

Tabular Database Systems

⑥

The Structured Query Language (SQL)

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 | The Origins of SQL



Don Chamberlin



Ray Boyce (†1974)

Don Chamberlin and **Ray Boyce**, co-inventors of SQL, back in 1972/73 both members of IBM's research project *System R*.

The Origins of SQL

- Development of the language started in 1972, first as **SQUARE**, from 1973 on as **SEQUEL** (*Structured English Query Language*). In 1977, SEQUEL became **SQL** because of a trademark dispute. (Thus, both “S-Q-L” /,ɛskjuːˈɛl/ and “*sequel*” /ˈsiːkwəl/ are okay pronunciations.)
- First commercial implementations in the late 1970s/early 1980s. By 1986, the ANSI/ISO standardization process begins.
- Since then, SQL has been in active development and remains the “*Intergalactic Dataspeak*”¹ to this day.

Current SQL standard (as of April 2025): SQL:2023.

¹ Due to Mike Stonebraker, inventor of Ingres (1972, precursor of Postgres, PostgreSQL)

2 | SQL: Row Variables


Like most regular programming languages, SQL features a construct to **bind variables to values**.

- `FROM` is the *only* SQL clause that can introduce variables:

```

⋮
FROM t AS v -- bind variable v to the rows in table t
⋮

```


- If $|t| = m$, `v` iterates over all m rows of `t` in some order. `v` is thus known as **row variable** (sometimes: table alias .
- If table `t` has schema `t(c1 τ1, ..., cn τn)`, then `v` has type `row(c1 τ1, ..., cn τn)`.²
- Can access column `ci` of `v` using **dot notation**: `v.ci` (`:: τi`).

² In DuckDB, `row(c1 τ1, ..., cn τn)` is a synonym for type `struct(c1 τ1, ..., cn τn)`: think of a table row like a record/struct with n fields.

FROM generates Cross Products

- If a SQL query needs to read data from **multiple tables** t_1, \dots, t_m , list all tables in the **FROM** clause:

```
⋮  
FROM  $t_1$  AS  $v_1$ , ...,  $t_m$  AS  $v_m$  -- row variable names  $v_i$  unique  
⋮
```

- **Cross product** semantics: This generates **all** $|t_1| \times \dots \times |t_m|$ **combinations** of bindings for row variables v_i in some order.
- Row variable names v_i are unique, tables t_j may repeat (**FROM** t **AS** v_1 , t **AS** v_2 allows to combine each row of t with its peers).
-  Yes, **FROM** clauses may generate *MANY* bindings. Typical queries will use predicates on the v_i to only focus on the combinations that make sense.

3 | SQL: Joining Tables

Cross products between tables are (too?) general: it is common to draw rows from tables based on *conditions* that identify (un)wanted row combinations. SQL supports several such **table joins**.

- **Inner Join:**³

```












⋮
FROM  $t_1$  AS  $v_1$  [INNER] JOIN  $t_2$  AS  $v_2$  ON ( $p$ )
⋮

```



- Draw all combinations of rows from tables t_1 , t_2 .
- Only keep those bindings of v_1 , v_2 that satisfy **predicate p** (v_1 , v_2 typically do occur free in p).

³ Inner join is often referred to as simply *join* or *θ -join* (initially, DB literature used the fancy greek theta θ instead of p to denote the join predicate).

Inner Join Follows Foreign Keys

vehicles						peeps			
vid	vehicle	kind	seats	wheels?	pid	pid	pic	name	born
v ₁		car	5	true	p ₄	p ₁		Cleo	2013
v ₂		SUV	3	true	p ₄	p ₂		Bert	1968
v ₃		bus	42	true	□	p ₃		Drew	□
v ₄		bus	7	true	□	p ₄		Alex	2002
v ₅		bike	1	true	p ₂				
v ₆		tank	□	false	p ₃				
v ₇		cabrio	2	true	p ₄				

```
SELECT v.vehicle, p.name AS driver, p.pic
FROM vehicles AS v INNER JOIN peeps AS p ON (v.pid = p.pid)
                        "equi-join" ↗
```

