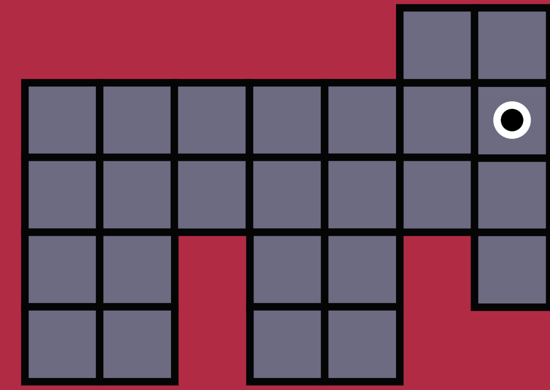
BUILDING TETRIS IN A SQL QUERY!

Nuno Faria, INESC TEC

PostgreSQL Conference Europe 2025 – Riga, Latvia



COMMON TABLE EXPRESSIONS

- Also known as CTEs.
- Introduced in SQL:1999. [1]
- Allow subqueries to be organized into auxiliary expressions.

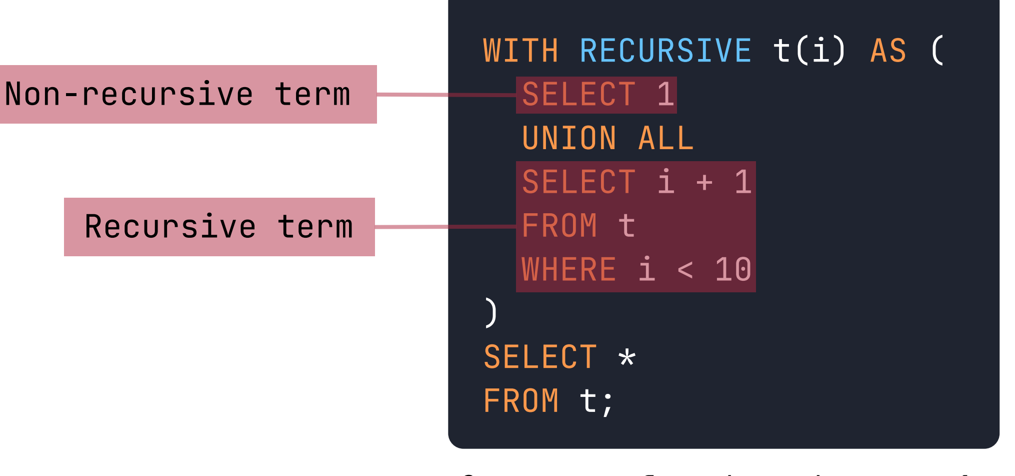
```
WITH avg_price_monthly(month, avg_price) AS (
  SELECT date_trunc('month', created),
         avg(price)
  FROM orders
  GROUP BY 1
)
SELECT o.id, o.price
FROM orders o
JOIN avg_price_monthly a
  ON a.month = date_trunc('month', o.created)
WHERE o.price > a.avg_price;
```

Orders with price above the respective monthly average.

[1] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL3. SIGMOD Rec. 28, 1 (March 1999), 131-138. https://doi.org/10.1145/309844.310075

RECURSIVE CTEs

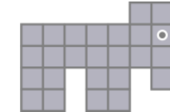
- Introduced in SQL:1999. [1]
- Allow CTEs to recursively refer to themselves, enabling complex workloads.
- Are evaluated iteratively in PostgreSQL. [2]



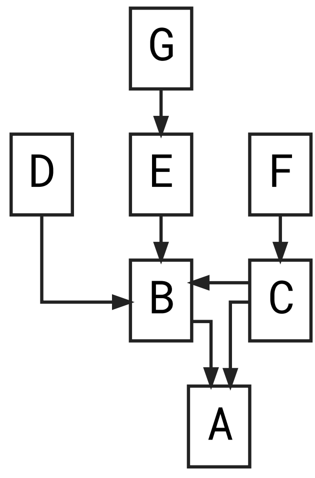
Sequence of numbers between 1 and 10

[1] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL3. SIGMOD Rec. 28, 1 (March 1999), 131-138. https://doi.org/10.1145/309844.310075

[2] https://www.postgresql.org/docs/18/queries-with.html#QUERIES-WITH-RECURSIVE



RECURSIVE CTEs

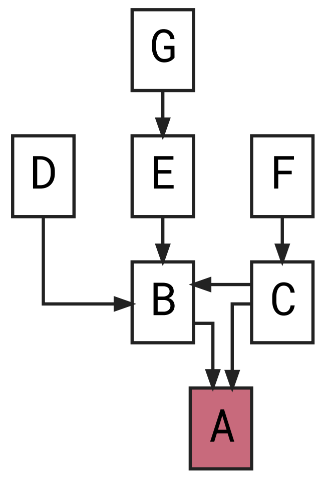


src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

```sql
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at ‘A’.

RECURSIVE CTEs



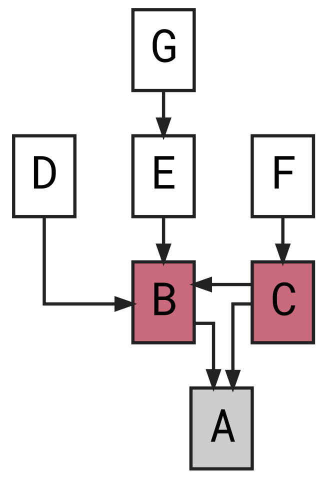
src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

```
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at 'A'.

Result	Working Table	Intermediate Table
{}	{}	{A}

RECURSIVE CTEs



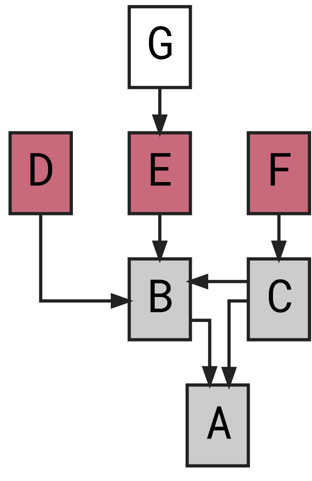
src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

```sql
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at 'A'.

Result	Working Table	Intermediate Table
{} {A}	{} {A}	{A} {B, C}

RECURSIVE CTEs

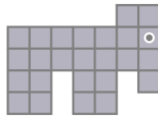


src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

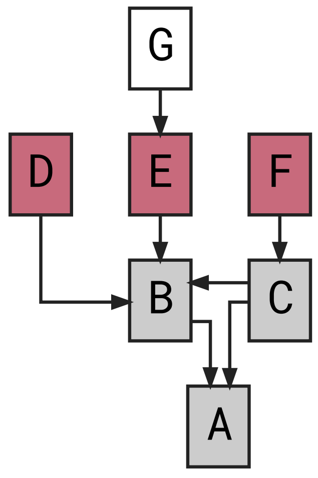
```sql
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at 'A'.

Result	Working Table	Intermediate Table
{}	{}	{A}
{A}	{A}	{B, C}
{A, B, C}	{B, C}	{C, D, E, F}



RECURSIVE CTEs



src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

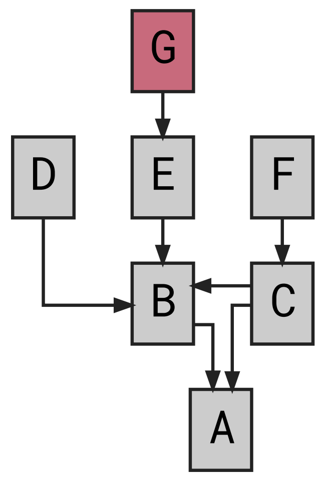
```sql
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at 'A'.

Result	Working Table	Intermediate Table
{}	{}	{A}
{A}	{A}	{B, C}
{A, B, C}	{B, C}	{C, D, E, F}

"UNION" prevents duplicate rows from being considered.

RECURSIVE CTEs



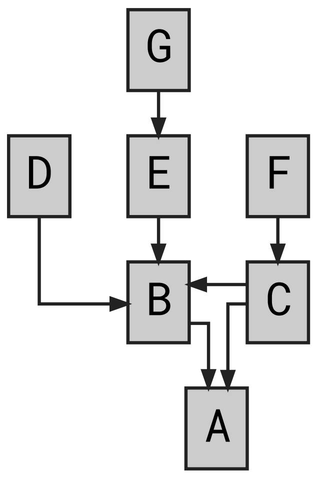
src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

```sql
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at ‘A’.

Result	Working Table	Intermediate Table
{}	{}	{A}
{A}	{A}	{B, C}
{A, B, C}	{B, C}	{C, D, E, F}
{A, B, C, D, E, F}	{D, E, F}	{G}

RECURSIVE CTEs



src	dst
B	A
C	A
C	B
D	B
E	B
F	C
G	E

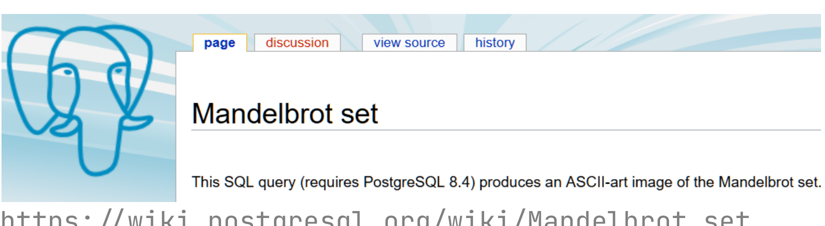
```sql
WITH RECURSIVE t (name) AS (
  SELECT 'A'::text
  UNION
  SELECT dependency.src
  FROM t
  JOIN dependency
    ON dependency.dst = t.name
)
SELECT *
FROM t;
```

Transitive closure of library dependencies, starting at 'A'.

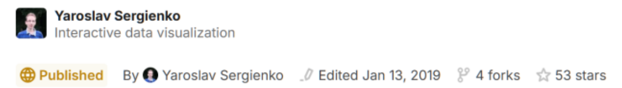
Result	Working Table	Intermediate Table
{}	{}	{A}
{A}	{A}	{B, C}
{A, B, C}	{B, C}	{C, D, E, F}
{A, B, C, D, E, F}	{D, E, F}	{G}
{A, B, C, D, E, F, G}	{G}	{}

RECURSIVE CTEs

- Addition of Recursive CTEs makes SQL Turing complete.
- Several real-world applications:
  - Bill of materials.
  - Find recommendations.
  - Determine reachability.
  - Shortest path between entities.
  - Ad-hoc generators.
- Several non-real-world applications too.

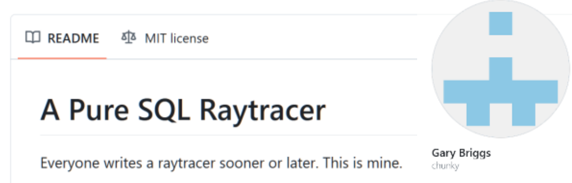


https://wiki.postgresql.org/wiki/Mandelbrot_set



SQL 3d engine (interactive preview)

https://observablehq.com/@pallada-92/sql-3d-engine

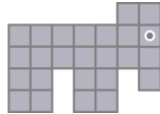


https://github.com/chunky/sqlraytracer

EXPLAIN EXTENDED

Happy New Year: GPT in 500 lines of SQL

https://explainextended.com/2023/12/31/happy-new-year-15/



WHY RECURSIVE CTEs

- Compute closer to the data, avoiding large data transfers / multiple round trips.
- Use existing indexes.
- Automatic and efficient disk spilling.

