CQL: A CSV Query Language

User Manual

Ross Aiden Melo

Contents

1 Introduction

CQL is a domain-specific language for querying CSV files using SQL-inspired syntax. It allows users to load one or more CSV tables, filter rows, project columns, sort results, and combine query results via set operations. The interpreter is written in Haskell using Alex and Happy, and emphasises robust error handling and ease of use for real-world CSV data.

2 General Query Structure

All CQL queries follow this high-level pattern:

```
FROM <TableList>
[WHERE <Condition>]
SELECT <ExprList>
[ORDER BY <ExprList>]
```

Optional clauses are in brackets. You may also combine full queries with `UNION`, `INTERSECT` or `EXCEPT`.

3 Syntax Overview

Each clause in CQL serves a distinct role. Below we show the syntax and explain its purpose.

3.1 FROM Clause

```
FROM A
FROM A, B
```

Specifies one or more input tables (CSV files named `A.csv`, `B.csv`, etc.). Multiple tables trigger a Cartesian product, enabling implicit joins. You may alias a table with `AS`, for example:

```
FROM Orders AS o, Customers AS c
```

3.2 SELECT Clause

```
SELECT A.1, A.2
SELECT A.3 AS label
SELECT A.1 + A.2 AS combined
```

Defines which columns or expressions appear in the output. Use `AS` to alias fields. Expressions can mix columns, literals and functions, implementing projection.

3.3 WHERE Clause

```
WHERE A.2 == "apple"
WHERE NOT (A.1 == A.2)
WHERE EXISTS(A.3)
```

Filters rows according to a boolean condition. Supports comparisons (`==, !=, <, <=, >, >=`), logical operators (`AND, OR, NOT`) and functions such as `EXISTS`.

3.4 ORDER BY Clause

```
ORDER BY A.2 ASC, A.1 DESC
ORDER BY COALESCE(A.2, "zzz")
```

Specifies *multikey sorting* of the result set: rows are ordered firstly by the first expression, then by the second if the first keys tie, and so on. Each key may be `ASC` (ascending) or `DESC` (descending). Without this clause, results default to lexicographic order.

3.5 Set Operations

```
<Query1> UNION <Query2>
<Query1> INTERSECT <Query2>
<Query1> EXCEPT <Query2>
```

Combines two complete queries as sets of rows:

- **UNION** returns every row that appears in either query.
- **INTERSECT** returns only those rows that appear in both queries.
- **EXCEPT** returns rows that appear in the first query but not in the second.

4 Built-in Functions and Expressions

- **Column references:** `A.1`, `B.3`
- **Literals:** string in quotes, integer (treated as string)
- **EMPTY:** denotes the empty string
- **COALESCE(e1,e2):** returns the first non-empty of `e1` and `e2`; useful for supplying a default when data is missing
- **EXISTS(e):** true if `e` yields a non-empty string
- **Concatenation (+):** appends two strings
- **Comparison:** `==`, `!=`, `<`, `<=`, `>`, `>=`
- **Boolean:** `AND`, `OR`, `NOT`

5 Extension Features

Below are the key extensions beyond the basic clauses.

5.1 Aliasing

Assign names to tables or expressions via `AS`.

```
FROM A AS a, B AS b
SELECT a.1 AS id, b.2 AS name
```

5.2 Set Operators

Combine queries via relational set algebra:

- `UNION` merges two query results, removing duplicates.
- `INTERSECT` yields only rows present in both results.
- `EXCEPT` yields rows in the first result that are not in the second.

5.3 ORDER BY (Multikey Sorting)

```
ORDER BY A.2 ASC, A.1 DESC
```

Rows are first sorted by `A.2`, then any ties are sorted by `A.1`.

5.4 COALESCE

Provides a fallback for empty fields:

```
SELECT COALESCE(A.3, "none") AS label
```

5.5 EXISTS

Filters by presence of data:

```
WHERE EXISTS(A.4)
```

5.6 Input Padding

Short rows are padded with `""` so that references to missing columns yield empty strings rather than errors.

