KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Databázové systémy ◇ poznámky k přednášce

9. Plánovač

verze z 25. listopadu 2024

1 Plánování dotazu

Relační výrazy můžeme také nazývat **dotazy**. Při položení dotazu systém nejprve vytvoří **plán dotazu**, který určuje způsob výpočtu odpovědi na zadaný dotaz. Jeden dotaz může mít více plánů, kde každý spočítá odpověď na dotaz. Jednotlivé plány se mohou lišit v efektivitě výpočtu. Úkolem systému je zvolit plán, který bude nejefektivnější. Za tímto účelem systém postupně získává informace o uložených datech, které mu pomáhají v rozhodování. Může se tedy stát, že se plán dotazu s časem mění.

Plán dotazu *expression* lze získat příkazem:

```
EXPLAIN expression;
```

Předpokládejme, že máme základní relaci:

```
CREATE TABLE movie (
  movie_id text,
  title text,
  year integer,
  duration integer
);
```

s charakteristickou vlastností: „Film `movie_id` má název `title`, byl vydán v roce `year` a je dlouhý `duration` minut.“ Přitom naše databáze se omezuje jen na některé filmy.

Relace obsahuje kolem 200 tisíc n-tic. Jak ji vytvořit se dozvíte v přiloženém skriptu `01_imdb.sql`. Některé n-tice relace:

```
 movie_id  |                    title                    | year | duration
-----------+---------------------------------------------+------+----------
 tt0033290 | Young as You Feel                           | 1940 |       60
 tt0042035 | We Were Strangers                           | 1949 |      106
 tt4123430 | Fantastic Beasts: The Crimes of Grindelwald | 2018 |      134
 tt1945228 | Voyage of Time: Life's Journey              | 2015 |       90
 tt1395030 | Teahouse                                    | 1982 |      125
 tt0093987 | Slammer Girls                               | 1987 |       80
```

```
 tt0197614 | Landru, the Bluebeard of Paris              | 1923 |       81
 tt0189825 | Nudi per vivere                            | 1963 |       93
 tt0059833 | The Wheat and the Tares                    | 1965 |       90
 tt0363048 | The Longing                                | 2003 |       80
```

Na disku jsou n-tice relace uloženy ve **stránkách**. Velikost jedné stránky lze získat dotazem `SELECT current_setting('block_size')`. Velikost je uváděná v bajtech. Výchozí hodnota je 8 KB což je $8 \cdot 1024 = 8192$ bajtů.

```
# SELECT current_setting('block_size');

 current_setting
-----------------
 8192
(1 row)
```

Počet n-tic a stránek, které zabírá aktuální hodnota relační proměnné ***relation***, získáme dotazem:

```
SELECT reltuples AS tuples_count,
       relpages AS pages_count
FROM pg_class
WHERE relname = 'relation';
```

Například:

```
# SELECT reltuples AS tuples_count,
         relpages AS pages_count
  FROM pg_class
  WHERE relname = 'movie';

 tuples_count | pages_count
--------------+-------------
       201921 |        1736
(1 row)
```

Získejme plán dotazu:

```
# EXPLAIN
  SELECT *
  FROM   movie;
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..3540.04 rows=192504 width=36)
(1 row)
```

Plán dotazu je strom, kde každý uzel odpovídá části výpočtu. Zde má strom jediný uzel:

```
Seq Scan on movie  (cost=0.00..3540.04 rows=192504 width=36)
```

Uzly stromu jsou určitého typu. Náš jediný uzel je typu `Seq Scan`. Uzly tohoto typu odpovídají za postupné procházení n-tic relace na disku. Jedná se o nejzákladnější způsob zpracování dat. Data jsou po stránkách načítána z disku a ze stránek se získávají n-tice relace.

Za typem uzlu vidíme upřesnění jeho použití. Část `on movie` specifikuje, že se jedná o postupné čtení n-tic základní relace `movie`.