Data source: crates.io

Dependency table: 1.2M rows, 64MB

CTE

```
WITH RECURSIVE t (name) AS (
 SELECT '{target}'::text
 UNION
 SELECT dependency.src
 FROM t
 JOIN dependency
  ON dependency.dst = t.name
)
SELECT *
FROM t;
```

```
CTE Scan
 CTE t
  Recursive Union
   Result
   Subquery Scan
    Nested Loop
     WorkTable Scan on t t_1
     Index Only Scan
      Index Cond
       (dst = t_1.name)
```

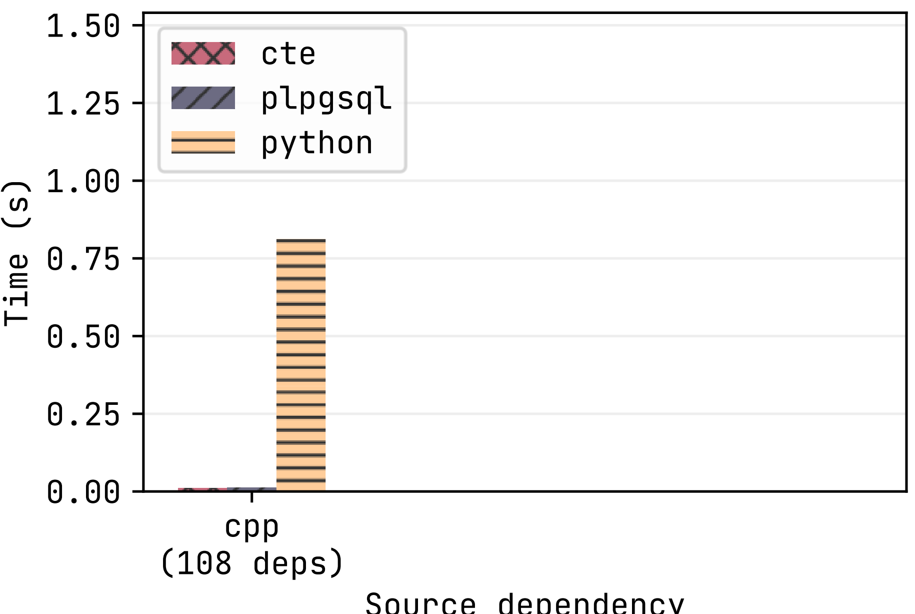
PLPGSQL

```
CREATE TEMP TABLE temp_t (
 name TEXT PRIMARY KEY
) ON COMMIT DROP;

INSERT INTO temp_t VALUES (target);
level := ARRAY[target];

WHILE level ≠ ARRAY[NULL] LOOP
 WITH t AS (
  INSERT INTO temp_t
  SELECT DISTINCT d.src
  FROM dependency d
  WHERE d.dst = ANY(level)
   AND d.src NOT IN (
     SELECT t_.name FROM temp_t t_)
  RETURNING temp_t.name
 )
 SELECT array_agg(t.name) INTO level
 FROM t;
END LOOP;

RETURN QUERY SELECT * FROM temp_t;
```

Python

```
cursor.execute("""
 select src, dst
 from dependency
""")

data = defaultdict(list)
for row in cursor.fetchall():
 data[row[1]].append(row[0])

result = set()
level = [target]

while len(level) > 0:
 next = set()
 for dep in level:
  if dep not in result:
   result.add(dep)
   next.update(data[dep])
 level = next
```

(Deployed in the same server as the database)



LIMITATIONS & CONSIDERATIONS

Limitation: Speed

CTE

```
WITH RECURSIVE t (name) AS (
 SELECT '{target}'::text
 UNION
 SELECT dependency.src
 FROM t
 JOIN dependency
  ON dependency.dst = t.name
)
SELECT *
FROM t;
```

PLPGSQL

```
CREATE TEMP TABLE temp_t (
 name TEXT PRIMARY KEY
) ON COMMIT DROP;

INSERT INTO temp_t VALUES (target);
level := ARRAY[target];

WHILE level ≠ ARRAY[NULL] LOOP
 WITH t AS (
  INSERT INTO temp_t
  SELECT DISTINCT d.src
  FROM dependency d
  WHERE d.dst = ANY(level)
   AND d.src NOT IN (
     SELECT t_.name FROM temp_t t_)
  RETURNING temp_t.name
 )
 SELECT array_agg(t.name) INTO level
 FROM t;
END LOOP;

RETURN QUERY SELECT * FROM temp_t;
```

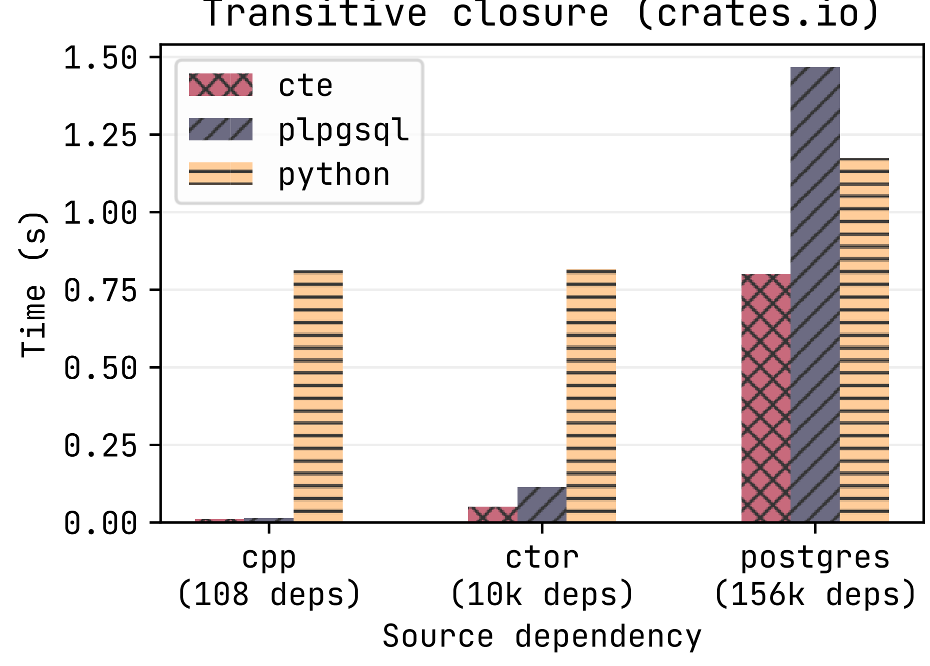
Python

```
cursor.execute("""
 select src, dst
 from dependency
""")

data = defaultdict(list)
for row in cursor.fetchall():
 data[row[1]].append(row[0])

result = set()
level = [target]

while len(level) > 0:
 next = set()
 for dep in level:
  if dep not in result:
   result.add(dep)
   next.update(data[dep])
 level = next
```

Transitive closure implementations.

Data source: crates.io

Dependency table: 1.2M rows, 64MB



LIMITATIONS & CONSIDERATIONS

Limitation: Speed

CTE

```sql
WITH RECURSIVE t (origin, name) AS (
 VALUES ('{target1}', '{target1}'), ...
 UNION
 SELECT t.origin, dependency.src
 FROM t
 JOIN dependency
  ON dependency.dst = t.name
)
SELECT *
FROM t;
```

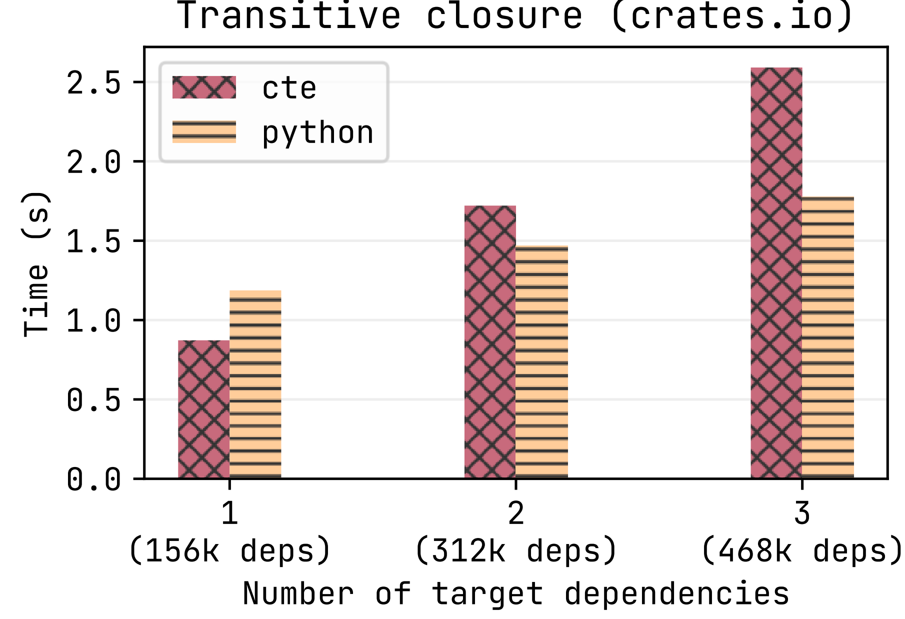
Python

```python
cursor.execute("""
 select src, dst
 from dependency
""")

data = defaultdict(list)
for row in cursor.fetchall():
 data[row[1]].append(row[0])

result = defaultdict(set)
level = {dep: {dep} for dep in target_dependencies}

while len(level) > 0:
 next = defaultdict(set)
 for origin, dependencies in level.items():
  for dependency in dependencies:
   if dependency not in result[origin]:
    result[origin].add(dependency)
    next[origin].update(data[dependency])
 level = next
```

Transitive closure implementations of multiple target dependencies.