5.7 Comment Support

Single-line comments begin with `-`.

```
-- This is a comment
FROM A -- load table A
SELECT A.1, A.2 -- project two columns
```

5.8 VS Code Syntax Highlighting

A TextMate grammar provides:

- Keyword colouring (`FROM`, `SELECT`, etc.)
- String literal and number highlighting
- Comment styling for `-`

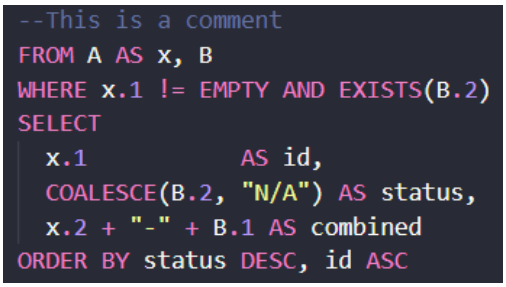

Figure 1: Syntax highlighting of CQL code in VS Code

6 Error Handling

CQL intercepts and reports all errors cleanly—no stack traces.

6.1 File I/O Errors

```
Error: Cannot open query file 't1.cql'. Please check the filename.
Error: Cannot open CSV file './A.csv'.
```

6.2 CSV Arity Errors

```
Error: Inconsistent number of columns in './B.csv': row lengths = [2,1]
```

6.3 Parse and Lex Errors

```
Error: Parse error at line 3, column 1: unexpected token
Error: lexical error at line 1, column 5: unrecognised input '#'
```

6.4 Runtime Expression Errors

```
Error: Column not found: A.5
Error: Expression is not boolean: Literal "foo"
```

7 Design Rationale

CQL's design balances power, clarity and usability:

SQL-inspired syntax The familiar `SELECT-FROM-WHERE` form lowers the learning barrier; users with SQL experience can quickly use CQL.

Declarative style Users specify *what* data they want—filters, projections and joins—without prescribing *how* to navigate or iterate. This enables the interpreter to optimise execution, and makes queries easier to read, maintain and extend.

Relational completeness By supporting selection, projection, join (Cartesian product + filter), union, intersection and difference, CQL can express any query in classical relational algebra, ensuring full expressive power.

All-string data model Treating every field as a string avoids complex type systems and coercion rules. Empty fields are simply `""`, making parsing and evaluation straightforward.

Positional column references 1-based indexing (`A.1`, `A.2`) removes the need for explicit schema declarations, simplifying grammar and usage.

Robust error handling All failure modes—file I/O, CSV arity mismatches, lexical/parsing errors and runtime lookup failures—are caught and reported with clear, user-friendly messages.

8 BNF Grammar

```
<Query> ::= <SingleQuery>
              | <Query> UNION <Query>
              | <Query> INTERSECT <Query>
              | <Query> EXCEPT <Query>

<SingleQuery> ::= <FromClause> <WhereClause>? <SelectClause> <OrderByClause>?
<FromClause> ::= FROM <SourceList>
<SourceList> ::= <Source> ("," <Source>)*
<Source> ::= ident (AS ident)?

<WhereClause> ::= WHERE <Expr>

<SelectClause> ::= SELECT <SelectItemList>
<SelectItemList>::= <SelectItem> ("," <SelectItem>)*
<SelectItem> ::= <Expr> (AS ident)?

<OrderByClause> ::= ORDER BY <OrderItemList>
<OrderItemList> ::= <OrderItem> ("," <OrderItem>)*
<OrderItem> ::= <Expr> (ASC | DESC)?

<Expr> ::= <Term>
              | <Expr> "==" <Expr>
              | <Expr> "!=" <Expr>
              | <Expr> "<" <Expr>
              | <Expr> "<=" <Expr>
              | <Expr> ">" <Expr>
              | <Expr> ">=" <Expr>
              | <Expr> "+" <Expr>
              | <Expr> "-" <Expr>
              | <Expr> "*" <Expr>
              | <Expr> "/" <Expr>
              | <Expr> AND <Expr>
              | <Expr> OR <Expr>
              | NOT <Expr>
              | EXISTS "(" <Expr> ")"
              | COALESCE "(" <Expr> "," <Expr> ")"
```

```
<Term> ::= ident
                | ident "." ident
                | ident "." int
                | string
                | int
                | EMPTY
                | "(" <Expr> ")"
```

References

[1] Codd, E. F. (1970) 'A Relational Model of Data for Large Shared Data Banks', *Communications of the ACM*, 13(6), pp. 377–387. DOI: 10.1145/362384.362685.

[2] International Organization for Standardization and International Electrotechnical Commission (2023) *ISO/IEC 9075-1:2023 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*. Geneva: ISO. Available at: `https://www.iso.org/standard/76583.html` (Accessed: 8 April 2025).