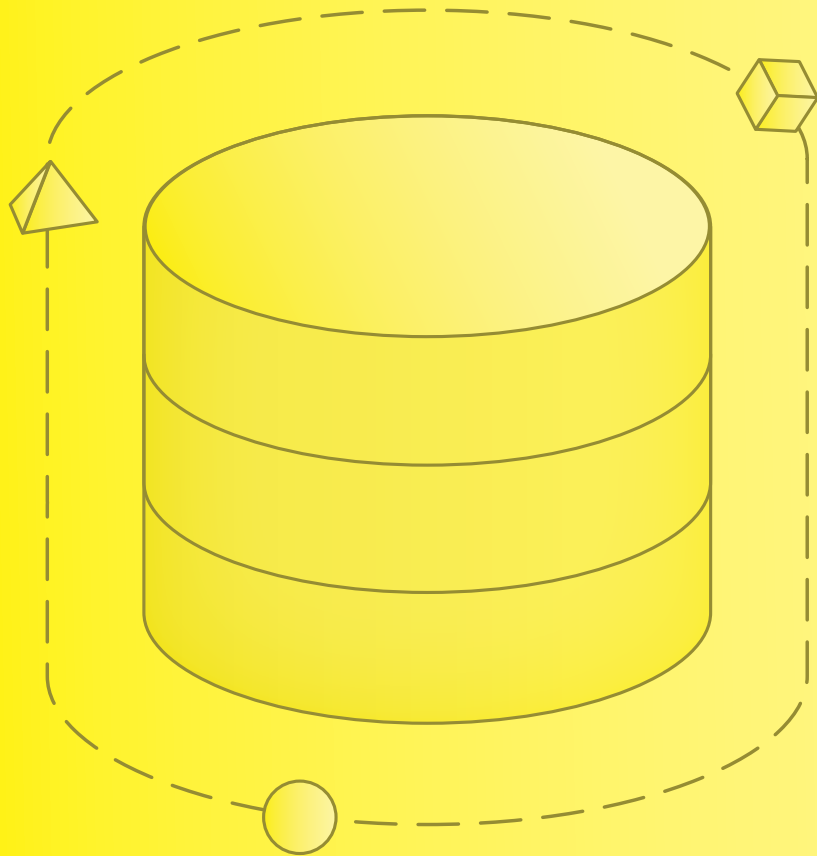
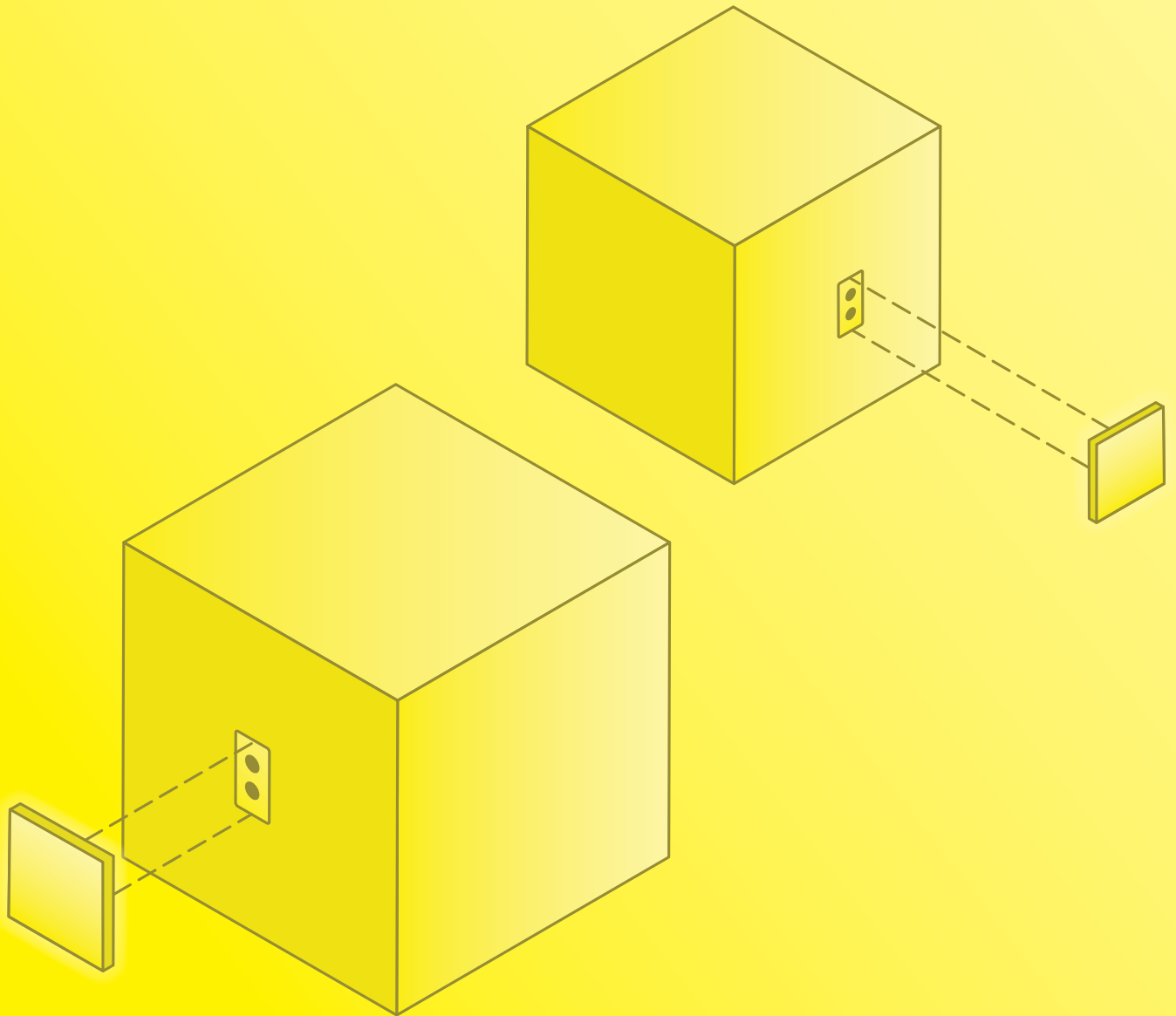