Data source: crates.io

Dependency table: 1.2M rows, 64MB



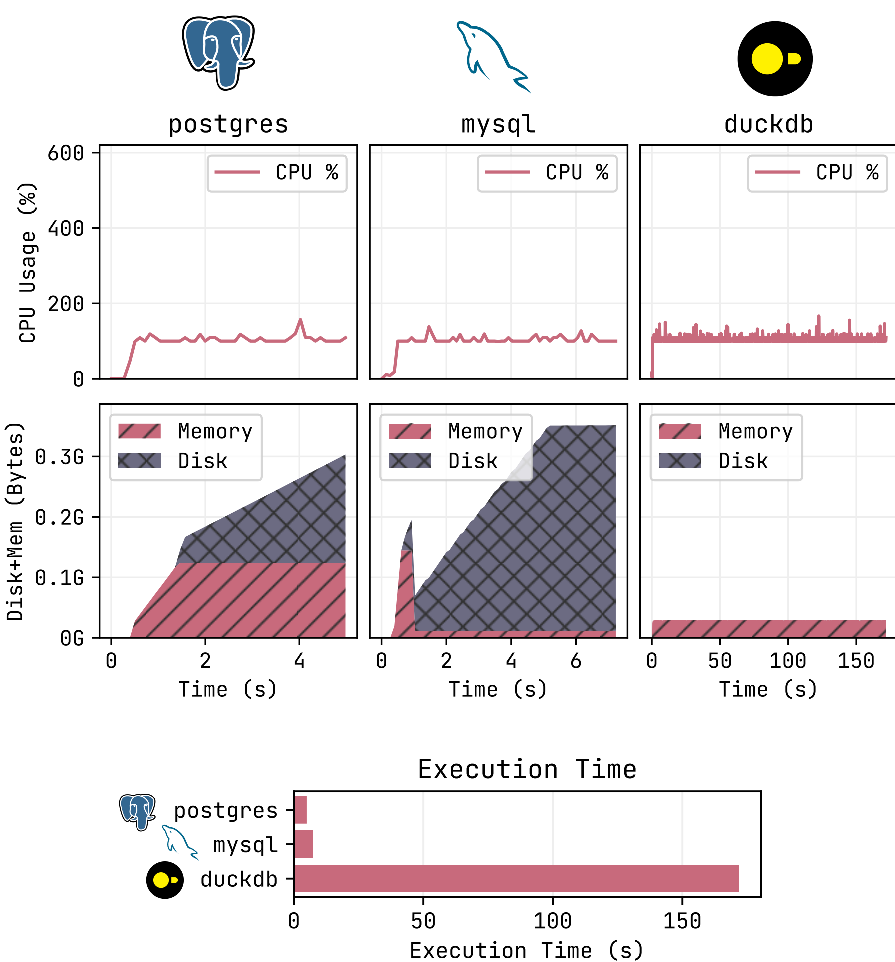
LIMITATIONS & CONSIDERATIONS

Limitation: Memory growth

```
WITH RECURSIVE t(i, data) AS (
    SELECT 1, '...'
    UNION ALL
    SELECT i + 1, '...'
    FROM t
    WHERE i < 10000000
)
SELECT data
FROM t
ORDER BY i DESC
LIMIT 1;
```

Recursive CTE where only the last row is kept.

- Postgres stores the entire CTE result. DuckDB optimizes this.

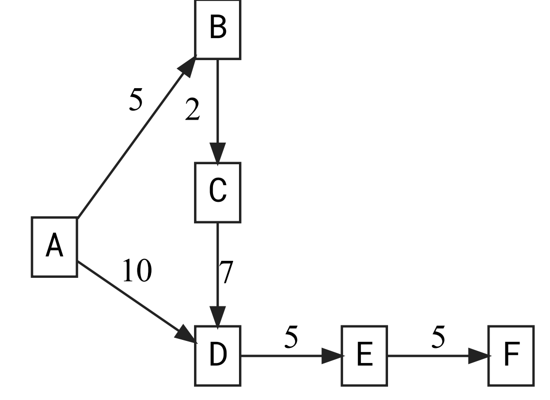


Postgres: work_mem=100MB | MySQL: tmp_table_size=200MB | DuckDB: threads=1

Limitation: Cannot access result set

```sql
WITH RECURSIVE paths AS (
    SELECT 'A'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION ALL
    SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
    FROM paths p
    JOIN edges e ON p.node = e.src -- join with neighbors
    WHERE NOT e.dst = ANY(p.path) -- prevent cycles
)
SELECT node, path, weight
FROM (
    SELECT *, rank() OVER (
        PARTITION BY node ORDER BY weight, path) AS rank
    FROM paths
)
WHERE rank = 1; -- best path for each destination
```

All shortest paths from source A to every destination.



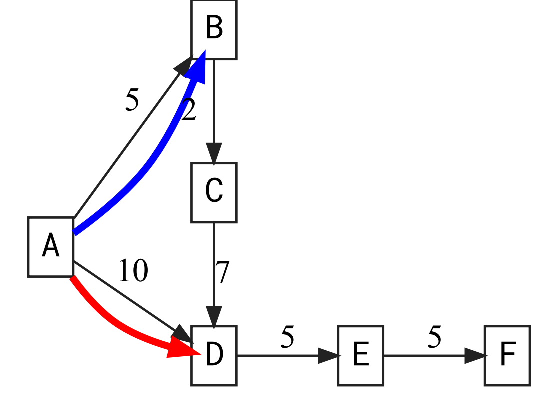
node	weight	path
A	0	{A}

LIMITATIONS & CONSIDERATIONS

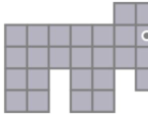
Limitation: Cannot access result set

```sql
WITH RECURSIVE paths AS (
    SELECT 'A'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION ALL
    SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
    FROM paths p
    JOIN edges e ON p.node = e.src -- join with neighbors
    WHERE NOT e.dst = ANY(p.path) -- prevent cycles
)
SELECT node, path, weight
FROM (
    SELECT *, rank() OVER (
        PARTITION BY node ORDER BY weight, path) AS rank
    FROM paths
)
WHERE rank = 1; -- best path for each destination
```

All shortest paths from source A to every destination.



node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}

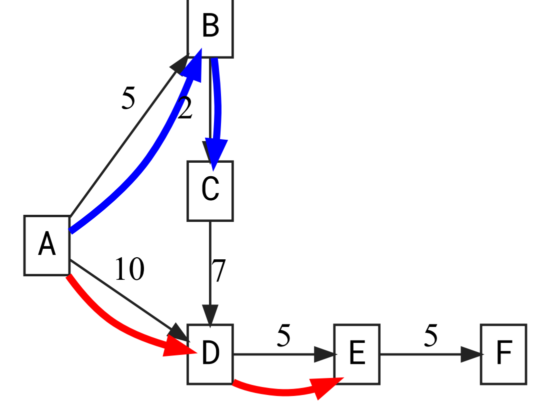


LIMITATIONS & CONSIDERATIONS

Limitation: Cannot access result set

```
WITH RECURSIVE paths AS (
    SELECT 'A'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION ALL
    SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
    FROM paths p
    JOIN edges e ON p.node = e.src -- join with neighbors
    WHERE NOT e.dst = ANY(p.path) -- prevent cycles
)
SELECT node, path, weight
FROM (
    SELECT *, rank() OVER (
        PARTITION BY node ORDER BY weight, path) AS rank
    FROM paths
)
WHERE rank = 1; -- best path for each destination
```

All shortest paths from source A to every destination.



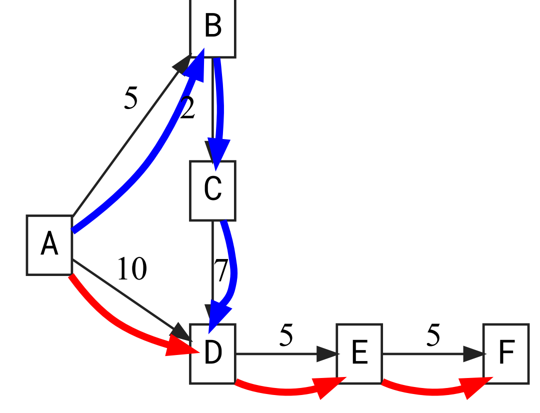
node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}
C	7	{A,B,C}
E	15	{A,D,E}

LIMITATIONS & CONSIDERATIONS

Limitation: Cannot access result set

```sql
WITH RECURSIVE paths AS (
    SELECT 'A'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION ALL
    SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
    FROM paths p
    JOIN edges e ON p.node = e.src -- join with neighbors
    WHERE NOT e.dst = ANY(p.path) -- prevent cycles
)
SELECT node, path, weight
FROM (
    SELECT *, rank() OVER (
        PARTITION BY node ORDER BY weight, path) AS rank
    FROM paths
)
WHERE rank = 1; -- best path for each destination
```

All shortest paths from source A to every destination.



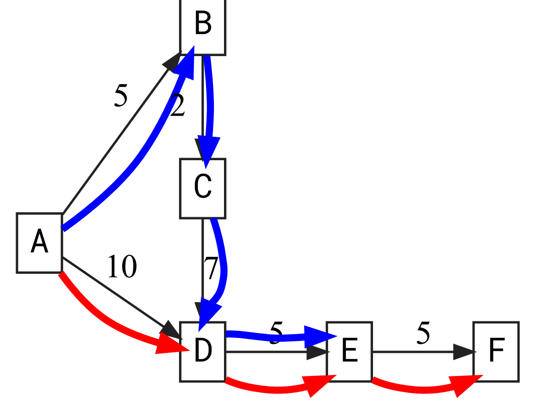
node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}
C	7	{A,B,C}
E	15	{A,D,E}
D	14	{A,B,C,D}
F	20	{A,D,E,F}

LIMITATIONS & CONSIDERATIONS

Limitation: Cannot access result set

```sql
WITH RECURSIVE paths AS (
    SELECT 'A'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION ALL
    SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
    FROM paths p
    JOIN edges e ON p.node = e.src -- join with neighbors
    WHERE NOT e.dst = ANY(p.path) -- prevent cycles
)
SELECT node, path, weight
FROM (
    SELECT *, rank() OVER (
        PARTITION BY node ORDER BY weight, path) AS rank
    FROM paths
)
WHERE rank = 1; -- best path for each destination
```

All shortest paths from source A to every destination.



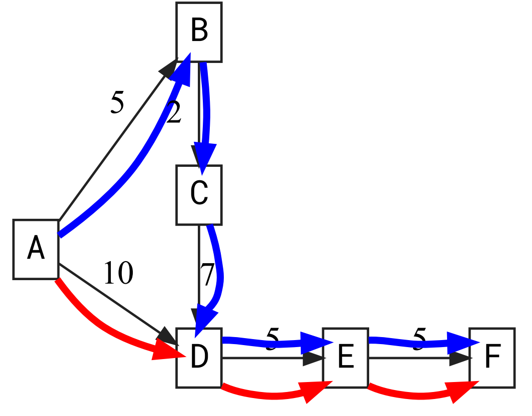
node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}
C	7	{A,B,C}
E	15	{A,D,E}
D	14	{A,B,C,D}
F	20	{A,D,E,F}
E	19	{A,B,C,D,E}

LIMITATIONS & CONSIDERATIONS

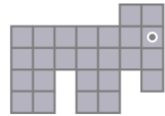
Limitation: Cannot access result set

```sql
WITH RECURSIVE paths AS (
    SELECT 'A'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION ALL
    SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
    FROM paths p
    JOIN edges e ON p.node = e.src -- join with neighbors
    WHERE NOT e.dst = ANY(p.path) -- prevent cycles
)
SELECT node, path, weight
FROM (
    SELECT *, rank() OVER (
        PARTITION BY node ORDER BY weight, path) AS rank
    FROM paths
)
WHERE rank = 1; -- best path for each destination
```

All shortest paths from source A to every destination.