Informace uzlu v závorkách udávají odhady náročnosti výpočtu, kde `cost` je odhad času do získání první hodnoty a odhad celkového času výpočtu uzlu, `rows` je odhad počtu n-tic a `width` je odhad délky jedné n-tice v bajtech.

Čas se měří v abstraktních jednotkách a zahrnuje počet přečtených stránek z disku S a počet zpracovaných n-tic R:

$$S + R \cdot 0.01$$

Přidáním klauzule ANALYZE se vypočte hodnota výrazu a příkaz zobrazí také naměřené hodnoty:

```
# EXPLAIN ANALYZE
  SELECT *
  FROM   movie;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..3540.04 rows=192504 width=36)
                     (actual time=0.005..14.492 rows=192504 loops=1)
 Planning Time: 0.023 ms
 Execution Time: 24.020 ms
```

Čas naměřených hodnot (`actual time`) se měří v milisekundách.

Uzly v plánu dotazu mohou mít **atributy** ovlivňující jeho činnost. Plán dotazu

```
# EXPLAIN
  SELECT *
  FROM   movie
  WHERE  year >= 2000;

                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..4021.30 rows=66234 width=36)
   Filter: (year >= 2000)
(2 rows)
```

se stále skládá z jednoho uzlu, ale ten má atribut `Filter`, který způsobí, že uzlem přečtené n-tice se filtrují. V závorkách za atributem je uvedena podmínka filtrování `year >= 2000`.

Opět si můžeme nechat zaznamenat informace o výpočtu:

```
# EXPLAIN ANALYZE
  SELECT *
  FROM   movie
  WHERE  year >= 2000;

                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..4021.30 rows=66234 width=36)
                     (actual time=0.007..17.938 rows=66236 loops=1)
   Filter: (year >= 2000)
   Rows Removed by Filter: 126268
 Planning Time: 0.031 ms
 Execution Time: 21.297 ms
(5 rows)
```

Vidíme i kolik filtrování odstranilo n-tic.

Plán dotazu:

```
# EXPLAIN
  SELECT   title
  FROM     movie
  ORDER BY year;

                               QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=24384.63..24865.89 rows=192504 width=22)
   Sort Key: year
   ->  Seq Scan on movie  (cost=0.00..3540.04 rows=192504 width=22)
(3 rows)
```

je tvořen dvěma uzly. Kořen je typu `Sort` a jeho následník typu `Seq Scan`. Šipka (`->`) zachycuje následníky uzlu. Uzel typu `Sort` provádí řazení n-tic, které dostane od svého následníka. Atribut `Sort Key` udává atribut, podle kterého se n-tic budou řadit.

U informací o aktuálním zpracování:

```
# EXPLAIN ANALYZE
  SELECT   title
  FROM     movie
  ORDER BY year;
```

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=24384.63..24865.89 rows=192504 width=22)
       (actual time=79.242..99.563 rows=192504 loops=1)
   Sort Key: year
   Sort Method: external merge  Disk: 6456kB
   ->  Seq Scan on movie  (cost=0.00..3540.04 rows=192504 width=22)
                           (actual time=0.006..21.898 rows=192504 loops=1)
 Planning Time: 0.028 ms
 Execution Time: 117.089 ms
(6 rows)
```

vidíme, že se třídění uskutečnilo vnějším tříděním za použitím disku.

PostgreSQL si udržuje statistiky o uložených datech. Jednou ze statistik je histogram hodnot v určitém atributu relační proměnné. To je rostoucí posloupnost hodnot domény atributu taková, že sousední prvky posloupnosti chápeme jako intervaly dělící aktuální hodnoty v atributu do přibližně stejně početných skupin. Histogram hodnot v atributu *attribute* proměnné *relation* lze získat dotazem:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename = 'relation' AND attname = 'attribute'
```