2026

FRIENDLY SQL





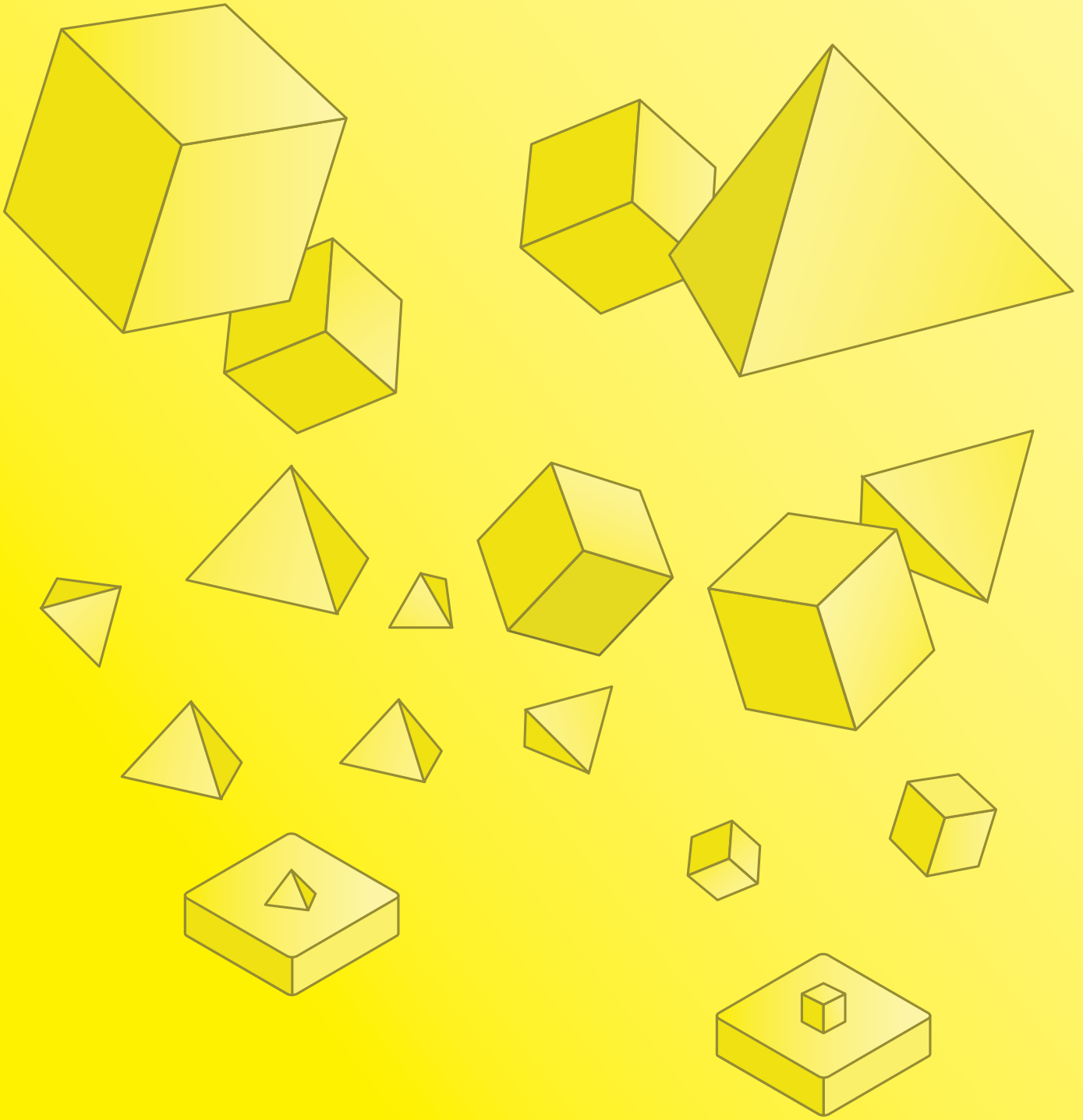
JANUARY

PREFIX ALIASES

Prefix aliases allow you to formulate queries by first specifying the alias followed by the expression that defines it. This syntactic sugar improves query readability and works in the SELECT, JOIN, and FROM clauses.

```
-- Return each train station's distance
-- from Schiphol Airport station in miles
SELECT
    station: s2.name_long,
    distance_miles: d.distance / 1.61
FROM d: 'distances.csv'
JOIN s1: 'stations.csv' ON s1.code = d.station1
JOIN s2: 'stations.csv' ON s2.code = d.station2
WHERE s1.name_long = 'Schiphol Airport';
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | | | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 | |



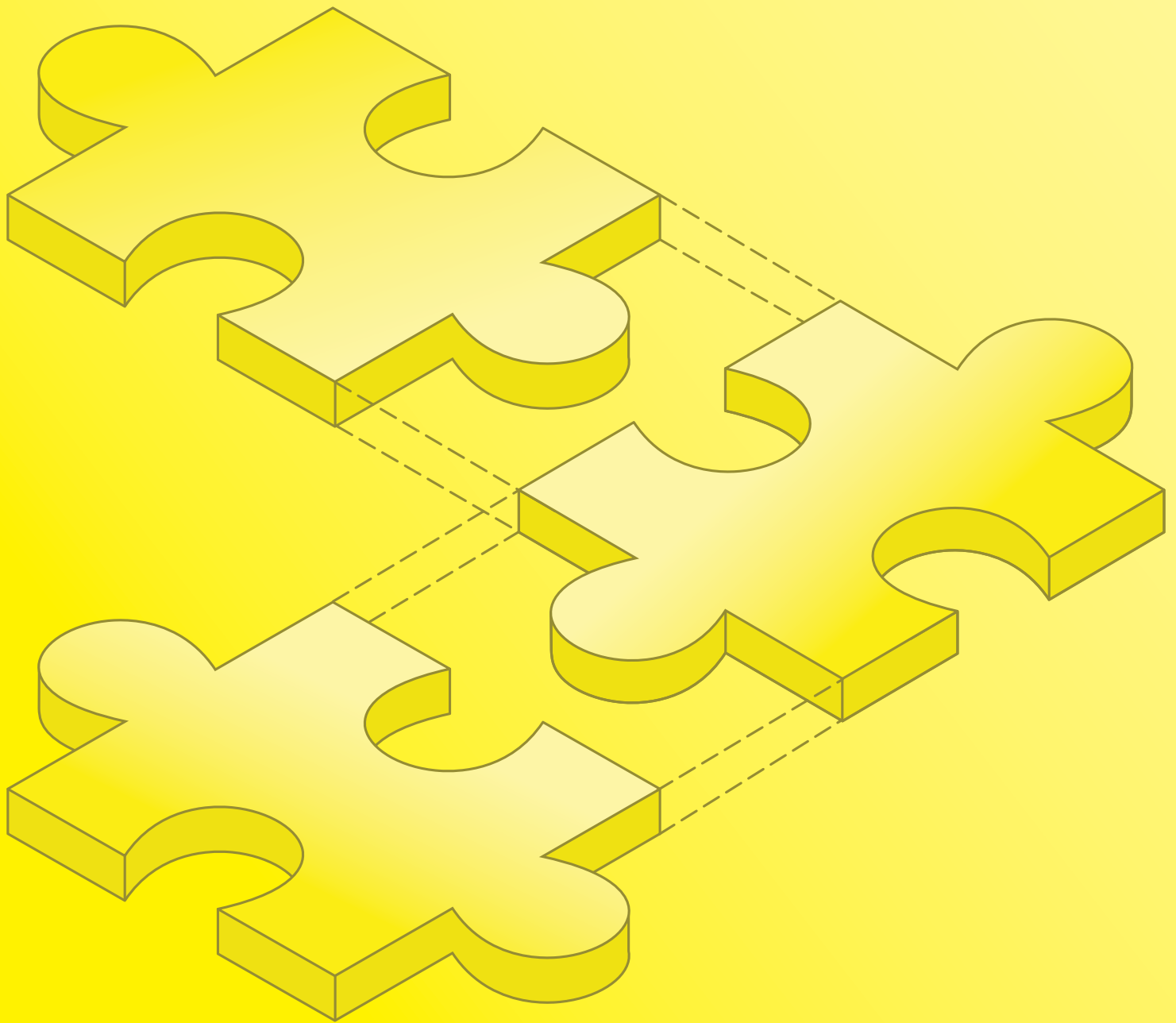
FEBRUARY

GROUP BY ALL

GROUP BY ALL, one of DuckDB's most popular "friendly SQL" features, enables you to formulate aggregation queries without manually specifying all columns that are not part of an aggregate expression. This syntax has been adopted by several other database systems and is under consideration for inclusion in the next SQL standard.