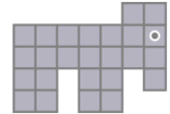
node	weight	path
A	0	{A}
------	---------	---------------
B	5	{A,B}
D	10	{A,D}
------	---------	---------------
C	7	{A,B,C}
E	15	{A,D,E}
------	---------	---------------
D	14	{A,B,C,D}
F	20	{A,D,E,F}
------	---------	---------------
E	19	{A,B,C,D,E}
------	---------	---------------
F	24	{A,B,C,D,E,F}



LIMITATIONS & CONSIDERATIONS

Limitation: Cannot access result set

- Non-optimal paths will be stored in the result set.
- Paths will still be computed from non-optimal sources.



LIMITATIONS & CONSIDERATIONS

Solution: “USING KEY”

- Allows past rows to be replaced based on the “key” (i.e., update instead of append-only).
- Allows access to the currently computed result table (using RECURRING.table).
- Recently added to DuckDB. [1,2]

[1] Björn Bamberg, Denis Hirn, and Torsten Grust. 2025. How DuckDB is USING KEY to Unlock Recursive Query Performance. In Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25). Association for Computing Machinery, New York, NY, USA, 31–34. https://doi.org/10.1145/3722212.3725107

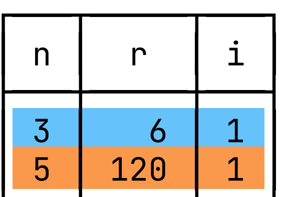
[2] https://duckdb.org/docs/stable/sql/query_syntax/with#recursive-ctes-with-using-key


```sql
WITH RECURSIVE t (n, r, i) AS (
    VALUES (3, 1, 3), (5, 1, 5)
    UNION
    SELECT n, r * i, i - 1
    FROM t
    WHERE i > 1
)
SELECT * FROM t;
```

Factorial of 3 and 5.

n	r	i
3	1	3
5	1	5
3	3	2
5	5	4
3	6	1
5	20	3
5	60	2
5	120	1

```sql
WITH RECURSIVE t (n, r, i)
 USING KEY (n) AS (
    VALUES (3, 1, 3), (5, 1, 5)
    UNION
    SELECT n, r * i, i - 1
    FROM t
    WHERE i > 1
)
SELECT * FROM t;
```



n	r	i
3	6	1
5	120	1

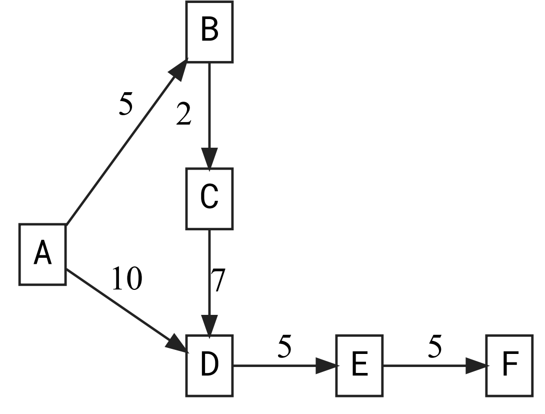
Factorial of 4 and 8 with “USING KEY”. New rows with the same “n” replace the previous ones in the result set.

LIMITATIONS & CONSIDERATIONS

Solution: "USING KEY"

```sql
WITH RECURSIVE paths USING KEY (node) AS (
    SELECT 'n-1'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION
    SELECT *
    FROM (
        SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
        FROM paths p
        JOIN edges e ON p.node = e.src
        WHERE p.weight + e.weight < coalesce( -- only if better
            (SELECT weight FROM RECURRING.paths WHERE node = e.dst),
            'inf'::float
        )
        ORDER BY 2 DESC -- in the same iter, the best is applied last
    )
)
SELECT * FROM paths;
```

All shortest paths from source A to every destination, with "USING KEY".



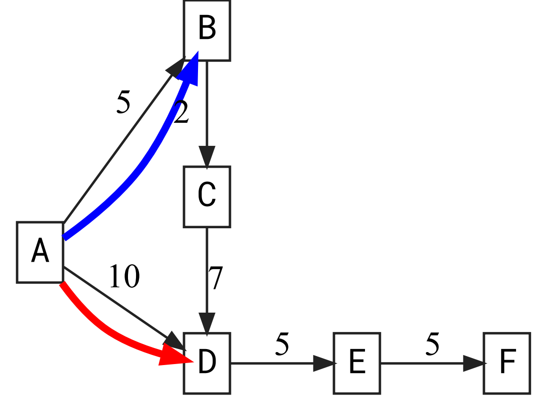
node	weight	path
A	0	{A}

LIMITATIONS & CONSIDERATIONS

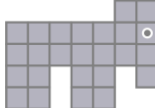
Solution: “USING KEY”

```sql
WITH RECURSIVE paths USING KEY (node) AS (
    SELECT 'n-1'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION
    SELECT *
    FROM (
        SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
        FROM paths p
        JOIN edges e ON p.node = e.src
        WHERE p.weight + e.weight < coalesce( -- only if better
            (SELECT weight FROM RECURRING.paths WHERE node = e.dst),
            'inf'::float
        )
        ORDER BY 2 DESC -- in the same iter, the best is applied last
    )
)
SELECT * FROM paths;
```

All shortest paths from source A to every destination, with “USING KEY”.



node	weight	path
A	0	{A}
------	--------	-------------
B	5	{A,B}
D	10	{A,D}

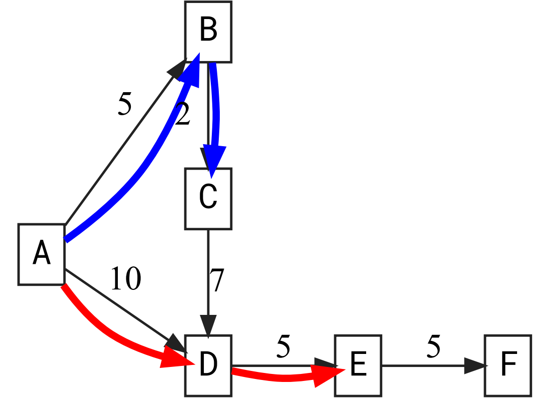


LIMITATIONS & CONSIDERATIONS

Solution: “USING KEY”

```sql
WITH RECURSIVE paths USING KEY (node) AS (
    SELECT 'n-1'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION
    SELECT *
    FROM (
        SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
        FROM paths p
        JOIN edges e ON p.node = e.src
        WHERE p.weight + e.weight < coalesce( -- only if better
            (SELECT weight FROM RECURRING.paths WHERE node = e.dst),
            'inf'::float
        )
        ORDER BY 2 DESC -- in the same iter, the best is applied last
    )
)
SELECT * FROM paths;
```

All shortest paths from source A to every destination, with “USING KEY”.



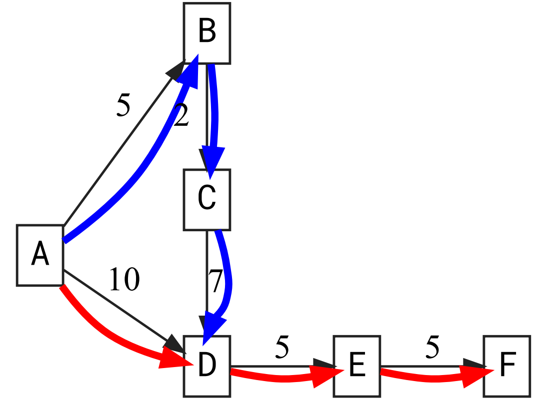
node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}
C	7	{A,B,C}
E	15	{A,D,E}

LIMITATIONS & CONSIDERATIONS

Solution: “USING KEY”

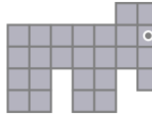
```sql
WITH RECURSIVE paths USING KEY (node) AS (
    SELECT 'n-1'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION
    SELECT *
    FROM (
        SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
        FROM paths p
        JOIN edges e ON p.node = e.src
        WHERE p.weight + e.weight < coalesce( -- only if better
            (SELECT weight FROM RECURRING.paths WHERE node = e.dst),
            'inf'::float
        )
        ORDER BY 2 DESC -- in the same iter, the best is applied last
    )
)
SELECT * FROM paths;
```

All shortest paths from source A to every destination, with “USING KEY”.



node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}
C	7	{A,B,C}
E	15	{A,D,E}
D	14	{A,B,C,D}
F	20	{A,D,E,F}

“WHERE” filter prevents this path from being added.

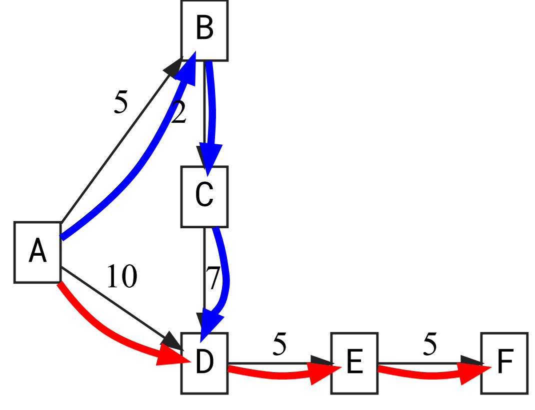


LIMITATIONS & CONSIDERATIONS

Solution: “USING KEY”

```sql
WITH RECURSIVE paths USING KEY (node) AS (
    SELECT 'n-1'::varchar AS node, 0 AS weight, ARRAY['A'] AS path
    UNION
    SELECT *
    FROM (
        SELECT e.dst, p.weight + e.weight, array_append(p.path, e.dst)
        FROM paths p
        JOIN edges e ON p.node = e.src
        WHERE p.weight + e.weight < coalesce( -- only if better
            (SELECT weight FROM RECURRING.paths WHERE node = e.dst),
            'inf'::float
        )
        ORDER BY 2 DESC -- in the same iter, the best is applied last
    )
)
SELECT * FROM paths;
```

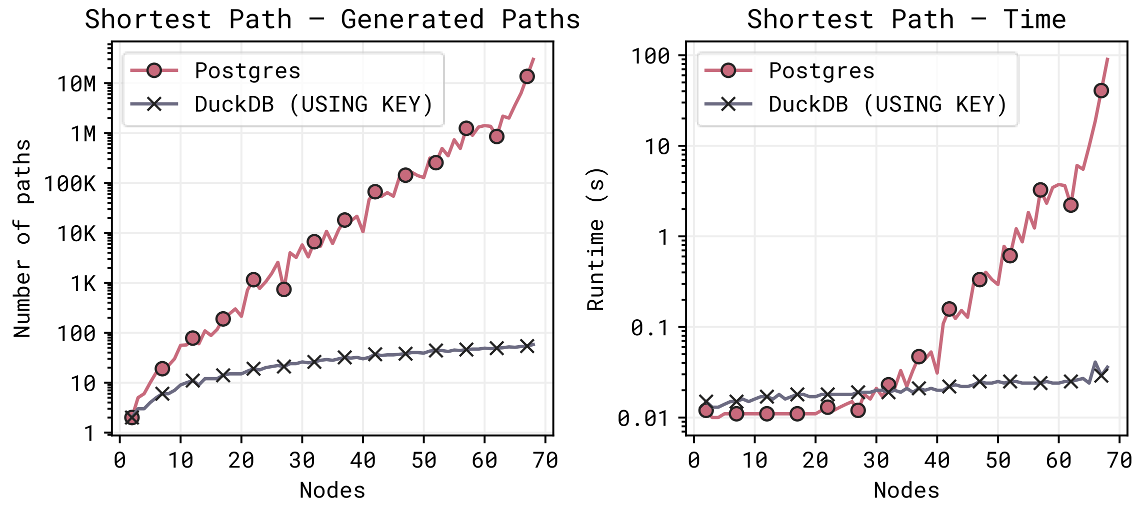
All shortest paths from source A to every destination, with “USING KEY”.



