-queries*.

~~~

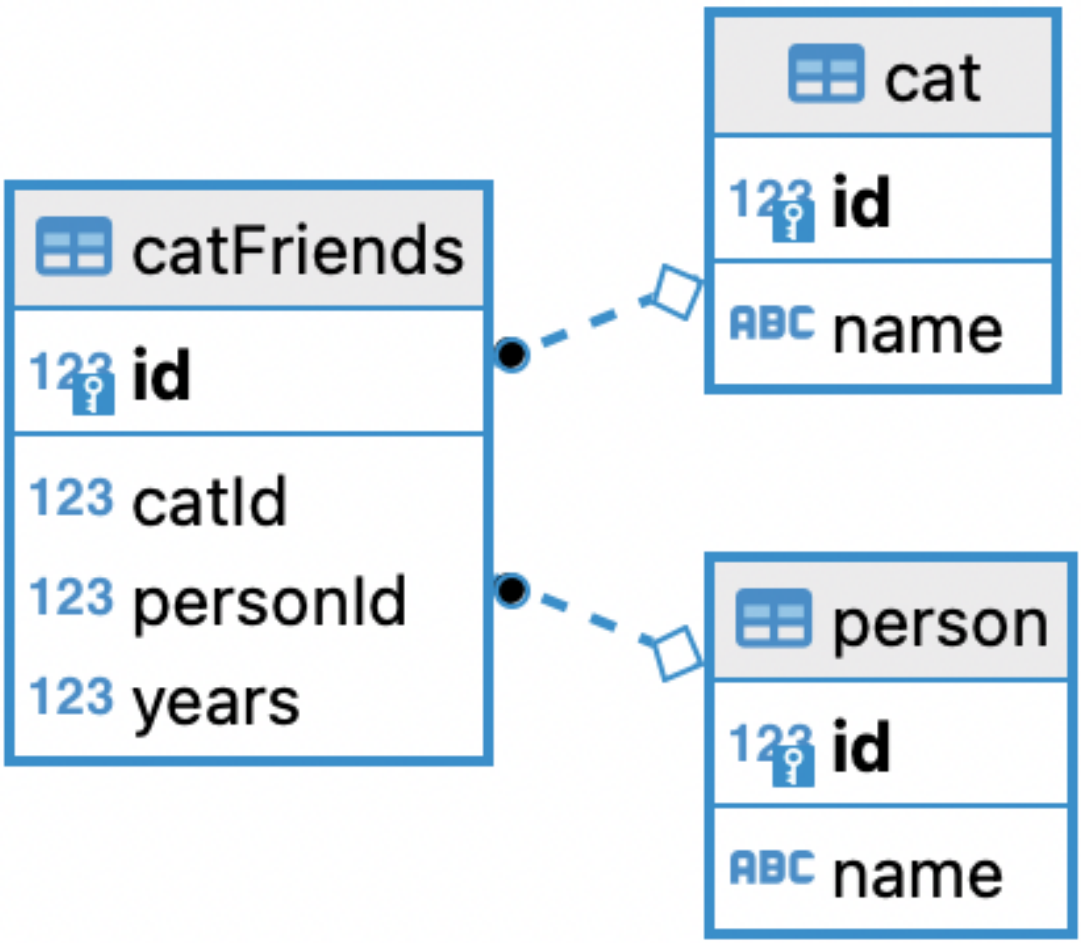
```
-- @chaisql:check
>> Instructing ChaiSQL to check this file.
-- @chaisql:newtype Key = Number
-- @chaisql:newtype Name = String
>> Providing custom aliases for reuse.
-- @chaisql:returns
      DbView <bag> {id: Key, catName: Name, personName: Name}
SELECT      cf.id, c.name AS catName, p.name AS personName
> This select does not remove any duplicates.
FROM        catFriends AS cf, cat AS c, person AS p
WHERE       cf.catId = c.id
AND         cf.personId = p.id
AND         p.name IN (
      -- @chaisql:returns DbView <set> {name: Name}
      SELECT DISTINCT name FROM cat
      >> This select does remove duplicate names from the cat table.
);
```
~~~

>> This query, therefore, returns a joint result of the `catFriends` and the two entity tables, meanwhile checking only for cases where the name of a `person` appears among the `cat` names.

It is worth keeping in mind that the query is purely artificial and only serves to illustrate the discussed concepts. Given its sub-optimal nature, there exist alternative ways of implementing such a request.

Running DB schema (zoomed)

~~~


~~~