```
-- For each station, return the average and
-- the maximum distance of other stations
SELECT station1, avg(distance), max(distance)
FROM 'distances.csv'
GROUP BY ALL;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | | | | | | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | |



MARCH

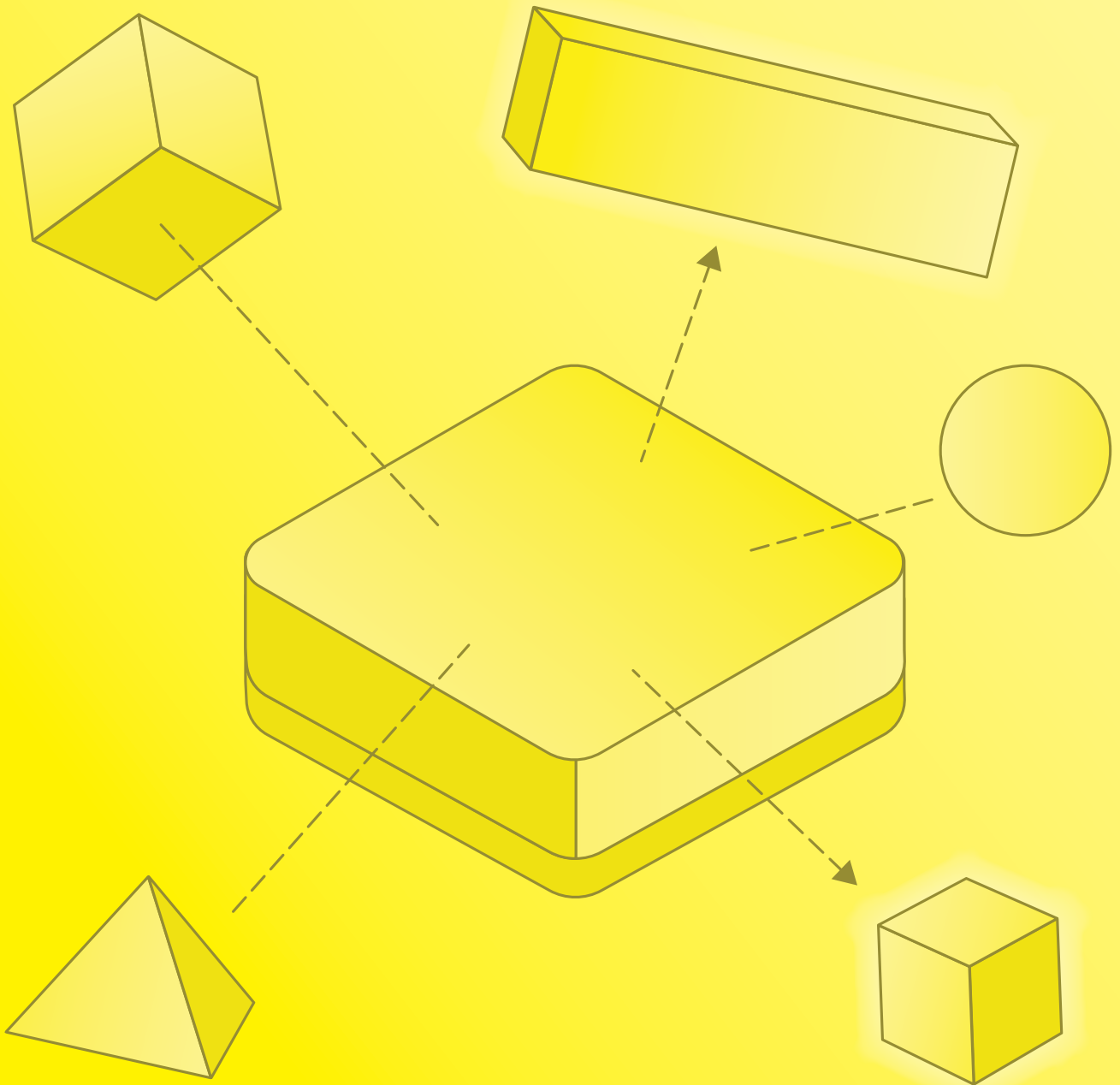
UNION BY NAME

The UNION BY NAME clause allows vertical stacking of datasets by matching columns by name rather than by position. This makes queries more concise and readable.

```
-- Calculate the union of two tables  
-- with a different number and order of columns
```

```
SELECT  
  'pi' AS constant,  
  3.14 AS value  
UNION BY NAME  
SELECT  
  6.28 AS value,  
  'tau' AS constant,  
  'defined as 2*pi' AS comment;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----------|-----------|
| | | | | | | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | | | | | |



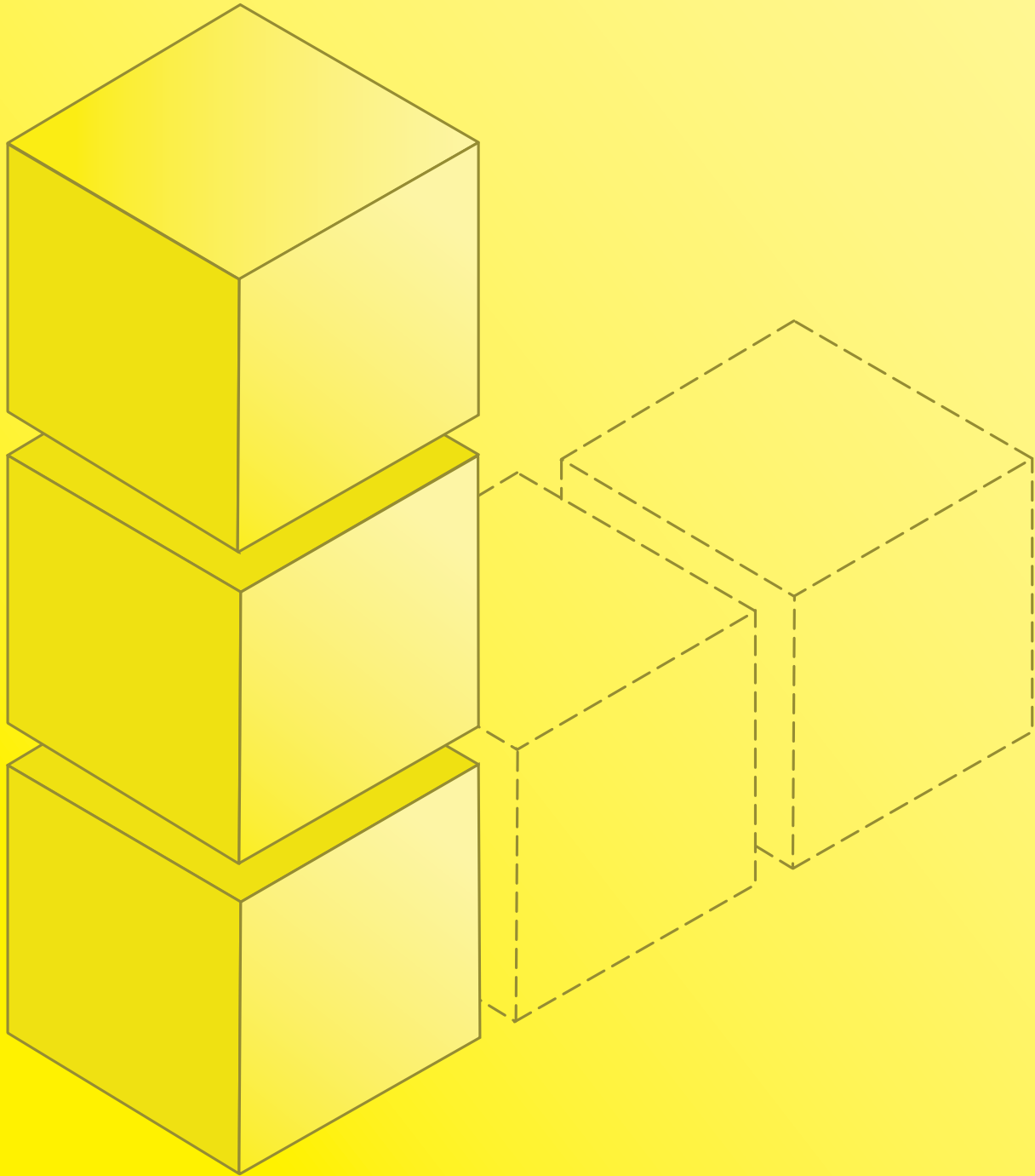
APRIL

FROM-FIRST SYNTAX

DuckDB can parse SQL queries where the `SELECT` and `FROM` clauses are swapped - this is called the `FROM-first` syntax. One immediate advantage of this syntax is that text editors already know the schema when you are typing the `SELECT` clause, allowing them to provide better suggestions. And here's a fan favorite: you can omit the `SELECT` clause entirely, and DuckDB will assume a `SELECT *` query.

```
-- Calculate the average delay of trains
FROM train_services
SELECT date, avg(delay)
WHERE NOT cancelled
GROUP BY date;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 | | | |



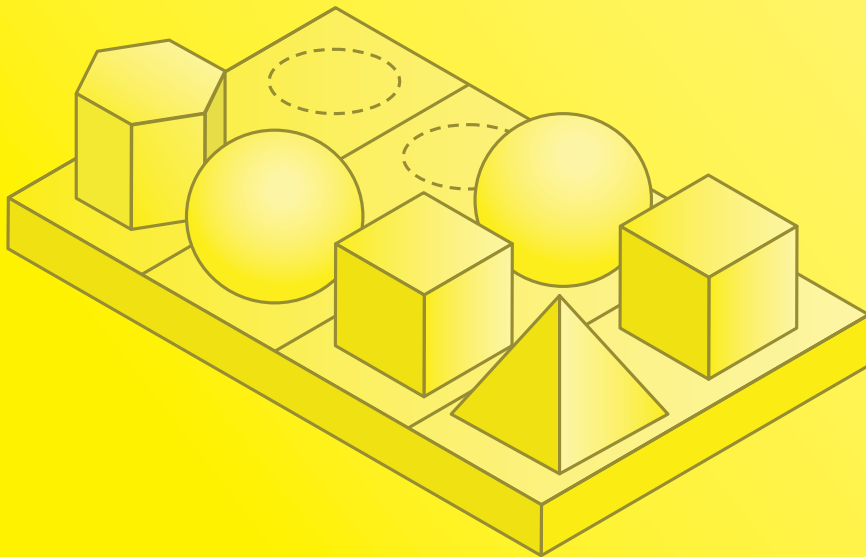
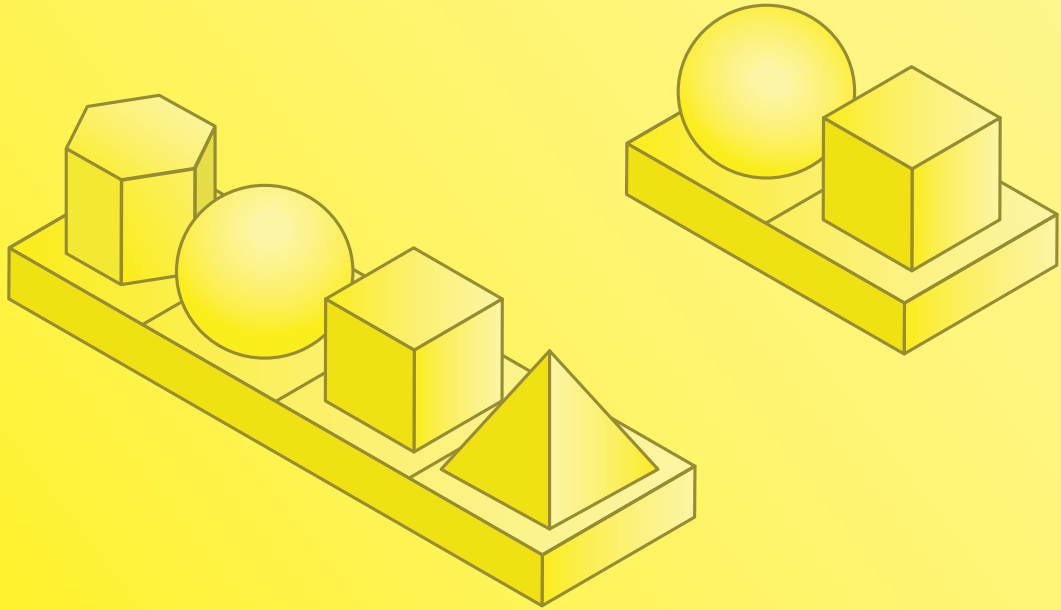
MAY