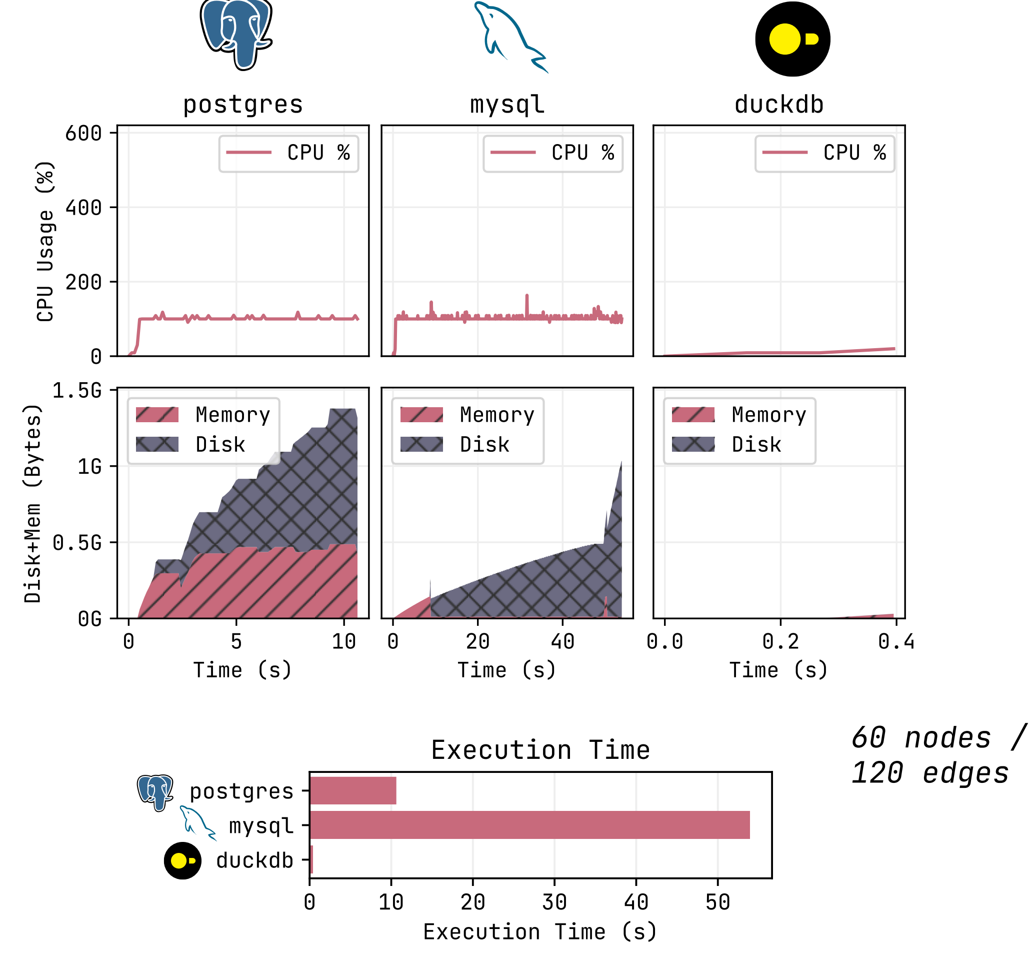
node	weight	path
A	0	{A}
B	5	{A,B}
D	10	{A,D}
C	7	{A,B,C}
E	15	{A,D,E}
F	20	{A,D,E,F}

LIMITATIONS & CONSIDERATIONS

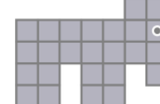
Solution: “USING KEY”



Total paths and execution time with a variable number of nodes (2 edges per node).



60 nodes / 120 edges



LIMITATIONS & CONSIDERATIONS

```
SET parallel_setup_cost = 0;
SET parallel_tuple_cost = 0;
SET min_parallel_table_scan_size = 0;
SET max_parallel_workers_per_gather = 6;
```

Limitation: Parallelism

- Parallel scans are not used with CTEs. [1]
  It might be faster to use a temporary table.

```
WITH RECURSIVE t(i) AS (
    SELECT 1
    UNION ALL
    SELECT i + 1
    FROM t
    WHERE i < 1000000
)
SELECT *
FROM t
WHERE i::text ~ '^(.*?){100}$';
```

Expensive regex filter + recursive CTE.

```
CREATE UNLOGGED TABLE t AS
    WITH RECURSIVE t(i) AS (...)
    SELECT *
    FROM t;

SELECT *
FROM t
WHERE i::text ~ '^(.*?){100}$';

DROP TABLE t;
```

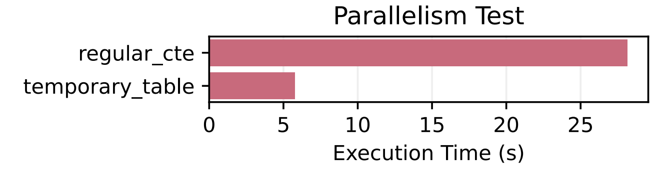
Expensive regex filter + table.

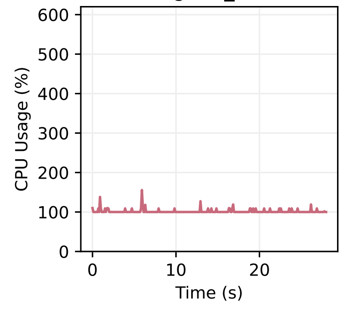
```
CTE Scan on t
 Filter: ...
 CTE t
  Recursive Union
   Result
   WorkTable Scan on t t_1
    Filter: (i < 1000000)
```

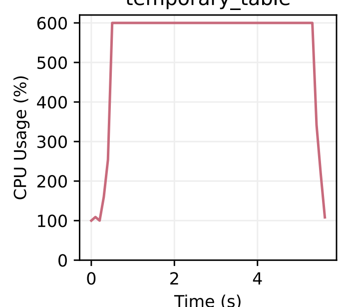
Plan excerpt for "Expensive regex filter + recursive CTE".

```
Gather
 Parallel Seq Scan on t
  Filter: ...
```

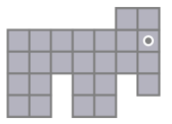
Plan excerpt for "Expensive regex filter + table".







[1] https://www.postgresql.org/docs/18/parallel-safety.html



LIMITATIONS & CONSIDERATIONS

Consideration: Isolation

- All iterations in a CTE execute over the same snapshot, even when using READ COMMITTED. [1]

```
WITH RECURSIVE t(i, value) AS (
    SELECT 1 AS i, value
    FROM data
    UNION ALL
    SELECT i + 1, data.value
    FROM t, data
    WHERE i < 5
)
SELECT *
FROM t;
```

Recursive CTE that reads 5 times from table “Data”.

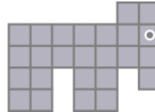
Transaction 1

i	value
1	A
2	A
3	A
4	A
5	A

Transaction 2

```
BEGIN;
 UPDATE data SET value = 'B';
COMMIT;
```

[1] https://www.postgresql.org/docs/18/transaction-iso.html#XACT-READ-COMMITTED



WHEN NOT TO USE RECURSIVE CTEs

- When it's faster to compute externally (number of CTE iterations is large and source data is relatively small).
- When the algorithm requires access to the current result set (solved with “USING KEY”).
- When the algorithm requires complex logic that is not easily expressed in SQL.

TETRIS IN A SQL QUERY

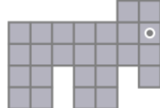
Game Loop

- Recursive CTEs to provide the looping logic.
- “pg_sleep” function to implement a consistent frame rate. [1]

```
WITH RECURSIVE main AS (
  -- initial state
  SELECT ...
  UNION ALL
  -- game loop
  SELECT ..., pg_sleep(...)
  FROM main, ...
  WHERE NOT game_over
)
...
```

Basic game structure.

[1] https://www.postgresql.org/docs/18/functions-datetime.html#FUNCTIONS-DATETIME-DELAY



TETRIS IN A SQL QUERY

Output

- Problem: cannot rely on the query's output, since in "psql" the output only starts to appear when the query completes.
- Solution: use the "RAISE" command to render the output. [1]

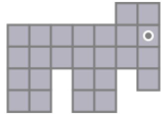
```
CREATE OR REPLACE FUNCTION notify(str varchar)
RETURNS void AS $$
BEGIN
  RAISE NOTICE '%', str;
END
$$ LANGUAGE PLPGSQL;
```

Function to use "RAISE" in regular SQL code.

```
WITH RECURSIVE t(i, output, sleep) AS (
  SELECT 1, null::void, null::void
  UNION ALL
  SELECT i + 1,
    notify(
      repeat(E'\n', 10) ||
      repeat(' ', 20) || (i + 1)::varchar ||
      repeat(E'\n', 10)),
    pg_sleep(1)
  FROM t
)
SELECT *
FROM t;
```

Example: printing a sequence of numbers with some formatting.

[1] https://www.postgresql.org/docs/18/plpgsql-errors-and-messages.html#PLPGSQL-STATEMENTS-RAISE



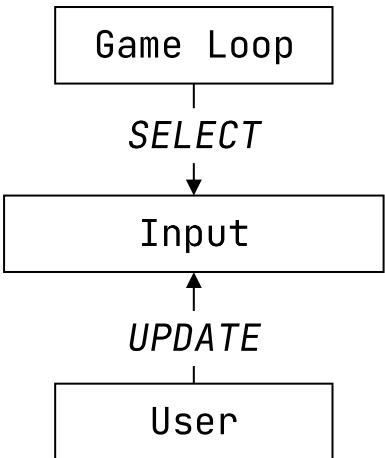
TETRIS IN A SQL QUERY

Input

- Problem: query executes server-side, cannot directly read client-side commands.
- Solution: use a single-row table “Input” a as communication bus.
- Also add a timestamp to ensure the same input is processed only once.

```
CREATE TABLE Input (cmd char, ts timestamp);
```

Table to store the user inputs.



