Tabular
Database Systems

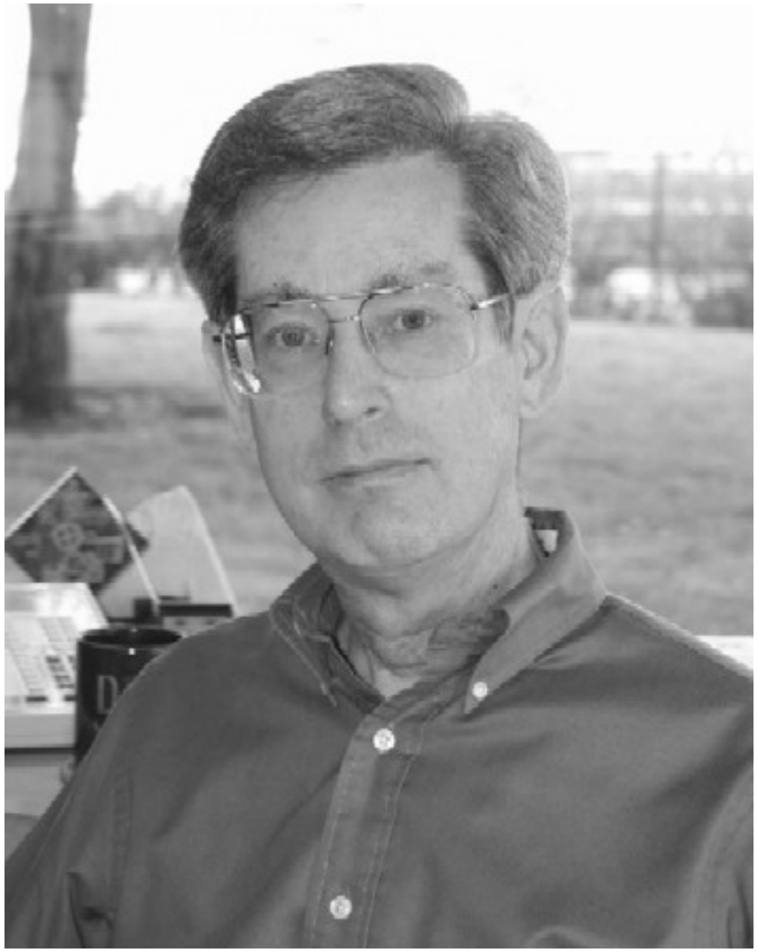
⑥

The Structured Query Language (SQL)

Summer 2025

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*" /ˌɛskjuːˈɛl/ and "*sequel*" /ˈsiːkwəl/ are okay pronounciations.)

- 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 $t(c_1\ \tau_1, \ldots, c_n\ \tau_n)$, then v has type $\mathrm{row}(c_1\ \tau_1, \ldots, c_n\ \tau_n)$.[2]
  - Can access column c_i of v using **dot notation:** $v.c_i$ (:: τ_i).

[2] In DuckDB, $\mathrm{row}(c_1\ \tau_1, \ldots, c_n\ \tau_n)$ is a synonym for type $\mathrm{struct}(c_1\ \tau_1, \ldots, c_n\ \tau_n)$: think of a table row like a record/struct with n fields.

#028

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 t₁ AS v₁, …, tₘ AS vₘ -- row variable names vᵢ unique
  ⋮
```

  - **Cross product** semantics: This generates **all** $|t_1| \times \cdots \times |t_m|$ **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.

#029

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 t₁ AS v₁ [INNER] JOIN t₂ AS v₂ 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 referrered 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

vid	vehicle	kind	seats	wheels?	pid
v_1		car	5	true	p_4
v_2		SUV	3	true	p_4
v_3		bus	42	true	□
v_4		bus	7	true	□
v_5		bike	1	true	p_2
v_6		tank	□	false	p_3
v_7		cabrio	2	true	p_4

peeps

pid	pic	name	born
p_1		Cleo	2013
p_2		Bert	1968
p_3		Drew	□
p_4		Alex	2002

```sql
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 p_4), some find none (p_1).
- General: A join between t_1 and t_2 may yield $0 \ldots |t_1| \times |t_2|$ rows.