Například:

```
# SELECT histogram_bounds FROM pg_stats
  WHERE tablename = 'movie' AND attname = 'year';

                            histogram_bounds
-------------------------------------------------------------------------
 {1894,1915,1921,1921,1922,1922,1923,1924,
 1924,1925,1925,1928,1929,1929,1945,1945,2025}
(1 row)
```

Z histogramu atributu proměnné lze odhadnout počet n-tic po restrikci s nerovností. Proto v následujícím dotazu lze použít vnitřní řazení pomocí `quicksort`.

```
# EXPLAIN ANALYZE
  SELECT   title
  FROM     movie
  WHERE    year >= 2010
  ORDER BY year;

                            QUERY PLAN
-------------------------------------------------------------------
```

```
 Sort  (cost=7844.07..7959.71 rows=46254 width=22)
         (actual time=45.691..50.958 rows=46057 loops=1)
   Sort Key: year
   Sort Method: quicksort  Memory: 3382kB
   ->  Seq Scan on movie
           (cost=0.00..4260.01 rows=46254 width=22)
           (actual time=0.030..29.858 rows=46057 loops=1)
         Filter: (year >= 2010)
         Rows Removed by Filter: 155864
 Planning Time: 0.440 ms
 Execution Time: 56.322 ms
(8 rows)
```

Zde:

```
# EXPLAIN
  SELECT *
  FROM   movie
  WHERE  year >= 2000
  LIMIT 10;

                              QUERY PLAN
-----------------------------------------------------------------------
 Limit  (cost=0.00..0.36 rows=10 width=37)
   ->  Seq Scan on movie  (cost=0.00..7942.55 rows=218439 width=37)
         Filter: (year >= 2000)
(3 rows)
```

Přibyl nový typ uzlu `Limit`, který dokáže zastavit vykonávání svého následníka.

U dotazu:

```
# EXPLAIN
  SELECT   title
  FROM     movie
  ORDER BY year
  LIMIT    10;

                              QUERY PLAN
-----------------------------------------------------------------------
 Limit  (cost=9196.83..9197.99 rows=10 width=23)
   ->  Gather Merge  (cost=9196.83..46054.87 rows=315904 width=23)
         Workers Planned: 2
         ->  Sort  (cost=8196.80..8591.68 rows=157952 width=23)
               Sort Key: year
               ->  Parallel Seq Scan on movie
                   (cost=0.00..4783.52 rows=157952 width=23)
(6 rows)
```

je v plánu paralelní zpracování, které poznáme použitím prefixu `Parallel` u typu uzlu (zde `Parallel Seq Scan`). Uzel `Gather Merge` bude skládat výsledky paralelního zpracování.

Průběh:

```
# EXPLAIN ANALYZE
   SELECT   title
   FROM     movie
   ORDER BY year
   LIMIT    10;

                                QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=9196.83..9197.99 rows=10 width=23)
       (actual time=68.415..70.527 rows=10 loops=1)
   ->  Gather Merge  (cost=9196.83..46054.87 rows=315904 width=23)
                     (actual time=68.413..70.522 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=8196.80..8591.68 rows=157952 width=23)
                   (actual time=62.335..62.337 rows=8 loops=3)
               Sort Key: year
               Sort Method: top-N heapsort  Memory: 26kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 26kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 26kB
               ->  Parallel Seq Scan on movie
                     (cost=0.00..4783.52 rows=157952 width=23)
                     (actual time=0.019..33.080 rows=126361 loops=3)
 Planning Time: 0.101 ms
 Execution Time: 70.561 ms
(12 rows)
```

Uzel typu `HashAggregate` řídí vytváření skupin. Například:

```
# EXPLAIN
  SELECT   count(*),
           title
  FROM     movie
  GROUP BY title;

                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=34241.50..41620.92 rows=293703 width=27)
   Group Key: title
   Planned Partitions: 8
   ->  Seq Scan on movie  (cost=0.00..6994.84 rows=379084 width=19)
(4 rows)
```

Atribut uzlu `Group Key` udává atributy relace, podle kterých se skupiny vytvářejí.