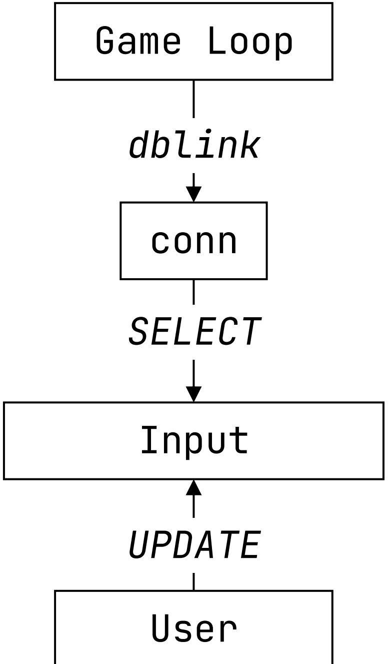
TETRIS IN A SQL QUERY

Input

- Problem: the isolation prevents the game loop from reading updates to “Input”.
- Solution: use “dblink” to read in a new txn. [1]

```
WITH RECURSIVE main AS (
  WITH conn AS (
    SELECT 'conn' AS name, dblink_connect('conn', 'dbname=' || current_database())
  )
  SELECT notify('')
  UNION ALL
  SELECT notify(input.cmd)
  FROM main, conn, dblink(conn.name, 'SELECT * FROM Input') input(cmd char, ts timestamp)
)
```

Using “dblink” to read the “Input” table in a new snapshot. The connection is created only once.



[1] https://www.postgresql.org/docs/18/contrib-dblink-function.html

TETRIS IN A SQL QUERY

Input

- Problem: “Memoize” operator caches the “dblink” query, preventing updates from being read.

```
Memoize
 Cache Key: conn.name
 Cache Mode: binary
 Hits: 998 Misses: 1 Evictions: 0 Overflows: 0 Memory Usage: 1kB
   Function Scan on dblink input (... rows=1 loops=1)
```

Excerpt of the plan showing the “dblink” query being cached (LIMIT 1000).

- Solution: append a unique identifier (e.g., current frame number) to prevent caching. (Alternative: disable both “enable_memoize” and “enable_material”.)

```
dblink(conn.name,
       'SELECT * FROM Input --' || i
) input(cmd char, ts timestamp)
```

```
Function Scan on dblink input
  (... rows=1 loops=999)
```

Appending “i” to prevent caching.

```
NOTICE:  a
NOTICE:  a
NOTICE:  a
NOTICE:  b
NOTICE:  b
NOTICE:  b
NOTICE:  c
NOTICE:  c
NOTICE:  c
```

```
UPDATE Input SET cmd = 'b'
```

```
UPDATE Input SET cmd = 'c'
```

Recursive CTE can now read the updated input.

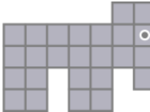
TETRIS IN A SQL QUERY

Game State

- Constant data:
  - *const* – constant parameters.
  - *points_per_line* – points awarded per number of lines cleared.
  - *tetromino* – information about each piece, in all rotations: id (0-6), rotation (0-3), and initial position (array with 4 integers).
  - *conn* – single row table that starts the “dblink” connection.

```
WITH RECURSIVE main AS (
  -- Constant parameters
  WITH const AS (
    SELECT
      -- board width
      10 AS width,
      -- board height
      20 AS height,
      -- frames per second
      60 AS fps,
      -- initial interval between piece drops,
i.e., gravity (secs)
      48/60.0 AS init_drop_delta,
      -- min interval between piece drops (secs)
      6/60.0 AS min_drop_delta,
      -- amount to decrease the drop interval
per each level (secs)
      2/60.0 AS drop_delta_decrease,
      ...
```

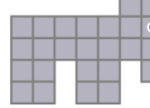
Excerpt of the “const” table.



TETRIS IN A SQL QUERY

Game State

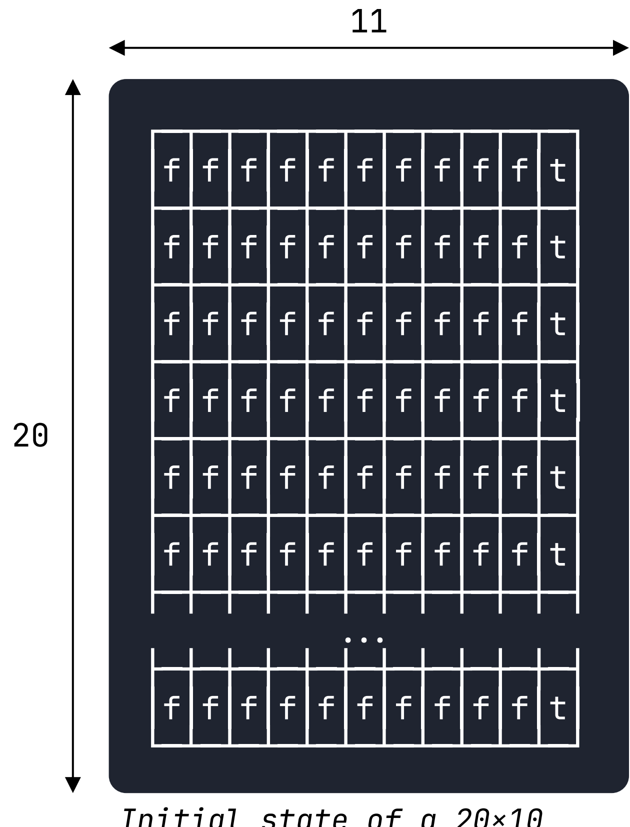
- Variable data (recursive CTE working row):
  - *frame* (int)
  - *board* (boolean[])
  - *score* (int)
  - *lines* (int)
  - *drop_delta* (numeric)
  - *pos* (int[])
  - *max_drop_lines* (int)
  - *next_piece* (integer)
  - *last_drop_time* (timestamp)
  - *last_input_time* (timestamp)
  - *render* (void)
  - *pg_sleep* (void)
  - *last_frame_time* (timestamp)



TETRIS IN A SQL QUERY

Game State

- Board encoding:
  - Boolean 1D array (*false*: empty, *true* → filled) (1D arrays are easier to work than 2D in PostgreSQL).
  - There is an additional, always-filled column in the board, to help collision detection logic.



Initial state of a 20×10 board (extra column is hidden from the user).

TETRIS IN A SQL QUERY

Game Logic

- Basic structure:
  - Read input → Update state → Render.
  - Subqueries + Lateral Joins (allows the current row to be used inside the respective subquery).
  - Stop when we cannot drop any lines (piece stuck at spawn).
  - Project the final score at the end.

```sql
WITH RECURSIVE main AS (
  ... -- constant data declaration
  SELECT
    ...
    notify(render.string),
    pg_sleep(...),
    ...
  UNION ALL
  SELECT
    ...
  FROM main,
    const,
    conn,
    dblink(...) input (cmd char, ts timestamp),
    LATERAL ( ... ) movement,
    LATERAL ( ... ) next_board,
    LATERAL ( ... ) drop_piece,
    LATERAL ( ... ) next_piece,
    LATERAL ( ... ) render
  WHERE main.max_drop_lines ⩾ 0
)
SELECT 'score: ' || max(score) AS game_over
FROM main;
```

Overview of Tetris query.

TETRIS IN A SQL QUERY

Game Logic

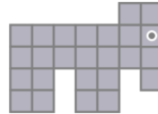
- Movement:
  - *pos* (int[]) column contains:
    - id of the current piece.
    - id of the current rotation.
    - number of cells moved based on the initial piece position.
    - if the piece moved down.

```
tetromino(id, rotation, piece) AS (
  SELECT id, rotation, piece
  FROM const c(w), LATERAL (
  VALUES
    ...
    -- T
    (2, 0, ARRAY[3, 4, 5, (c.w+1) + 4]),
    (2, 1, ARRAY[-(c.w+1) + 4, 3, 4, (c.w+1) + 4]),
    (2, 2, ARRAY[-(c.w+1) + 4, 3, 4, 5]),
    (2, 3, ARRAY[-(c.w+1) + 4, 4, 5, (c.w+1) + 4]),
    ...
```

Excerpt of the “tetromino” table.

```
     [2, 0, 0, 0]
+--------------------+
|. . . [][][]. . . . |
|. . . . []. . . . . |
|. . . . . . . . . . |
```

“pos” example.



TETRIS IN A SQL QUERY

Game Logic

- Movement:
    - *pos* (int[]) column contains:
        - id of the current piece.
        - id of the current rotation.
        - number of cells moved based on the initial piece position.
        - if the piece moved down.

```
tetromino(id, rotation, piece) AS (
  SELECT id, rotation, piece
  FROM const c(w), LATERAL (
  VALUES
    ...
    -- T
    (2, 0, ARRAY[3, 4, 5, (c.w+1) + 4]),
    (2, 1, ARRAY[-(c.w+1) + 4, 3, 4, (c.w+1) + 4]),
    (2, 2, ARRAY[-(c.w+1) + 4, 3, 4, 5]),
    (2, 3, ARRAY[-(c.w+1) + 4, 4, 5, (c.w+1) + 4]),
    ...
```

Excerpt of the “tetromino” table.

```
     [2, 0, 1, 0]
+--------------------+
|. . . . [][][]. . . |
|. . . . . []. . . . |
|. . . . . . . . . . |
```

“pos” example.

TETRIS IN A SQL QUERY

Game Logic

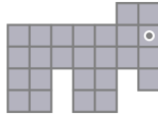
- Movement:
  - *pos* (int[]) column contains:
    - id of the current piece.
    - id of the current rotation.
    - number of cells moved based on the initial piece position.
    - if the piece moved down.

```
tetromino(id, rotation, piece) AS (
  SELECT id, rotation, piece
  FROM const c(w), LATERAL (
  VALUES
    ...
    -- T
    (2, 0, ARRAY[3, 4, 5, (c.w+1) + 4]),
    (2, 1, ARRAY[-(c.w+1) + 4, 3, 4, (c.w+1) + 4]),
    (2, 2, ARRAY[-(c.w+1) + 4, 3, 4, 5]),
    (2, 3, ARRAY[-(c.w+1) + 4, 4, 5, (c.w+1) + 4]),
    ...
```

Excerpt of the “tetromino” table.

```
    [2, 0, 12, 1]
+--------------------+
|. . . . . . . . . . |
|. . . . [][][]. . . |
|. . . . . []. . . . |
```

“pos” example.



TETRIS IN A SQL QUERY

Game Logic

- Movement:
  - *pos* (int[]) column contains:
    - id of the current piece.
    - id of the current rotation.
    - number of cells moved based on the initial piece position.
    - if the piece moved down.

```
tetromino(id, rotation, piece) AS (
  SELECT id, rotation, piece
  FROM const c(w), LATERAL (
  VALUES
    ...
    -- T
    (2, 0, ARRAY[3, 4, 5, (c.w+1) + 4]),
    (2, 1, ARRAY[-(c.w+1) + 4, 3, 4, (c.w+1) + 4]),
    (2, 2, ARRAY[-(c.w+1) + 4, 3, 4, 5]),
    (2, 3, ARRAY[-(c.w+1) + 4, 4, 5, (c.w+1) + 4]),
    ...
```

Excerpt of the “tetromino” table.

```
     [2, 1, 12, 0]
+--------------------+
|. . . . . []. . . . |
|. . . . [][]. . . . |
|. . . . . []. . . . |
```

“pos” example.