PIVOT

In 2015, Joel Spolsky of Stack Overflow fame gave a presentation for intermediate Excel users. While covering pivot tables, he remarked that about once every three months, a new startup reinvents pivot tables as a neat way of working with aggregate data. However, these startups always have trouble securing funding because venture capitalists already know about Excel's pivot table feature! Well, with the PIVOT statement, you already have this functionality at your disposal in DuckDB and don't need to rely on other implementations.

```
-- Pivot the purchases so each year has its own column
-- with the total count per item sold
PIVOT purchases
  ON year
  USING sum(count)
  GROUP BY item;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | | | | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 |



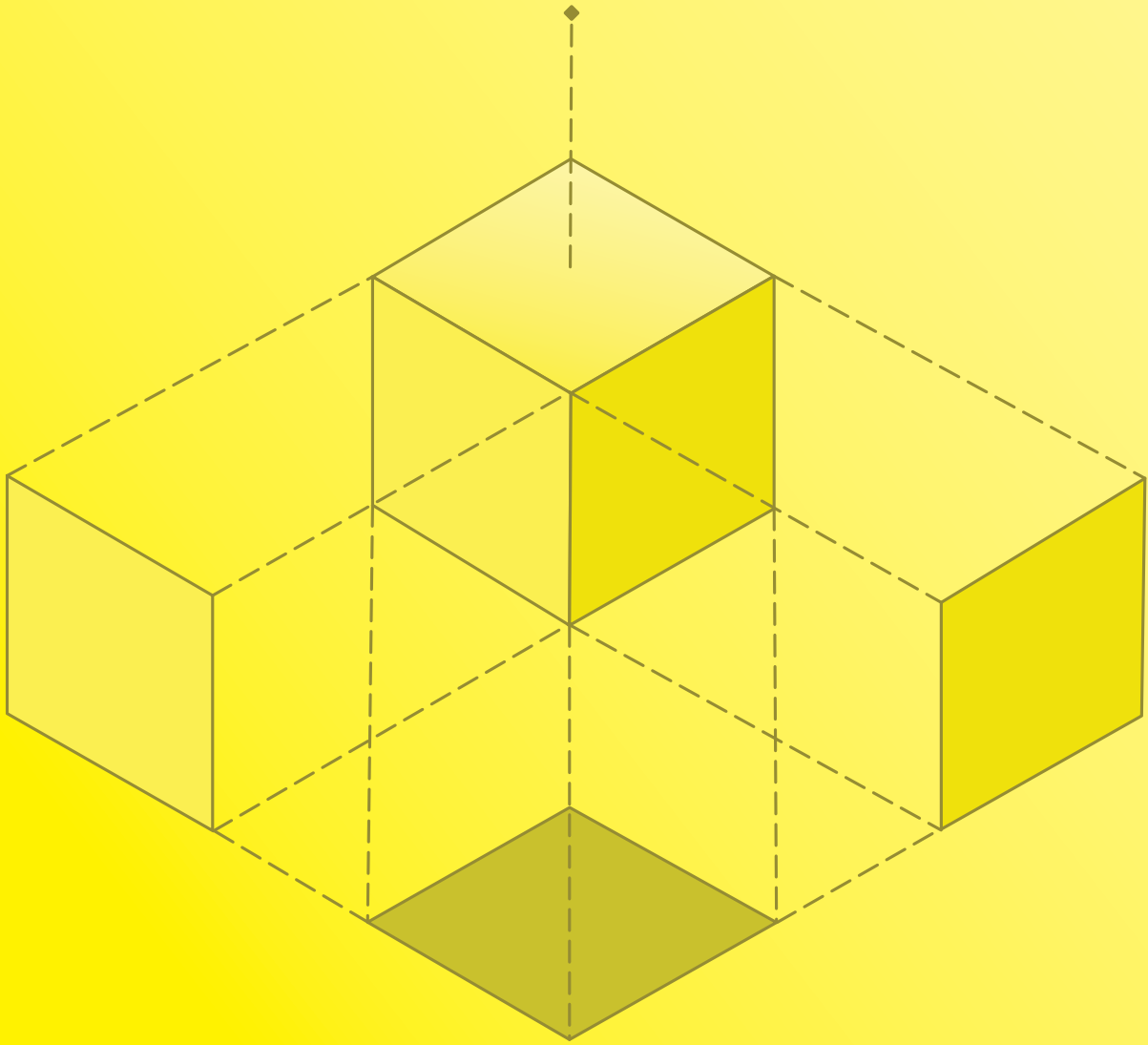
JUNE

POSITIONAL JOIN

DuckDB was designed to work closely with dataframe libraries, where the order of rows is often preserved between operations. In these libraries, common operations include slicing a column from a dataframe and concatenating dataframes together based on the positions of their rows. DuckDB's POSITIONAL JOIN operator captures the latter operation: it allows you to join tables based on the positions of their rows.

```
-- Produce rows with DuckDB version number and codename pairs
SELECT *
FROM
  (VALUES ('1.4'), ('1.3'), ('1.2')) t1(version_number)
POSITIONAL JOIN
  (VALUES ('Andium'), ('Ossivalis'), ('Histrionicus'))
t2(codename);
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | | | | | |



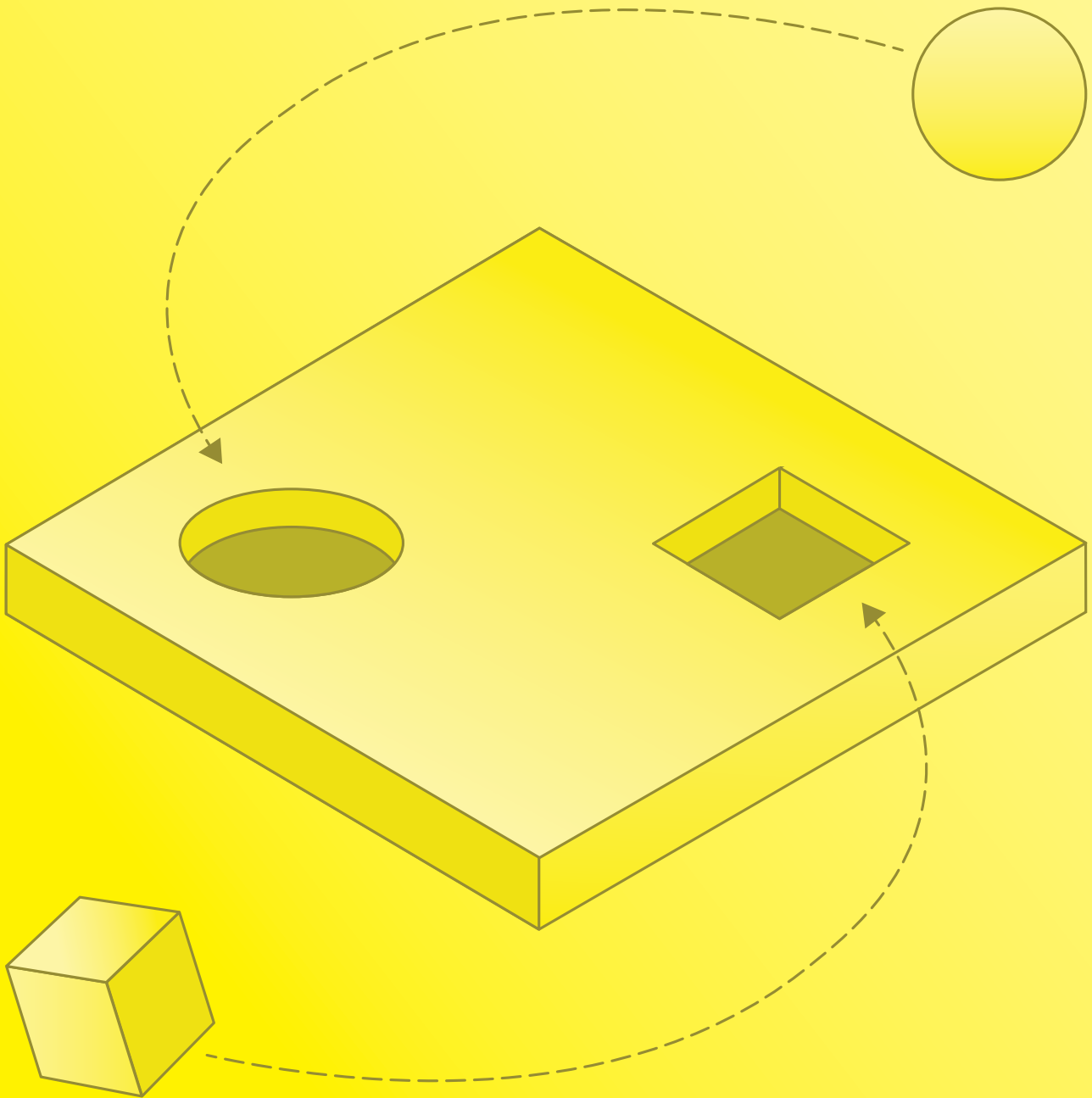
JULY

GROUP BY CUBE

You can use DuckDB's GROUP BY clause with the GROUPING SETS, ROLLUP, and CUBE keywords to perform aggregations over multiple dimensions within the same query. Notably, CUBE produces grouping sets for all combinations of its input.

```
-- Calculate the average income for {city, street_name},  
-- {city}, {street_name} and {} (full aggregation)  
SELECT city, street_name, avg(income)  
FROM addresses  
GROUP BY CUBE (city, street_name);
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 | 31 | | |



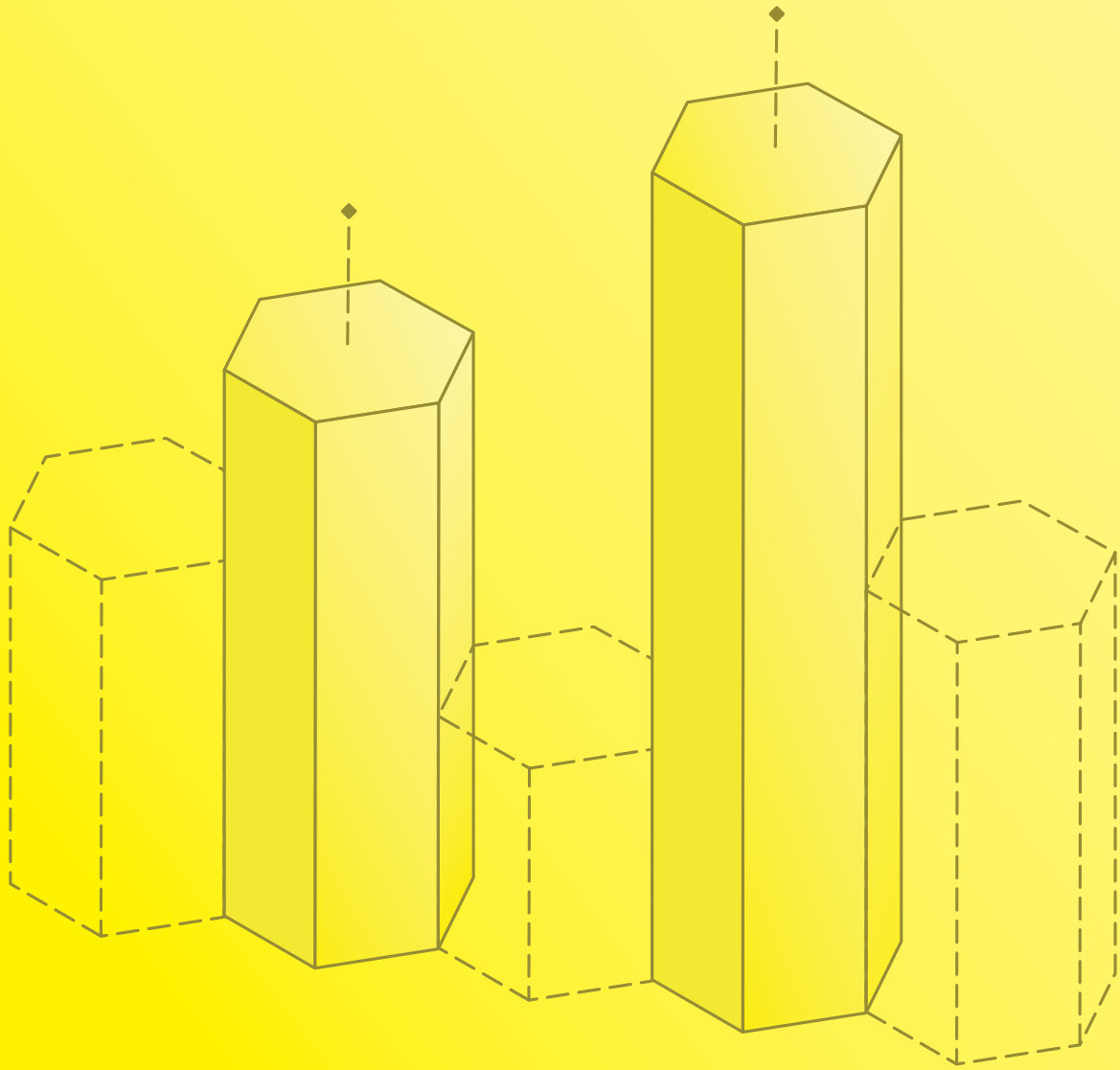
AUGUST

INSERT INTO ... BY NAME

DuckDB enhances the INSERT INTO statement with the BY NAME modifier. This variation matches the incoming data's column names against the columns of the target table. Unlike in vanilla INSERT INTO statements, the number and order of columns can differ. Omitted columns will be set to their default value.