Průběh výpočtu:

```
# EXPLAIN  ANALYZE
  SELECT   count(*),
           title
  FROM     movie
  GROUP BY title;

                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=34241.50..41620.92 rows=293703 width=27)
                (actual time=259.660..486.240 rows=348099 loops=1)
   Group Key: title
   Planned Partitions: 8  Batches: 41
                          Memory Usage: 4177kB
                          Disk Usage: 15552kB
   ->  Seq Scan on movie  (cost=0.00..6994.84 rows=379084 width=19)
                           (actual time=0.016..43.603 rows=379084 loops=1)
 Planning Time: 0.117 ms
 Execution Time: 516.692 ms
(6 rows)
```

Zajímavé je, že i odstranění duplicit se děje pomocí seskupování:

```
# EXPLAIN
  SELECT DISTINCT *
  FROM   movie;

                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=42060.11..51774.14 rows=379084 width=37)
   Group Key: movie_id, title, year, duration
   Planned Partitions: 16
   ->  Seq Scan on movie  (cost=0.00..6994.84 rows=379084 width=37)
(4 rows)
```

Přidáme další základní relaci:

```
CREATE TABLE director (
  movie_id text,
  person_id text
);
```

danou vlastností: „Osoba `person_id` režírovala film `movie_id`“. Ukládáme jen některé režiséry. Relace obsahuje přes 200 tisíc n-tic.

Některé n-tice relace:

```
 movie_id  | person_id
-----------+-----------
 tt0121171 | nm0812150
 tt4169390 | nm0963103
 tt0101404 | nm0001003
 tt0245096 | nm0683248
 tt7587280 | nm0271792
 tt0046497 | nm0792450
 tt0065073 | nm0765856
 tt6217564 | nm2116358
 tt0874963 | nm0529616
 tt0083763 | nm0071157
(10 rows)
```

Prohlídneme si plán dotazu:

```
# EXPLAIN
  SELECT *
  FROM   movie AS m,
         director AS d
  WHERE  m.movie_id = d.movie_id;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=14695.39..26226.30 rows=204796 width=57)
   Hash Cond: (d.movie_id = m.movie_id)
   ->  Seq Scan on director d  (cost=0.00..3352.96 rows=204796 width=20)
   ->  Hash  (cost=6994.84..6994.84 rows=379084 width=37)
         ->  Seq Scan on movie m  (cost=0.00..6994.84 rows=379084 width=37)
(5 rows)
```

Uzel `Hash` vytváří hešovací tabulku, která umožňuje efektivně zjišťovat, zda obsahuje nějaký prvek. Uzel `Hash Join` má dva následníky a spojuje n-tice, které produkují. Atribut `Hash Cond` udává podmínku na spojování.

Plán dotazu:

```
# EXPLAIN
  SELECT m.*, d.person_id
  FROM   movie AS m,
         director AS d
  WHERE  m.movie_id = d.movie_id
  AND    m.movie_id = 'tt0021814';

                              QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.00..7891.59 rows=1 width=46)
```

```
   ->  Seq Scan on director d  (cost=0.00..3870.28 rows=1 width=20)
         Filter: (movie_id = 'tt0021814'::text)
   ->  Seq Scan on movie m  (cost=0.00..4021.30 rows=1 width=36)
         Filter: (movie_id = 'tt0021814'::text)
(5 rows)
```

Má v kořeni uzel `Nested Loop`, který prochází každou n-tici obdrženou od prvního následníka a snaží se jej spojit s každou n-ticí od druhého následníka.

2 Indexy

Vraťme se k základní relaci `movie` z minulé sekce.

Odpověď na dotaz

```
# EXPLAIN
  SELECT title
  FROM   movie
  WHERE  year = 1975;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..6440.00 rows=2616 width=19)
   Workers Planned: 2
   ->  Parallel Seq Scan on movie  (cost=0.00..5178.40 rows=1090 width=19)
         Filter: (year = 1975)
(4 rows)
```

zahrnuje průchod každým řádkem relace `movie`.