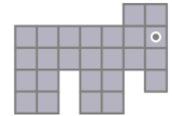
TETRIS IN A SQL QUERY

Game Logic

- Movement:
  - Natural drop – when we reach *last_drop_time + drop_delta*.
  - User movement:
    - left/right/down – update number of cells moved.
    - up – update piece rotation.
  - When natural drop / the user moves down, the *last_drop_time* is updated.

```
-- m = shorthand for main
CASE
  -- natural fall, increase the position by one line
  WHEN natural_fall THEN
    m.pos[:2] || ARRAY[m.pos[3] + const.width + 1] || 1
  -- user input
  WHEN input.ts > m.last_input_time THEN
    CASE
      WHEN input.cmd = 'u' THEN -- up
        m.pos[:1] || ARRAY[(m.pos[2] + 1) % 4] || m.pos[3:]
      WHEN input.cmd = 'd' THEN -- down
        m.pos[:2] || ARRAY[m.pos[3] + const.width + 1] || 1
      WHEN input.cmd = 'l' THEN -- left
        m.pos[:2] || ARRAY[m.pos[3] - 1] || m.pos[4]
      WHEN input.cmd = 'r' THEN -- right
        m.pos[:2] || ARRAY[m.pos[3] + 1] || m.pos[4]
      WHEN input.cmd = 's' THEN -- hard drop
        m.pos[:2] || ARRAY[m.pos[3] + m.max_drop_lines *
(const.width + 1)] || 1
    END
  -- nothing to do, position stays the same
  ELSE
    m.pos
END AS pos,
```

Excerpt of the movement subquery.



TETRIS IN A SQL QUERY

Game Logic

- Movement:
  - Collisions are determined by joining the current board with the new piece and checking if any of the matched board cells are filled.

```
collision(collides) AS (
  SELECT bool_or(cell) AS collides
  FROM unnest(main.board) WITH ORDINALITY b(cell, ordinality)
  JOIN unnest((SELECT new_piece FROM piece_after_movement)) p(coord)
    ON p.coord + 1 = b.ordinality
)
```

Collision detection subquery.

Game Logic

- Clearing lines: splitting the board into lines and removing the ones that are all filled.

```sql
new_board_compressed AS (
  -- aggregate back into a single array; count the number of remaining lines
  SELECT array_agg(cell ORDER BY line_number, col_number) AS board,
    (count(*) / (const.width + 1))::int AS num_lines
  FROM (
    -- filter out completed lines
    SELECT line_number, generate_series(0, const.width) AS col_number, unnest(line) AS cell
    FROM (
      -- split into one board line per row
      SELECT i AS line_number, board[i*(const.width + 1)+1:(i+1)*(const.width+1)] line
      FROM new_board, generate_series(0, const.height - 1) _(i)
    )
    -- filter out lines that have only true values
    WHERE NOT line <@ ARRAY[true]
  )
)
```

Board cleaning subquery.

TETRIS IN A SQL QUERY

Game Logic

- Max drop lines:
  - Used to implement the “ghost piece”, hard drops, and check for game over.
  - Implemented with a recursive CTE that drops the piece row by row until it collides.

```
t (lines) AS (
  SELECT -1
  UNION ALL
  SELECT lines + 1
  FROM t, curr_piece
  WHERE NOT ( -- drop until it collides
    SELECT bool_or(cell) OR bool_or(cell IS NULL)
    FROM unnest(piece) p(coord)
    LEFT JOIN unnest(next_board.board)
              WITH ORDINALITY b(cell, ordinality)
      ON (p.coord + movement.pos[3]) + 1
         + (lines + 1) * (const.width + 1)
         = b.ordinality
    WHERE (p.coord + movement.pos[3]) + 1
          + (lines + 1) * (const.width + 1)
          ⩾ 1
  )
)
SELECT max(lines) AS lines
FROM t
```

Excerpt of the drop_piece subquery.

TETRIS IN A SQL QUERY

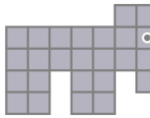
Game Logic

- Generating new pieces:
  - Done when the current piece is not longer playable.
  - Based on the NES Tetris random generator: [1]
    - Generate a random piece;
    - If the piece is equal to the previous one, generate one more time.

```
SELECT id
FROM (
  -- first piece roll
  SELECT id, 0 AS rank
  FROM (
    SELECT id
    FROM tetromino
    ORDER BY random() + main.frame -- add the frame to avoid caching
    LIMIT 1
  )
  WHERE id ≠ movement.pos[1] -- discard if equals to the previous
  UNION ALL
  ( -- second piece roll
    SELECT id, 1 AS rank
    FROM tetromino
    ORDER BY random() + main.frame -- add the frame to avoid caching
    LIMIT 1
  )
)
-- if we generated two valid pieces, select only the first one
ORDER BY rank
LIMIT 1
```

Excerpt of the next_piece subquery.

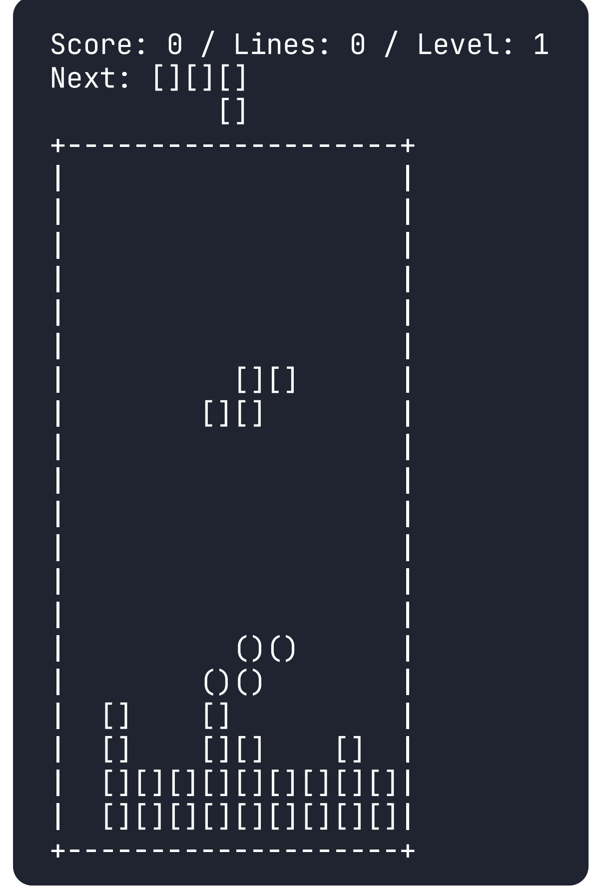
[1] https://tetris.wiki/Tetris_(NES)#Details



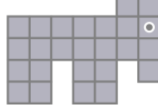
TETRIS IN A SQL QUERY

Game Logic

- Rendering: subquery that creates a string with the following info:
  - Score/lines/level.
  - Next piece.
  - Board (filled pieces rendered with “[]”; ghost piece with “()”; borders with “|”).
- “notify” function is then called with the final string.



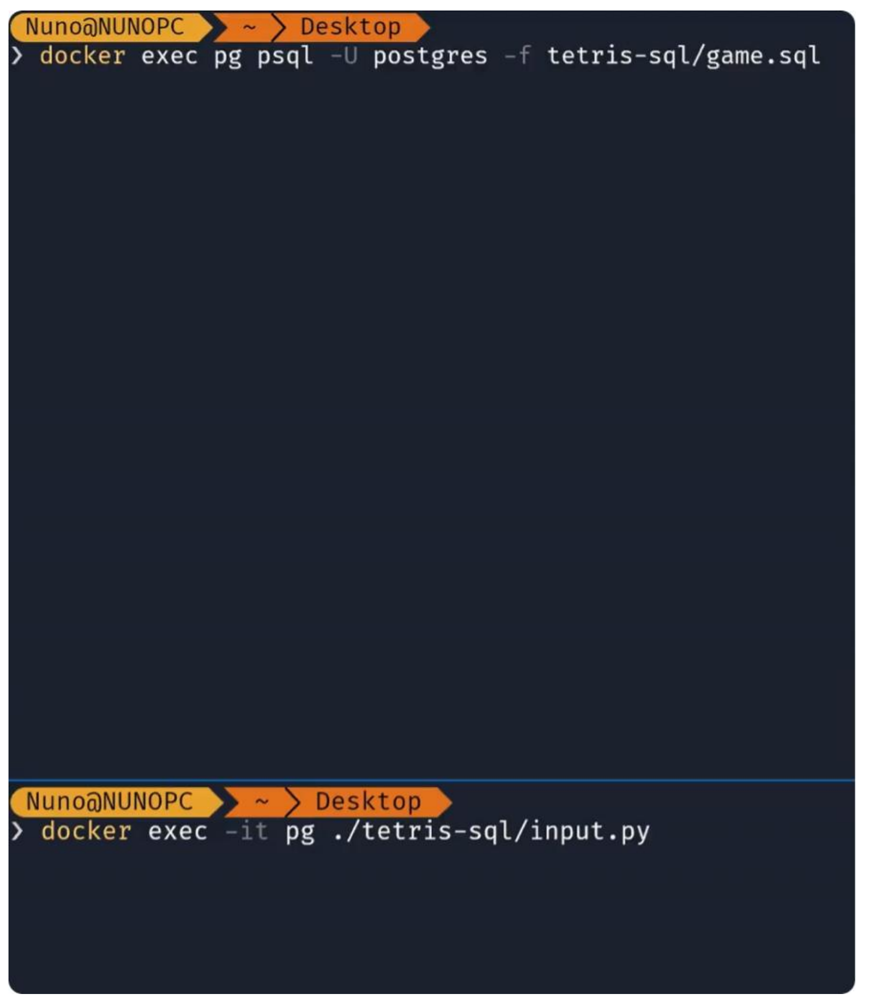
Game example.



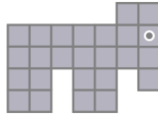
TETRIS IN A SQL QUERY

Demo

Code: /nuno-faria/tetris-sql



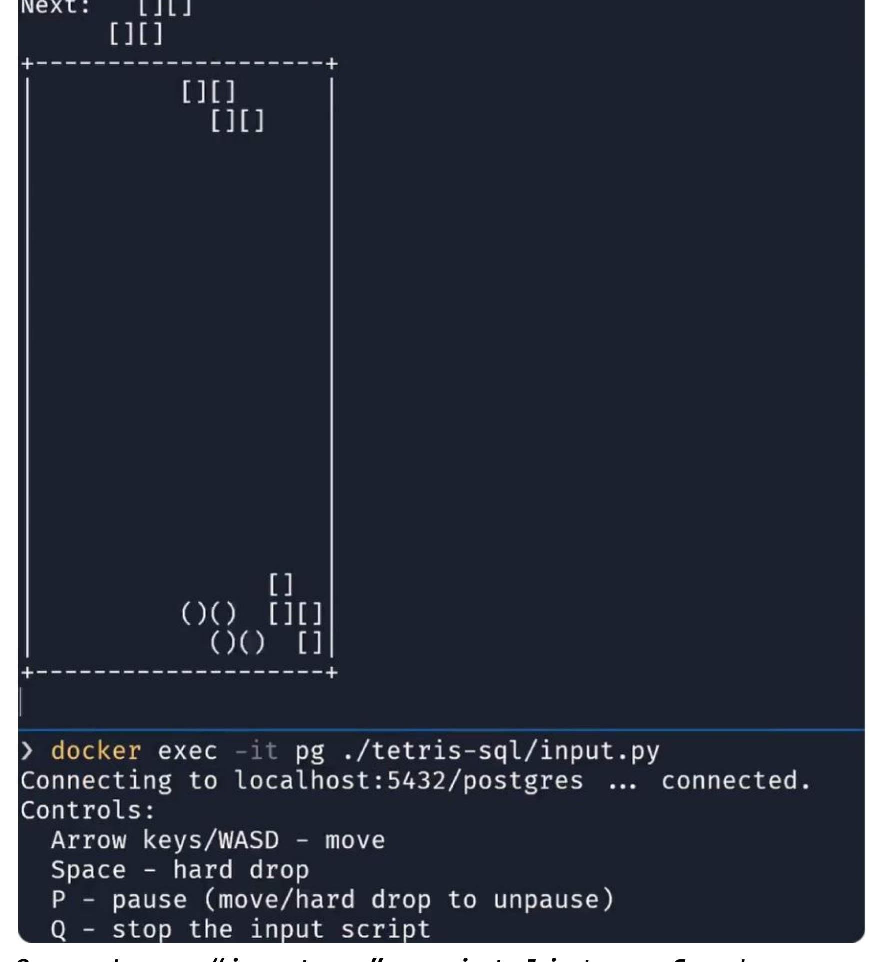
Game demo. "input.py" script listens for key stokes and updates the "Input" table.



TETRIS IN A SQL QUERY

Demo

Code: /nuno-faria/tetris-sql

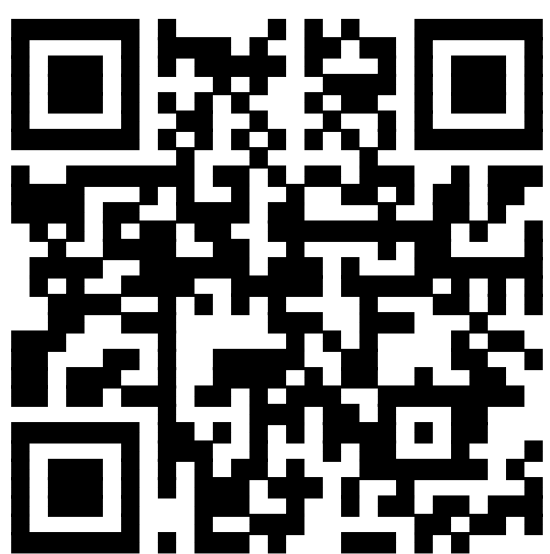


Game demo. "input.py" script listens for key stokes and updates the "Input" table.

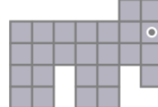
TETRIS IN A SQL QUERY

Demo

Code: /nuno-faria/tetris-sql



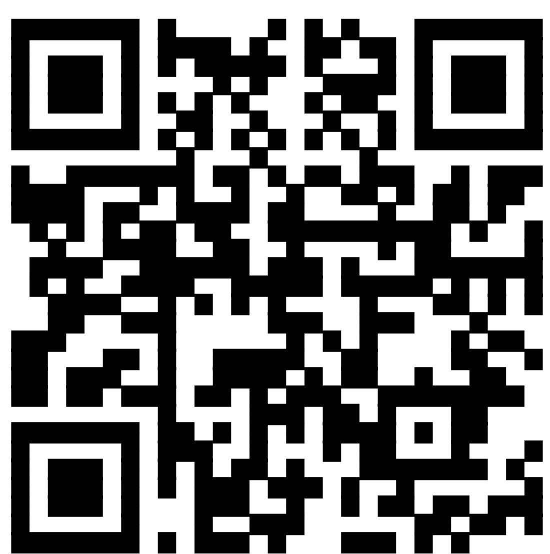
Game demo. "input.py" script listens for key stokes and updates the "Input" table.



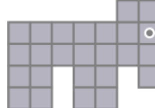
TETRIS IN A SQL QUERY

Demo

Code: /nuno-faria/tetris-sql



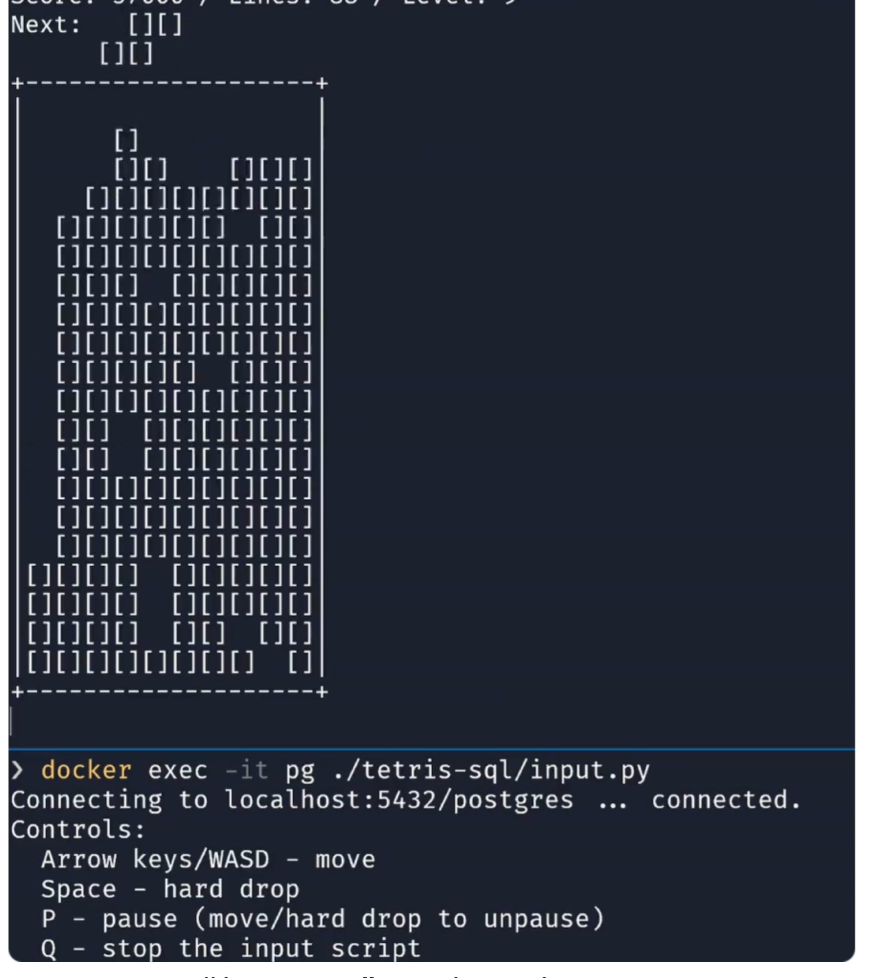
Game demo. "input.py" script listens for key stokes and updates the "Input" table.



TETRIS IN A SQL QUERY

Demo

Code: /nuno-faria/tetris-sql

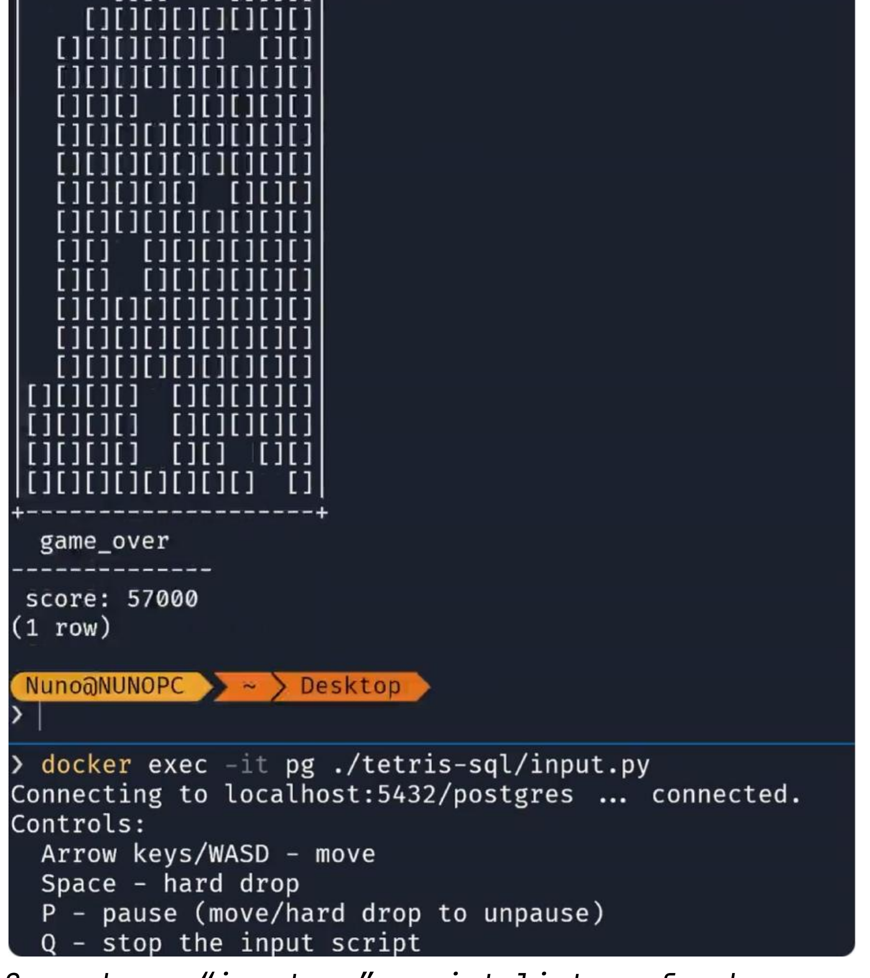


Game demo. "input.py" script listens for key stokes and updates the "Input" table.

TETRIS IN A SQL QUERY

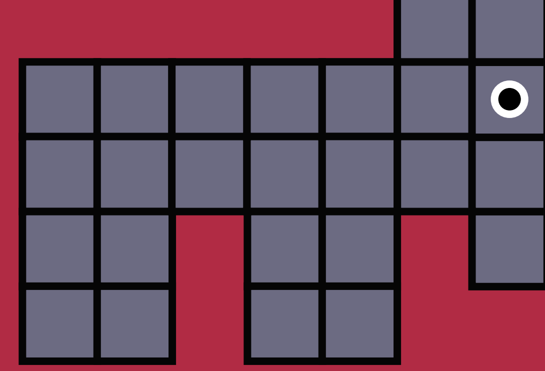
Demo

Code: /nuno-faria/tetris-sql



Game demo. "input.py" script listens for key stokes and updates the "Input" table.

BUILDING TETRIS IN A SQL QUERY!



Nuno Faria, INESC TEC

PostgreSQL Conference Europe 2025 – Riga, Latvia