More Join Operations ➊

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 $p \equiv L.k = R.k$.

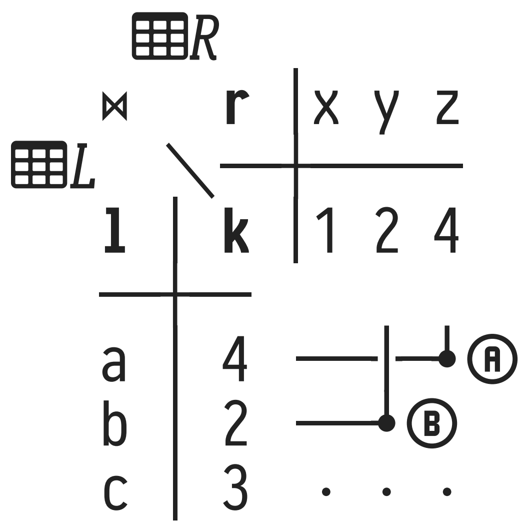
- **INNER JOIN** (⋈, `FROM L INNER JOIN R ON (L.k = R.k)`):

L

l	k
a	4
b	2
c	3

R

k	r
1	x
2	y
4	z



L ⋈ *R*

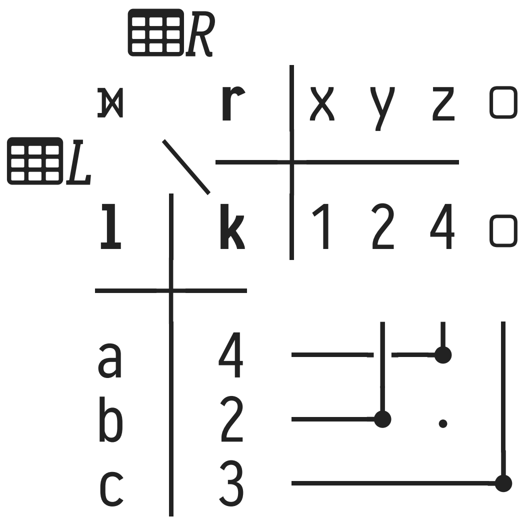
l	k	r	
a	4	z	Ⓐ
b	2	y	Ⓑ

two bindings (•) pass

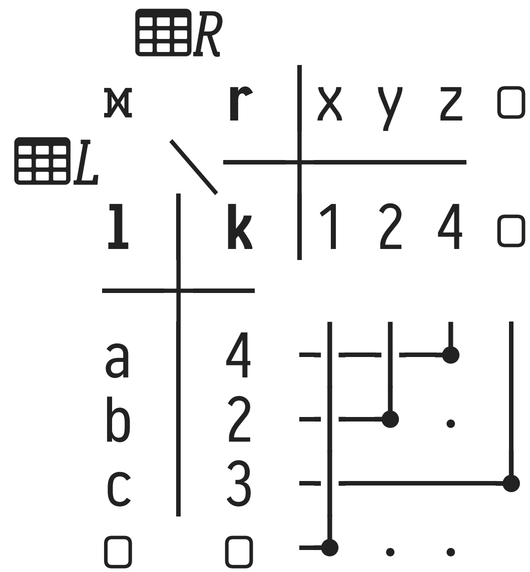
More Join Operations ❷: Outer Joins

- **LEFT/RIGHT/FULL OUTER JOIN** (⟕, ⟖, ⟗)

LEFT OUTER JOIN



FULL OUTER JOIN

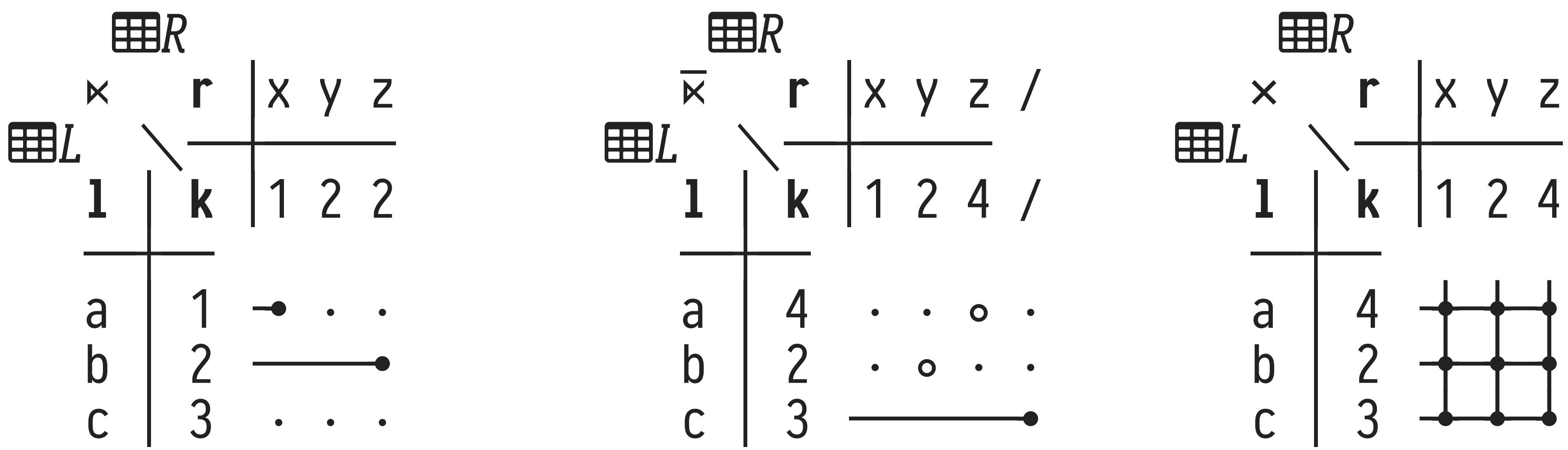


  - Left outer join: *All rows* of the left table L are kept. Rows from L that find no join partner (like (c,3)) are paired with an artificial all-NULL (□) row. #031

More Join Operations ❸: Semi/Anti/Cross Joins

- **SEMI/ANTI/CROSS JOIN** ($\ltimes$, $\overline{\ltimes}$, $\times$)

SEMI JOIN **ANTI JOIN** **CROSS JOIN**



- ○ Semi join: Keep rows from ▦L with *at least one join partner*.
- ○ Anti join: Keep rows from ▦L that have *no join partner*.

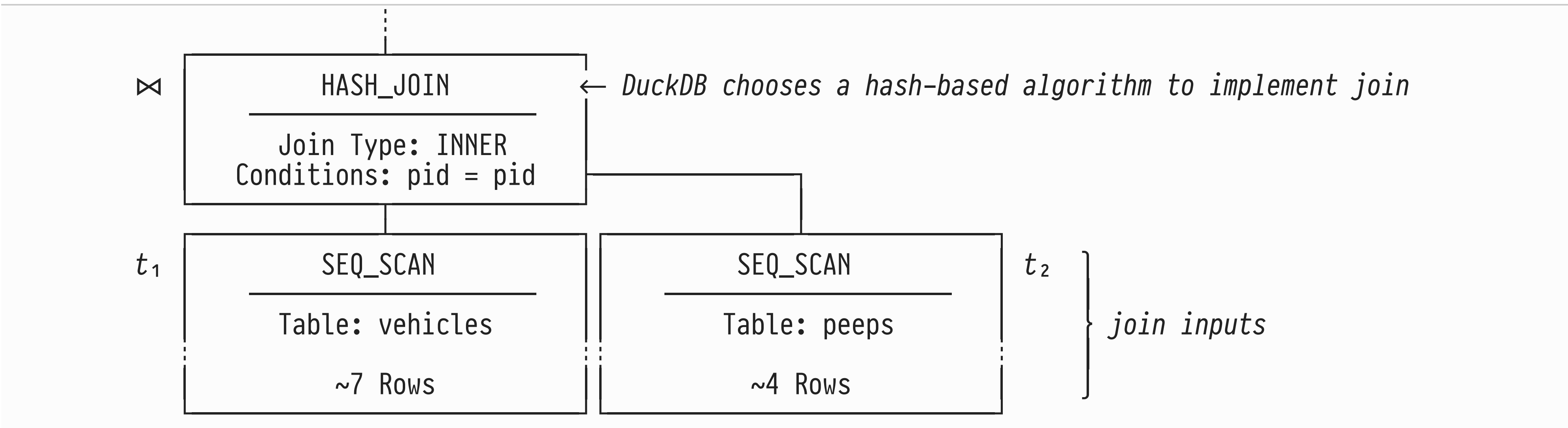
- ○ Semi/anti join return a subset of rows from ▦L: join predicate p aside, row variables bound to ▦R are inaccessible in the SQL query (thus above: ─• instead of ─┴•).

#032

Joins in Query Plans

Joins—in textbooks often: ⋈ (or ⟕ ⟖ ⟗ ⋉ ⋉̄ ⧓ ⧔…)—operate on two tables. A join between n tables requires n−1 binary ⋈ 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. #033

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** $\setminus_+$]

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

- 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:

⦃🅐🅐🅐🅑🅒⦄ ⊎ ⦃🅐🅑🅑🅓⦄ = ⦃🅐🅐🅐🅐🅑🅑🅑🅒🅓⦄ (add)
⦃🅐🅐🅐🅑🅒⦄ ∩₊ ⦃🅐🅑🅑🅓⦄ = ⦃🅐🅑⦄ (minimum)
⦃🅐🅐🅐🅑🅒⦄ \₊ ⦃🅐🅑🅑🅓⦄ = ⦃🅐🅐🅒⦄ (subtract, ⩾ 0)

🅐🅑🅒🅓 denote rows, ⦃…⦄ denotes a bag (table) of rows.

- To ignore multiplicities, omit the **ALL** modifier. Bag operations then discard row duplicates (δ(⦃🅐🅐🅐🅑🅒⦄) ≝ ⦃🅐🅑🅒⦄):

$$\mathit{query}_1\ \mathbf{OP}\ \mathit{query}_2 \quad \equiv \quad \delta(\ \delta(\mathit{query}_1)\ \mathbf{OP\ ALL}\ \delta(\mathit{query}_2)\ )$$

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

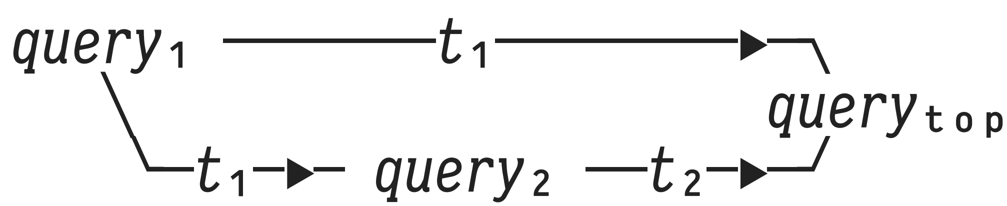
```
SELECT DISTINCT ⋯                  ⎛ SELECT ⋯ ⎞
FROM    ⋯               ≡     δ    ⎜ FROM   ⋯ ⎟
  ⋮                                ⎝   ⋮      ⎠
```

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** (	• locally define name t_i as result of $query_i$
$query_1$),	
$t_2(c_1,\dots,c_n)$ **AS** (	• may name result columns of intermediates
$query_2$)	
$query_{top}$	• 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)?

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 '*-starwars.db' from these Parquet sources and cleaned the data:
  - Normalized in-universe dates[5], 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? [★★★★]