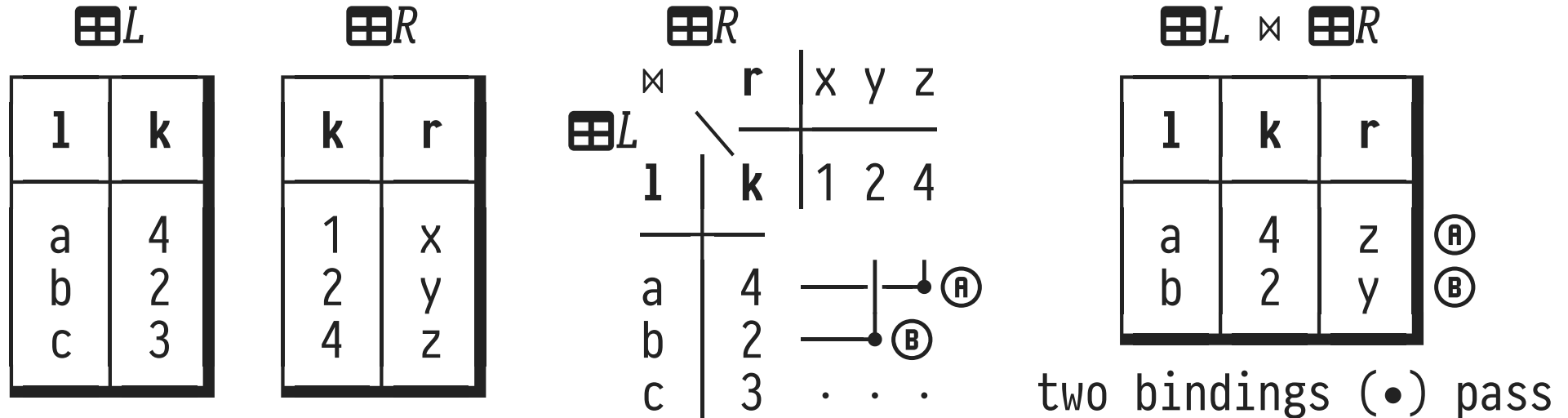
- Some rows in table `peeps` find multiple *join partners* in `vehicles` (like p₄ ) , some find none (p₁ ).
- General: A join between `t1` and `t2` may yield $0 \dots |t_1| \times |t_2|$ rows.

More Join Operations ①

Inner joins (in particular: equi-joins) are frequent. Daily SQL practice calls for a whole **family of join variants**, however.

To introduce the join family, let us use a sketch of their semantics (see below). We assume join predicate $p \equiv L.k = R.k$.

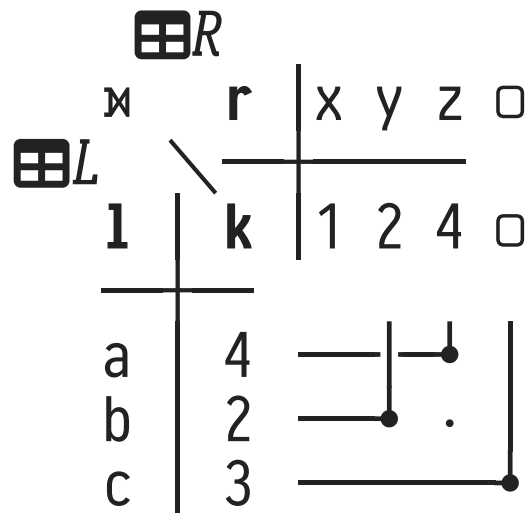
- **INNER JOIN** (\bowtie , `FROM L INNER JOIN R ON (L.k = R.k)`):