Pro efektivnější práci s vybranými atributy relace můžeme příkazem:

```
CREATE INDEX name ON relation ( attributes );
```

vytvořit index. Základní typ indexu vytváří a udržuje B-strom hodnot ze zadaných atributů. Konkrétně se jedná o B+ strom, který na rozdíl od B-stromu udržuje data pouze v listech a navíc každé jeho patro tvoří obousměrný spojový seznam. Každý uzel stromu je uložen v samostatné stránce.

Například:

```
CREATE INDEX movie_year_idx ON movie ( year );
```

Každý vytvořený index na základní relaci zpomaluje manipulaci s řádky (přidávání, odebírání a změnu), ale může být použit k zrychlení najití odpovědi na dotaz. Počet n-tic a stránek, které index zabírá, zjistíme stejně jako u proměnné:

```
# SELECT reltuples AS tuples_count,
          relpages AS pages_count
   FROM pg_class
   WHERE relname = 'movie_year_idx';

 tuples_count | pages_count
--------------+-------------
       201921 |         178
(1 row)
```

Například nyní se použije právě vytvořený index:

```
# EXPLAIN
  SELECT title
  FROM   movie
  WHERE  year = 1975;

                              QUERY PLAN
---------------------------------------------------------------------
 Bitmap Heap Scan on movie
         (cost=32.70..3252.73 rows=2616 width=19)
   Recheck Cond: (year = 1975)
   ->  Bitmap Index Scan on movie_year_idx
           (cost=0.00..32.04 rows=2616 width=0)
         Index Cond: (year = 1975)
(4 rows)
```

Uzel `Bitmap Index Scan` vytvoří bitovou mapu, kde bit odpovídá stránkám proměnné. Poté prochází index a v bitové mapě označuje ty stránky, které obsahují n-tici splňující podmínku `Index Cond`. Uzel `Bitmap Heap Scan` poté z bitové mapy načte označené stránky a vyfiltruje z nich hledané n-tice.

Index lze použít i pro nerovnost v podmínce:

```
# EXPLAIN
  SELECT title
  FROM   movie
  WHERE  year > 2020;

                              QUERY PLAN
---------------------------------------------------------------------
 Bitmap Heap Scan on movie
         (cost=238.65..3704.32 rows=20933 width=19)
   Recheck Cond: (year > 2020)
   ->  Bitmap Index Scan on movie_year_idx
           (cost=0.00..233.42 rows=20933 width=0)
         Index Cond: (year > 2020)
(4 rows)
```

Část systému, která je zodpovědná za vytváření plánu dotazu, se nazývá **plánovač**. Plánovač se může rozhodnout index nepoužít a to například v případě, kdy očekává velké množství n-tic, které by podmínku splňovalo:

```
# EXPLAIN
  SELECT title
  FROM   movie
  WHERE  year > 1950;

                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..7942.55 rows=343392 width=19)
   Filter: (year > 1950)
(2 rows)
```

Index lze použít přímo na získání n-tic uspořádaných podle indexovaného atributu:

```
# EXPLAIN
  SELECT   title
  FROM     movie
  ORDER BY year;

                            QUERY PLAN
-------------------------------------------------------------------
 Index Scan using movie_year_idx on movie
     (cost=0.42..19807.23 rows=379084 width=23)
(1 row)
```

Uzel `Index Scan` prochází index a načítá n-tice z disku v požadovaném pořadí. U uzlu je uvedený index, který uzel používá.

Použitím `UNIQUE` vytvoříme index, který nedovolí duplicity. Například:

```
CREATE UNIQUE INDEX movie_movie_id_idx ON movie ( movie_id );
```

Vyhledávání na rovnost:

```
# EXPLAIN
  SELECT title
  FROM   movie
  WHERE  movie_id = 'tt0087182';

                            QUERY PLAN
-------------------------------------------------------------------
 Index Scan using movie_movie_id_idx on movie
       (cost=0.42..8.44 rows=1 width=19)
   Index Cond: (movie_id = 'tt0087182'::text)
(2 rows)
```

použije uzel `Index Scan` s atributem `Index Cond`, který najde a přečte z disku případnou jedinou n-tici.

U dotazu:

```
# EXPLAIN
  SELECT title
  FROM   movie
  WHERE  year = 1962
  AND    title = 'Lolita';

                            QUERY PLAN
-------------------------------------------------------------------
 Bitmap Heap Scan on movie
        (cost=23.59..3009.16 rows=1 width=19)
   Recheck Cond: (year = 1962)
   Filter: (title = 'Lolita'::text)
   ->  Bitmap Index Scan on movie_year_idx
             (cost=0.00..23.59 rows=2022 width=0)
         Index Cond: (year = 1962)
(5 rows)
```

nejprve uzel `Bitmap Index Scan` nalezne pomocí indexu stránky obsahující n-tice splňující `year = 1962` a poté uzel `Bitmap Heap Scan` ze stránek vyzvedne n-tice splňující `year = 1962` a provede filtr na podmínku `title = 'Lolita'`.

Vytvořením indexu na oba atributy:

```
CREATE INDEX movie_title_year_idx ON movie ( title, year );
```

lze dotaz zodpovědět použitím indexu:

```
# EXPLAIN
  SELECT *
  FROM   movie
  WHERE  year = 1962
  AND    title = 'Lolita';

                            QUERY PLAN
-------------------------------------------------------------------
 Index Scan using movie_title_year_idx on movie
       (cost=0.42..8.44 rows=1 width=36)
   Index Cond: ((title = 'Lolita'::text) AND (year = 1962))
(2 rows)
```

Následující dotaz lze zodpovědět pouze použitím indexu bez potřeby čtení n-tic ze stránek:

```
# EXPLAIN
  SELECT year
  FROM   movie
  WHERE  title = 'Lolita';

                            QUERY PLAN
-------------------------------------------------------------------
 Index Only Scan using movie_title_year_idx on movie
     (cost=0.42..8.44 rows=1 width=4)
   Index Cond: (title = 'Lolita'::text)
(2 rows)
```

Uzel `Index Only Scan` tedy pouze čte index.

Vytvoříme si index na atribut `person_id` proměnné `director`:

```
CREATE INDEX director_person_id_idx
ON     director ( person_id );
```

Při spojování se použijí indexy na spojovaných proměnných:

```
# EXPLAIN
  SELECT m.title
  FROM   movie AS m,
         director AS d
  WHERE  m.movie_id = d.movie_id
  AND    d.person_id = 'nm0000040';

                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=4.91..114.37 rows=9 width=18)
   ->  Bitmap Heap Scan on director d  (cost=4.49..38.43 rows=9 width=10)
         Recheck Cond: (person_id = 'nm0000040'::text)
         ->  Bitmap Index Scan on director_person_id_idx
                 (cost=0.00..4.49 rows=9 width=0)
               Index Cond: (person_id = 'nm0000040'::text)
   ->  Index Scan using movie_movie_id_idx on movie m
          (cost=0.42..8.44 rows=1 width=28)
         Index Cond: (movie_id = d.movie_id)
(7 rows)
```

Deklarováním UNIQUE nebo PRIMARY KEY omezení u základní relace se vytvoří index na vyjmenovaných atributech, který nebude dovolovat duplicity. Například deklarace:

```
CREATE TABLE movie (
  movie_id text PRIMARY KEY,
```

```
  title text,
  year integer,
  duration text
);
```

vytvoří index na atributu `movie_id`.