```
-- Insert into the table with a missing column
CREATE TABLE tbl (year INTEGER, month INTEGER, day INTEGER);
INSERT INTO tbl BY NAME
  (SELECT 8 AS month, 2026 AS year);
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----------|-----------|
| | | | | | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | | | | | | |



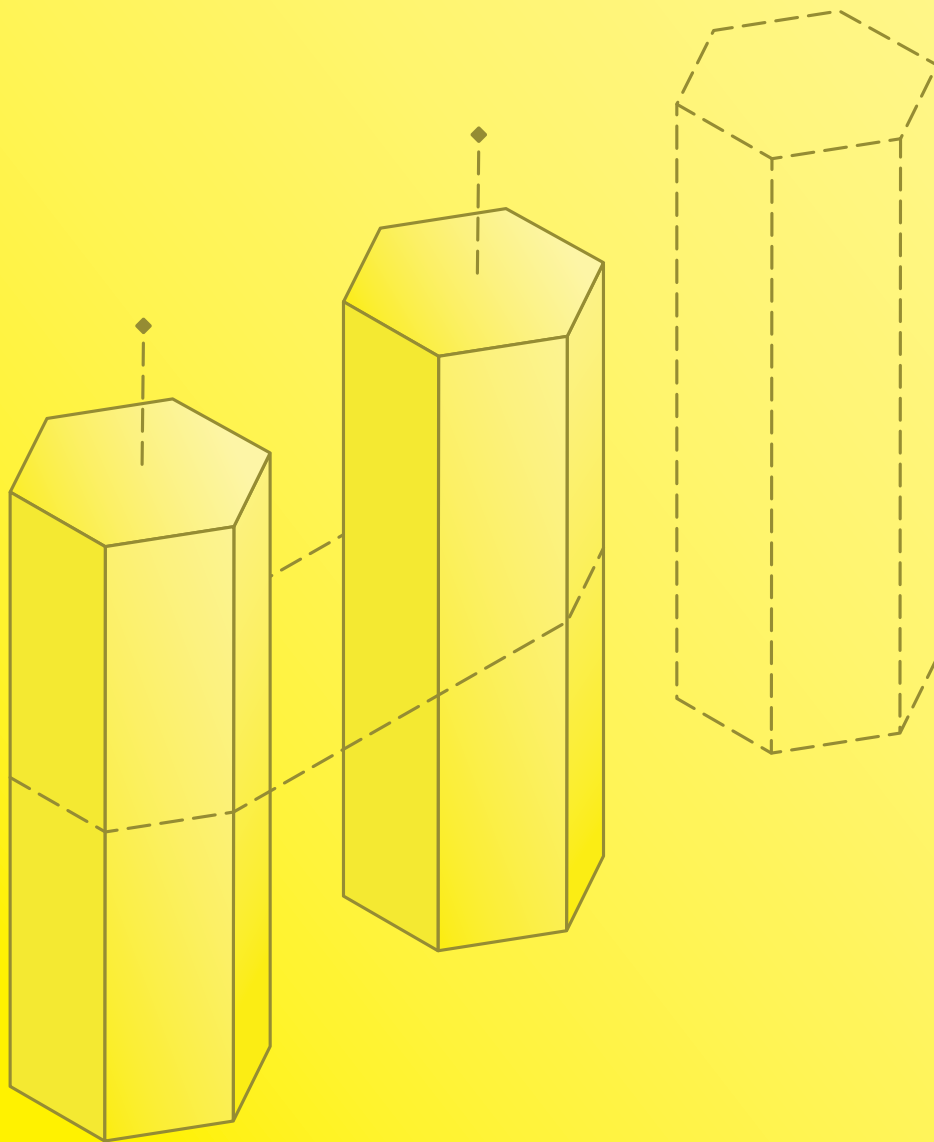
SEPTEMBER

MAX_BY FOR TOP-N SELECTION

The standard SQL solution for computing the values of a column for the top-N values of an expression is to use window functions. This approach not only introduces cumbersome syntax (`row_number`, `OVER`, `QUALIFY`, etc.) but also necessitates a full sort just to retrieve a few values. To address these issues, DuckDB introduces the `max_by` function (also available under the `arg_max` alias), which offers a simpler syntax and improved performance.

```
-- Calculate the top-3 most recent orders for each supplier
SELECT
    l_suppkey,
    max_by(l_orderkey, l_shipdate, 3) AS recent_orders
FROM lineitem
GROUP BY l_suppkey;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | | | | |



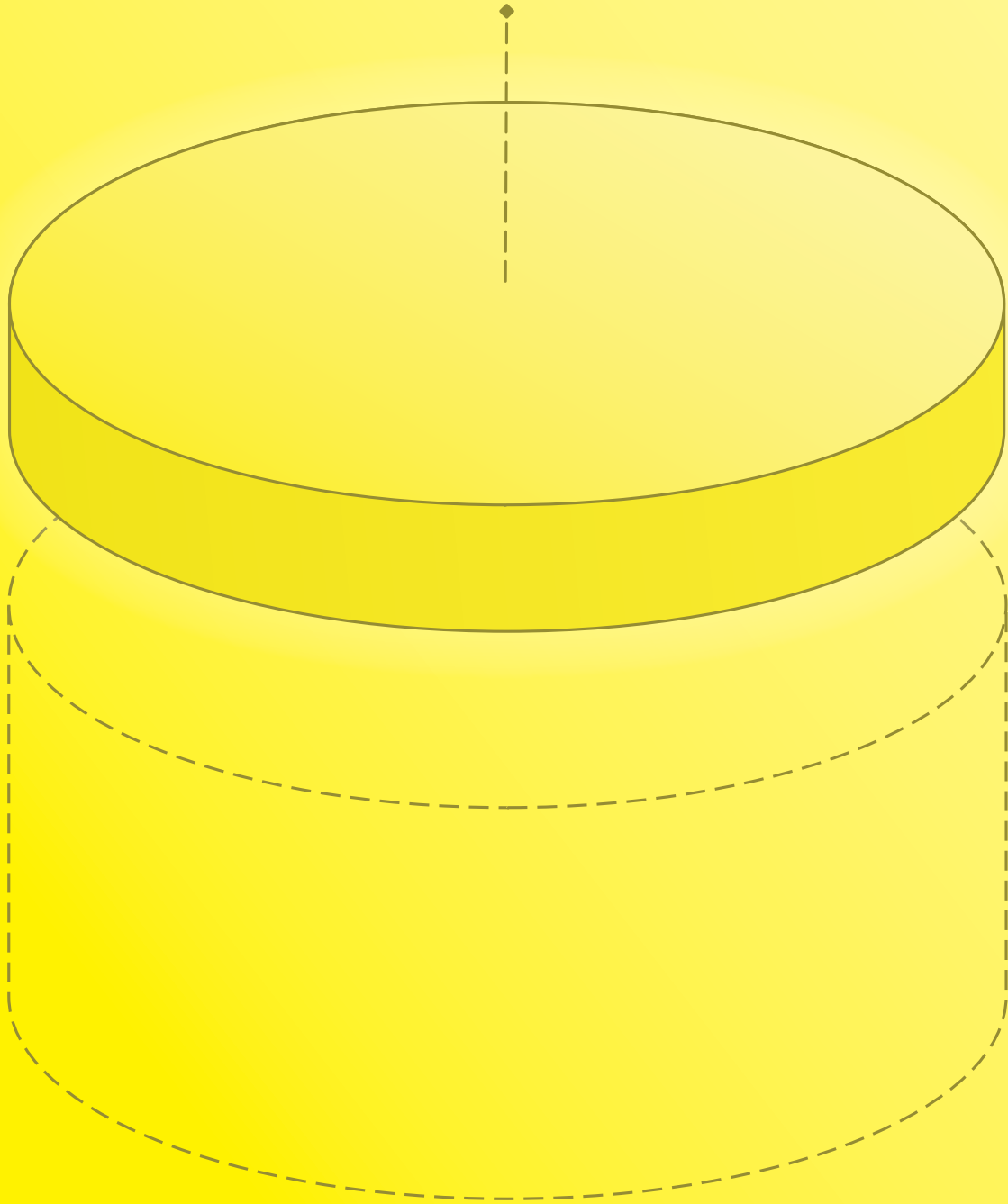
OCTOBER

COLUMNS EXPRESSION

The COLUMNS expression is similar to the star expression (as in SELECT *), but it is much more powerful. For example, you can evaluate the same SQL expression across multiple columns. You can also use the COLUMNS expression to filter column names and rename them using regular expressions.