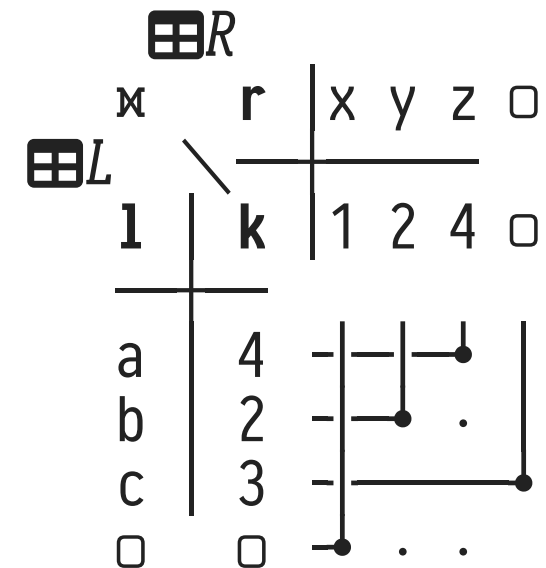
More Join Operations ②: Outer Joins

- LEFT/RIGHT/FULL OUTER JOIN (\bowtie , \bowtie , \bowtie)

LEFT OUTER JOIN



FULL OUTER JOIN



- Left outer join: *All rows* of the left table $\bowtie L$ are kept. Rows from $\bowtie L$ that find no join partner (like $(c,3)$) are paired with an artificial all-NULL (\square) row.

More Join Operations ③: Semi/Anti/Cross Joins

SEMI/ANTI/CROSS JOIN (\ltimes , $\bar{\ltimes}$, \times)

SEMI JOIN


		\ltimes	$\bar{\ltimes}$	\times
		\ltimes	$\bar{\ltimes}$	\times
\ltimes	$\bar{\ltimes}$	\times	$\bar{\ltimes}$	\times
\ltimes	$\bar{\ltimes}$	\times	$\bar{\ltimes}$	\times
1	k	1	2	2
a	1	•	.	.
b	2	—	—	•
c	3	.	.	.

ANTI JOIN

		\ltimes	$\bar{\ltimes}$	\times
		\ltimes	$\bar{\ltimes}$	\times
\ltimes	$\bar{\ltimes}$	\times	$\bar{\ltimes}$	\times
\ltimes	$\bar{\ltimes}$	\times	$\bar{\ltimes}$	\times
1	k	1	2	4
a	4	.	.	o
b	2	.	o	.
c	3	—	—	•

CROSS JOIN

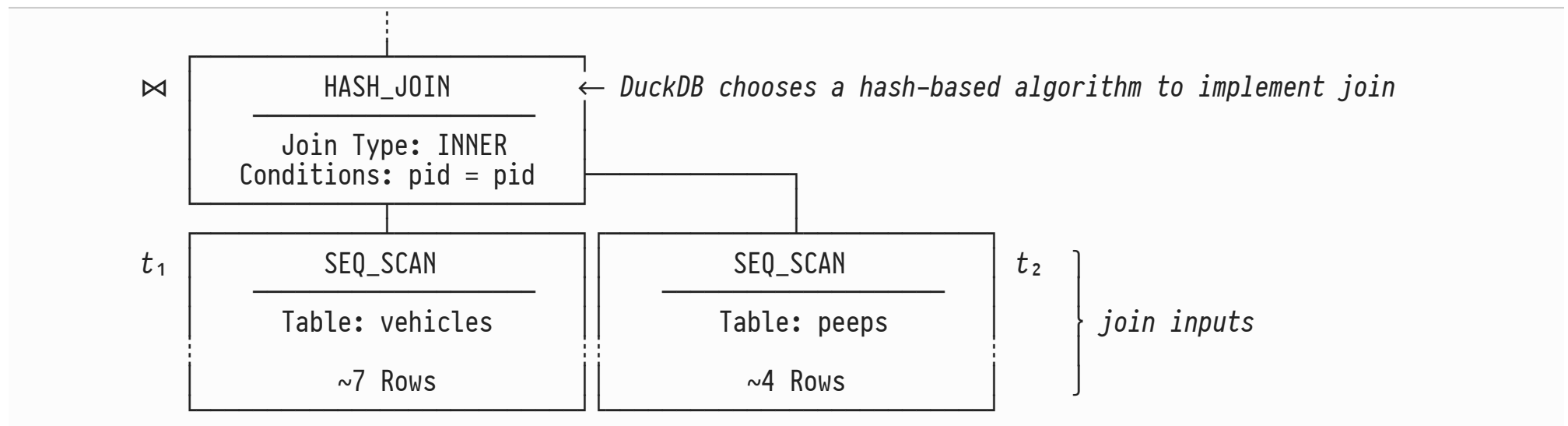
		\ltimes	$\bar{\ltimes}$	\times
		\ltimes	$\bar{\ltimes}$	\times
\ltimes	$\bar{\ltimes}$	\times	$\bar{\ltimes}$	\times
\ltimes	$\bar{\ltimes}$	\times	$\bar{\ltimes}$	\times
1	k	1	2	4
a	4	•	•	•
b	2	•	•	•
c	3	•	•	•

- Semi join: Keep rows from \ltimes with *at least one join partner*.
- Anti join: Keep rows from \ltimes that have *no join partner*.
- Semi/anti join return a subset of rows from \ltimes :  #032
join predicate p aside, row variables bound to \ltimes are inaccessible in the SQL query (thus above: \rightarrow instead of \rightarrow).

Joins in Query Plans

Joins—in textbooks often: \bowtie (or $\bowtie \bowtie \bowtie \bowtie \bar{\bowtie} \bowtie \bowtie \dots$)—operate on two tables. A join between n tables requires $n-1$ binary \bowtie operations).

- **FROM vehicles NATURAL JOIN peeps** in **EXPLAIN** output (excerpt):



- Inner join is *commutative* ($t_1 \bowtie t_2 \equiv t_2 \bowtie t_1$) and *associative* ($(t_1 \bowtie t_2) \bowtie t_3 \equiv t_1 \bowtie (t_2 \bowtie t_3)$).

Exploited by the DBMS's **query optimizer** to rearrange joins and inputs to find an efficient query plan alternative.

4 | SQL: Bag Algebra

A table instance is a **bag** of rows (no order, duplicate rows may occur). SQL features the binary operations of the bag algebra:

[Likewise: **INTERSECT ALL** \cap_+ , **EXCEPT ALL** \setminus_+]


bag of rows

$\underbrace{query_1 \text{ UNION ALL } query_2}_{\text{two bags of rows}} \quad \text{-- } query_1 \uplus query_2$

two bags of rows

- Rows returned by *query*₁, *query*₂ (and the result) conform:⁴
 - Number of columns are equal.
 - Types in the same column position match (or are castable).
 - *query*₁ determines the column names of the result.

 #034

⁴ Alternatively, DuckDB can match columns in a bag union operation by name (not position): **UNION ALL BY NAME**. This is non-standard. See the [DuckDB documentation on bag \(and set\) operations](#) .

SQL's Bag Algebra Respects Row Multiplicities

Operations of the bag algebra respect *multiplicities*. So does SQL:

$$\begin{array}{l}
 \{\text{A A A B C}\} \cup \{\text{A B B D}\} = \{\text{A A A A B B B C D}\} \quad (\text{add}) \\
 \{\text{A A A B C}\} \cap_+ \{\text{A B B D}\} = \{\text{A B}\} \quad (\text{minimum}) \\
 \{\text{A A A B C}\} \setminus_+ \{\text{A B B D}\} = \{\text{A A C}\} \quad (\text{subtract, } \geq 0)
 \end{array}$$

A B C D denote rows, $\{\dots\}$ denotes a bag (table) of rows.

- To ignore multiplicities, omit the **ALL** modifier. Bag operations then discard row duplicates ($\delta(\{\text{A A A B C}\}) \stackrel{\text{def}}{=} \{\text{A B C}\}$):

$$\text{query}_1 \text{ OP query}_2 \equiv \delta(\delta(\text{query}_1) \text{ OP ALL } \delta(\text{query}_2))$$

- Duplicate elimination** (δ) can be generally useful in queries. SQL thus supports the **DISTINCT** modifier in the **SELECT** clause:

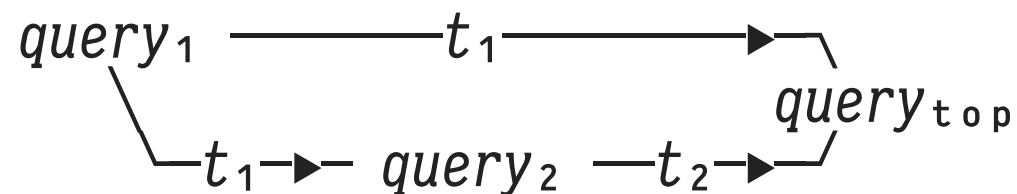
$$\begin{array}{l}
 \text{SELECT DISTINCT } \dots \\
 \text{FROM } \dots \\
 \vdots
 \end{array}
 \equiv
 \delta \left(\begin{array}{l}
 \text{SELECT } \dots \\
 \text{FROM } \dots \\
 \vdots
 \end{array} \right)$$

5 : SQL: Assembling Queries From Pieces (Common Table Expressions)

If a complex SQL query is best understood in terms of *intermediate result tables*, name these intermediates and refer to them later:


WITH	Common Table Expression (CTE)
t_1 AS ($query_1$),	• locally define name t_i as result of $query_i$
$t_2(c_1, \dots, c_n)$ AS ($query_2$)	• may name result columns of intermediates
$query_{top}$	• top-level query (= result of CTE)


- In such a CTE, the dependencies between queries and visibility (or: scopes) of the t_i form a DAG. No cycles:



- Decision: *materialize* t_i or *inline* $query_i$ at its use site(s)?

6 : A *Star Wars* Database

kaggle  is an extensive online repository of open data sources, many of those in CSV and/or Parquet formats.

- The “*Star Wars Dataverse*” collects bits of information on Disney's *Star Wars* franchise in 15 Parquet files.
- We created a native  database file '*035-starwars.db*' from these Parquet sources and cleaned the data:
 - Normalized in-universe dates⁵, map to type *int* (*'3 BBY'* → *-3*, *'2 ABY'* → *2*).
 - Turned comma-separated *text* values into *text[]* arrays (*'Luke, Leia'* → [*'Luke'*, *'Leia'*]).
 - Normalized film titles across tables. (*'Episode IV: A New Hope'* → *'A New Hope'*)

Load the *Star Wars* database to exercise your SQL skills.

 #035

⁵ BBY/ABY: Years *Before/After* the *Battle of Yavin* in which the Rebel Alliance  destroyed the Death Star .

DuckDB UI: Browsing a Database

Load the *Star Wars* database and use the browser-based **DuckDB UI**⁶ to get familiar with the schemata/instances of the 15 tables:

1. Install and load the DuckDB UI extension (first use only):

```
D INSTALL motherduck;  
D LOAD motherduck;
```

2. Load *Star Wars* database, invoke the UI (starts browser):

```
D ATTACH '035-starwars.db' AS starwars;  
D CALL start_ui();
```

```
UI started at http://localhost:4213/
```

⁶ Once the UI extension is installed, may start DuckDB UI from the shell via command `duckdb -ui <database>.db`.

SQL Training (Star Wars Database)

Use SQL to formulate these queries against the *Star Wars* database. If needed, use CTEs (**WITH ...**) to cope with query complexity.

1. Which characters have got “*a bad feeling about this*” in what film(s)? [★]
2. Which films (title, year) make up the prequels (released after *Return of the Jedi*, before *The Force Awakens*)? [★★]
3. Which films (title) were created by the congenial duo of director George Lucas and composer John Williams? [★★]
4. Which droids (name) appear in *any* film directed by George Lucas?
5. Which droids (name) appear in *all* films directed by George Lucas? [★★]
[★★★★]