3 Kategorie uzlů v plánu

Uzly v plánu dotazu se dělí do tří kategorií:

1. načítací uzly (`Seq Scan`, `Bitmap Index Scan`, `Index Scan`, `Bitmap Heap Scan`, ...),
2. výpočetní uzly (`Sort`, `Limit`, `Gather Merge`, `HashAggregate`, `Hash`, ...),
3. spojovací uzly (`Hash Join`, `Nested Loop`, ...).

Načítací uzly se nacházejí v listech stromu plánu dotazu. Jsou zodpovědné za čtení n-tic relací, které probíhá většinou z disku a případně za použití indexu. Výpočetní uzly mají většinou jediného následníka, z kterého získávají n-tice a dále je zpracovávají. Konečně spojovací uzly mají právě dva následníky a spojují n-tice z nich získané.

Otázky a úkoly na cvičení

1. Vezměme relační proměnnou:

   ```
   CREATE TABLE person (
     person_id text,
     name text,
     birth int
   );
   ```

   jejíž hodnota má velikost 43378. Proměnná je vytvořená podle skriptu `01_imdb.sql`. Určete plány následujících dotazů a pak své plány porovnejte s těmi, které vytvořil PostgreSQL.

   (a)
   ```
   SELECT name
   FROM   person;
   ```

   (b)
   ```
   SELECT name
   FROM   person
   WHERE  birth = 1993;
   ```

(c)
```
SELECT   name
FROM     person
WHERE    birth = 1993
ORDER BY name;
```

(d)
```
SELECT count(*) AS count, birth
FROM   person
GROUP BY birth;
```

(e)
```
SELECT DISTINCT birth
FROM   person;
```

2. Přidejme si index na atribut `birth` proměnné `person`. Atribut má histogram:

```
{1853,1862,1866,1869,1870,1872,1873,1875,1876,1877,1877,1878,1879,
1880,1880,1881,1882,1883,1883,1884,1884,1885,1885,1886,1886,1887,
1887,1888,1888,1888,1889,1889,1890,1890,1890,1891,1891,1896,1896,
1896,1993,1993,1993,1994,1994,1995,1995,1995,1996,1996,1997,1998,
1998,1999,2001,2003,2023}
```

Jaké budou plány následujících dotazů? Jakou očekáváme velikost odpovědí?

(a)
```
SELECT DISTINCT birth
FROM   person
WHERE  birth > 2000;
```

(b)
```
SELECT name
FROM   person
WHERE  birth > 2000;
```

(c)
```
SELECT *
FROM   person
WHERE  birth > 1800
AND    name = 'David Lynch';
```

(d)
```
SELECT   *
FROM     person
WHERE.   birth > 1900
AND      birth < 2000
ORDER BY birth;
```

3. Přidejme proměnnou:

```
CREATE TABLE director (
  movie_id text,
  person_id text
);
```

s hodnotou definovanou také ve skriptu `01_imdb.sql`. Aktuální hodnota má velikost 201921. Opět určete plány následujících dotazů.

(a)
```
SELECT director.movie_id, person.name
FROM   director, person
WHERE  director.person_id = person.person_id;
```

(b)
```
SELECT director.movie_id
FROM   director, person
WHERE  director.person_id = person.person_id
AND    person.name = 'David Lynch';
```

(c)
```
SELECT director.movie_id
FROM   director, person
WHERE  director.person_id = person.person_id
AND    person.birth = 1990;
```

4. Přidejme indexy na atributy `person_id` u proměnných `director` a `person`. Určete plány následujících dotazů.

(a)
```
SELECT director.movie_id
FROM   director, person
WHERE  director.person_id = person.person_id
AND    person.birth = 1900;
```

(b)
```
EXPLAIN
SELECT director.movie_id
FROM   director, person
WHERE  director.person_id = person.person_id
AND    person.name = 'David Lynch';
```

(c)
```
SELECT director.movie_id, person.name
FROM   director, person
WHERE  director.person_id = person.person_id;
```