```
-- Only return columns named Foo:<something>
-- and drop the colon in the middle
CREATE TABLE tbl (
  "Foo:Bar" INTEGER,
  "Foo:Baz" INTEGER,
  "Quz:Qux" INTEGER
);
SELECT COLUMNS('(Foo):(\w*)') AS '\1\2'
FROM tbl;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | | | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 | |



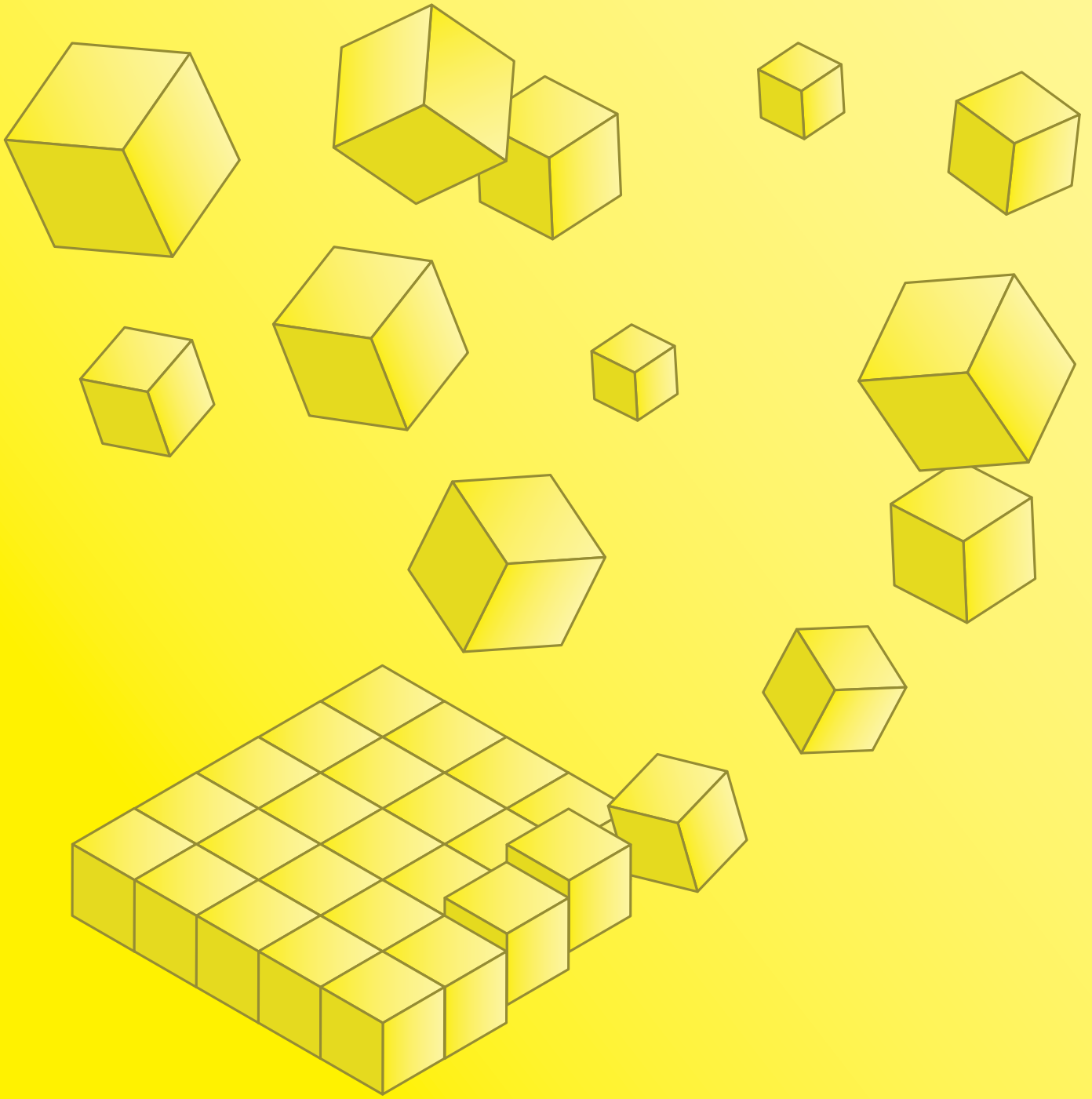
NOVEMBER

LIMIT %

DuckDB supports the LIMIT p% syntax to return p percent of the result table. If you need to obtain a random sample of the table, take a look at the USING SAMPLE clause.

```
-- Return 0.2% of the train services
SELECT *
FROM train_services
LIMIT 0.2%;
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----------|-----------|
| | | | | | | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | | | | | | |



DECEMBER

CREATE TABLE ... AS SELECT

DuckDB's CREATE TABLE ... AS syntax creates a table based on the output of a query. This syntax has a few variations, for example: (1) with CREATE OR REPLACE, you can make these statements idempotent so they work in repeated runs; (2) with DuckDB's CSV auto-detection feature, you can load a CSV file directly into a table; (3) using the FROM-first syntax, you can make these statements very concise.

```
-- Populate the train_services table using a remote CSV file
CREATE OR REPLACE TABLE train_services AS
  FROM 'https://blobs.duckdb.org/ca1/railway-services-2025-12.csv';
```

| MON | TUE | WED | THU | FRI | SAT | SUN |
|-----|-----|-----|-----|-----|-----|-----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |