Úvod do databázových systémov

http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/DB1

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2026–2027

Trvácne médiá

- Dáta sú uložené na **trvácnych (externých) médiách**, ktoré možno rozdeliť do dvoch kategórií:
  - médiá **so sekvenčným prístupom** (napr. magnetické pásky), práca s nimi pripomína prácu s linkovaným zoznamom
  - médiá **s ľubovoľným prístupom** (napr. magnetické disky), práca s nimi pripomína prácu s poľom
- Dáta v tabuľkách sú organizované po riadkoch (records), dáta na externých médiách sú organizované **v blokoch**
  - predpokladá sa, že **jeden blok obsahuje viacero záznamov**
  - pre jednoduchosť predpokladáme, že dáta v operačnej pamäti (RAM) sú organizované po rovnakých blokoch
  - drivery médií abstrahujú od konkrétnych fyzických detailov (napr. cylindre, sektory, stopy), ponúkajú adresáciu blokov a čítanie/písanie bloku z/do operačnej pamäte (operácie input/output v Cache Manager)

Fyzická optimalizácia dotazov: fyzická algebra

Literatúra:

• H. Garcia-Molina, J.D. Ullman, J. Widom: Database System Implementation, Prentice Hall, 2000 (Chapters 6–7)

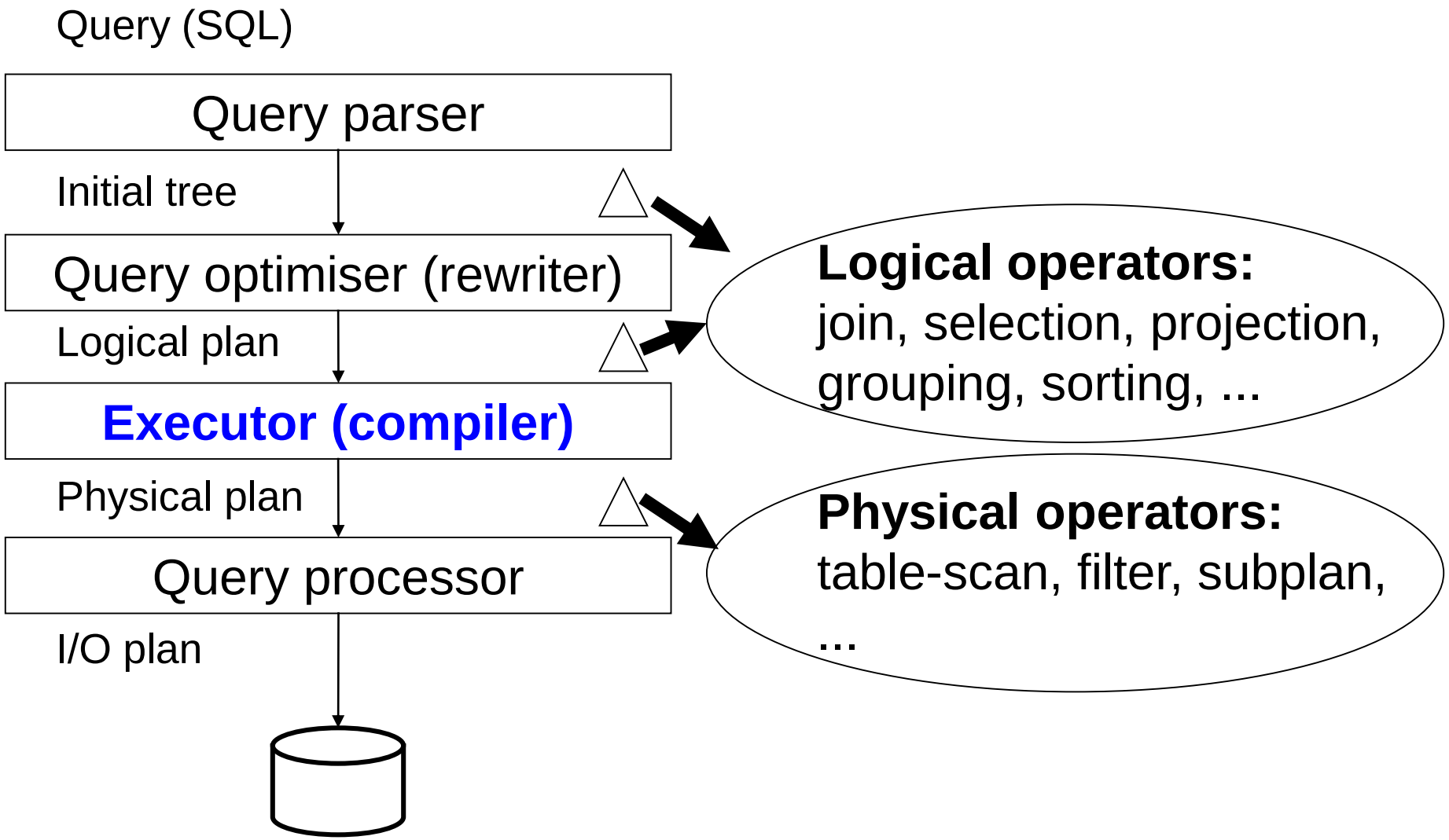
• R. Treat: Explaining Explain

• Postgres manual, http://www.postgresql.org/docs/current/static/performance-tips.html

• Y.E. Ioannidis: Query Optimization, http://www-db.stanford.edu/~widom/cs346/ioannidis.pdf

• S. Chaudhuri: An Overview of Query Optimization in Relational Systems, http://www-db.stanford.edu/~widom/cs346/chaudhuri.pdf

Fyzická optimalizácia dotazov: fyzická algebra



Fyzická optimalizácia dotazov: fyzická algebra

Možnosti implementácie fyzických operátorov:

- Materializácia: medzivýsledky operácií sa reprezentujú tabuľkou. Nevýhody: medzivýsledky môžu byť obrovské, medzivýsledky treba ukladať a čítať viackrát ako raz, ...

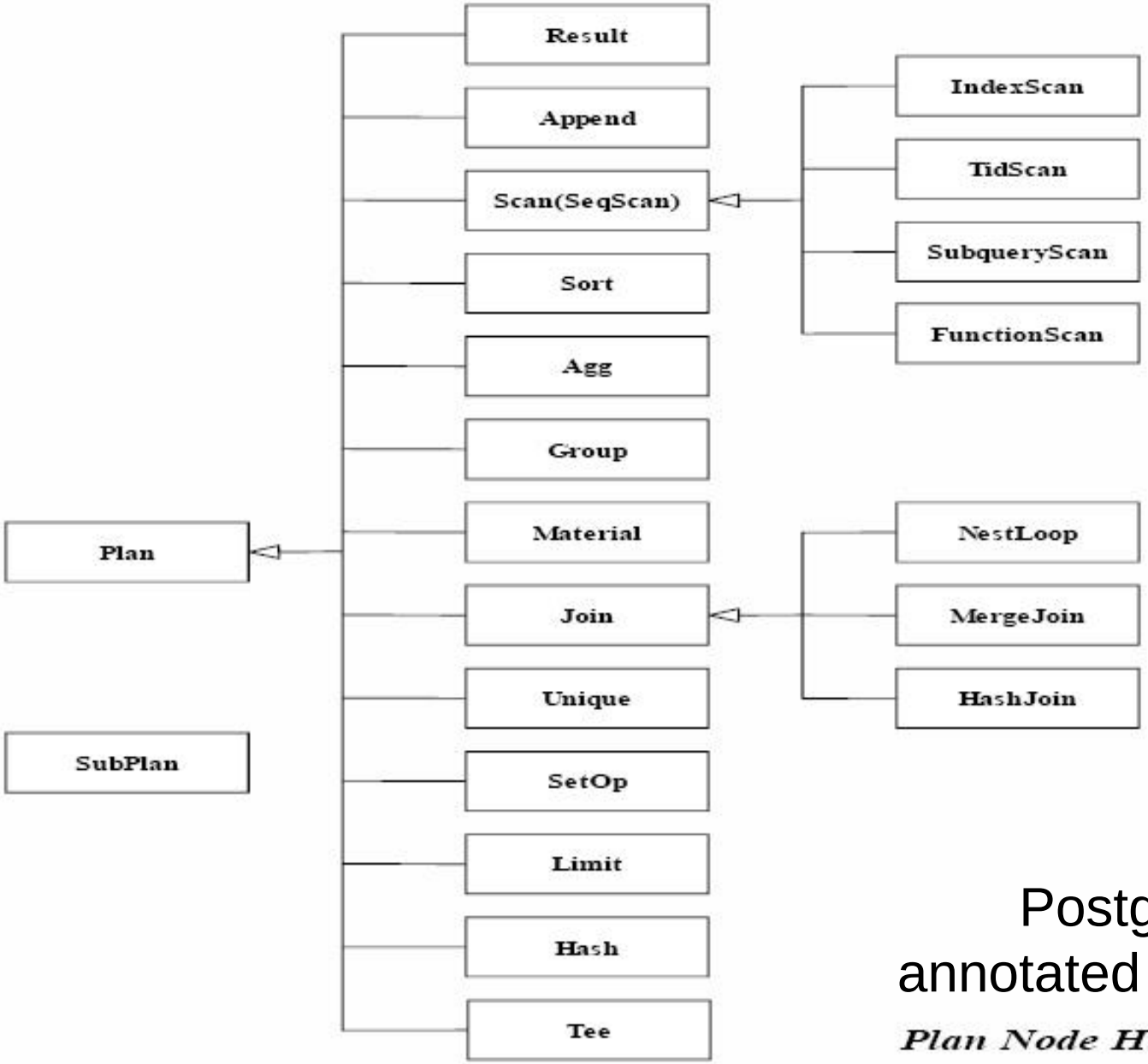
- **Iterátory (pipelining):** medzivýsledky sa reprezentujú vhodným previazaním pointerov na záznamy (riadky relácií). V jednej iterácii (next) sa vypočíta jeden riadok výslednej relácie, výsledok iterátora môže byť vstupom do ďalšieho iterátora

Metódy iterátorov: **open**, **next**, **close** (objektová filozofia)

Postgres, ako aj prakticky všetky súčasné DB systémy, používa iterátory na implementáciu fyzických operátorov. Materializácia je implementovaná ako špeciálny iterátor

Cieľom návrhu a implementácie fyzických operátorov je minimalizovať počet diskových prenosov, t.j. počet input/output operácií

Fyzická optimalizácia dotazov: fyzická algebra



PostgreSQL,
annotated source code

Plan Node Hierarchy Chart

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Miery zložitosti relácií (relácia je „bag“, nie set)
B(R): počet blokov R
T(R): počet tuples (záznamov) R, vrátane duplikátov
V(R, a_1, a_2, ..., a_N): počet rôznych tuples $\pi_{a_1, a_2, ..., a_N}(R)$

Miery zložitosti fyzických operátorov
počet I/O prenosov, **čas**

POSTGRESQL Cost Parameters

Parameter	Description	Default	vs. page read
page read	Cost to read fetch one page of data, by definition	1.00	-
cpu_tuple_cost	Cost of typical CPU time to process a tuple	0.01	100x quicker
cpu_index_tuple_cost	Cost of typical CPU time to process an index tuple	0.001	1000x quicker
cpu_operator_cost	Cost of CPU time to process a typical WHERE operator	0.0025	400x quicker
random_page_cost	Cost of a non-sequential page fetch	4	4x slower
effective_cache_size	Amount of cache = likelihood of finding a page in cache	1000	N/A

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Iterator **SeqScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.next();
}
```

Seq Scan Operator : Example

```
oscon=# explain select oid from pg_proc;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_proc  (cost=0.00..87.47
rows=1747 width=4)
```

- Most basic, simply scans from start to end
- Checks each row for any conditionals as it goes
- Large tables prefer index scans
- Cost (no startup cost), rows (tuples), width (oid)

Miera zložitosti fyzických operátorov: čas vykonávania

V konfigurácii Postgres je konštanta 0.01, ktorá hovorí že RAM je 100x rýchlejšia ako disk. Postgres meria zložitosť v počte I/O prenosov, ale zahŕňa do nej aj čas výpočtov v RAM. V tomto príklade B(R)=70 a T(R)=1747, tak odhad zložitosti seq_scan(R) je 70+1747*0.01=87.47 I/O prenosov. (To 0.00 je odhad „startup cost“, t.j. zložitosť „open“. V tomto prípade nestojí startup nič.)

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Iterator **SeqScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.next().filter();
}
```

```
oscon=# explain select oid from pg_proc where oid<>0;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_proc  (cost=0.00..91.84 rows=1746 width=4)
   Filter: (oid <> 0::oid)
```

IOcost+CPUcost=B(R)+T(R)*(0.01+0.0025)=91.835
(0.0025 je cena filtrovacej podmienky)

```
oscon=# explain select oid from pg_proc where oid<>0
and oid<100;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_proc  (cost=0.00..96.20 rows=582
width=4)
   Filter: ((oid <> 0::oid) AND (oid < 100::oid))
```

IOcost+CPUcost=B(R)+T(R)*(0.01+2*0.0025)=96.205
(cenu za testovanie podmienky platíme dvakrát)

Vybrané fyzické operátory: merge-sort

Algoritmus sort (externý merge-sort)

Triedenie dlhého súboru, ktorý sa nezmestí celý do operačnej pamäte. Nech je v operačnej pamäti miesto pre M diskových blokov

1.fáza, vytvorenie utriedených behov:

i=0;

Opakuj pre celú reláciu:

- Prečítaj M blokov do operačnej pamäte
- Utrieď bloky v operačnej pamäti (heapsort, resp. quicksort)
- Zapíš utriedené bloky do behu R_i; i=i+1;

Nech N je počet behov

Vybrané fyzické operátory: merge-sort

Algoritmus sort (externý merge-sort)

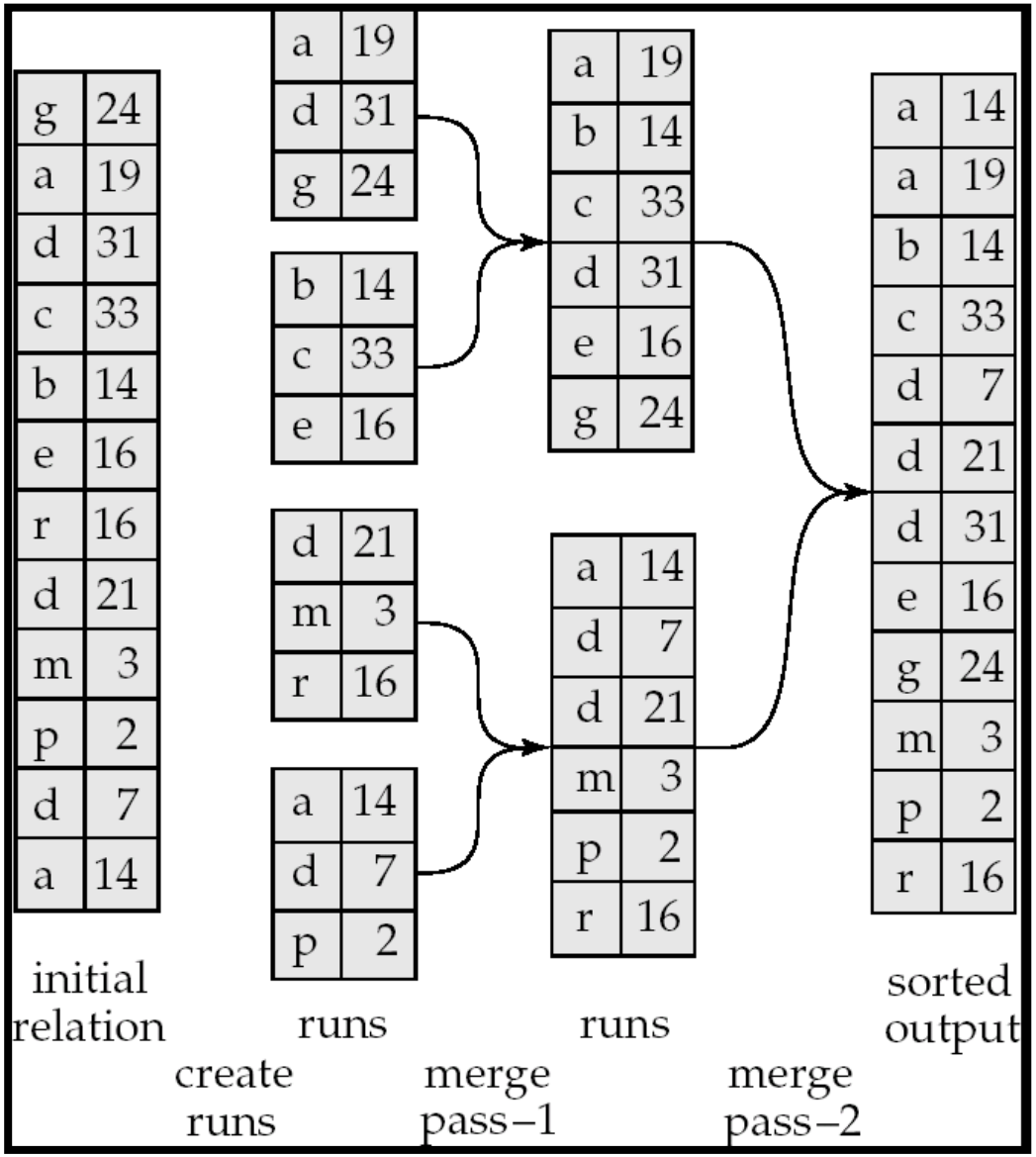
2.fáza, spájanie behov do väčších behov (pre jednoduchosť predpokladajme, že N<M). Použi M-1 blokov v pamäti pre vstup, 1 blok pre výstup:
Načítaj 1 blok *z každého* behu.
Opakuj kým sú vstupné bloky neprázdne:

- Vyber najmenší záznam spomedzi všetkých M-1 behov v pamäti
- Zapíš ten záznam do výstupného bloku. Ak je výstupný blok plný, zapíš ho na disk
- Presuň sa cez ten najmenší záznam. Ak v tom vstupnom bloku nie je ďalší záznam, načítaj ďalší blok z toho behu

Ak N≥M, druhá fáza sa musí niekoľko krát opakovať, až kým nevznikne ***jeden*** utriedený beh

Vybrané fyzické operátory: merge-sort

Príklad (Silberschatz et al.), M=3:



Vybrané fyzické operátory: merge-sort

Zložitosť:

- Počet opakovaní druhej fázy: $\lceil \log_{M-1} (B(R) / M) \rceil$
- Počet diskových operácií v 1. fáze aj v 2. fáze je 2 B(R), ak počítame aj výstup na disk
- Celkový počet diskových operácií je (približne)
  $cost = 2\ B(R)\ (1 + \lceil \log_{M-1} (B(R) / M) \rceil)$

Vybrané fyzické operátory: merge-sort

Iterator **Sort(R)**

```
private table oldR, newR;
open(R) {
  R.open();
  mergesort(R, newR);
  R.close();
  oldR=R;
  R=newR;
  R.open();
}
next(R) {
  return R.next();
}
close(R) {
  R.close();
  R.drop();
  R=oldR;
}
```

```
oscon=# explain select oid from pg_proc order by oid;
                        QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=181.55..185.92 rows=1747 width=4)
   Sort Key: oid
   ->  Seq Scan on pg_proc  (cost=0.00..87.47 rows=1747
width=4)
```

- Explicit: ORDER BY clause
- Implicit: Handling Unique, Sort-Merge Joins, and other operators
- Has startup cost: cannot return right away

Takmer celú prácu iterátora sort treba urobiť počas open(). V tomto príklade je odhad startup cost 185.92, čo zodpovedá internému volaniu mergesort. Ako už vieme, mergesort je 2-fázový (resp. multifázový, ak je málo RAM):

1. Číta postupne všetky bloky, utriedi každý z nich v RAM a zapíše na disk.
2. Spojí utriedené bloky do jednej tabuľky.

Vybrané fyzické operátory: nested-loop join

Algoritmus join (nested-loop join)

$R \bowtie_c S$

Predpokladajme, že R má menej blokov než S.

Prečítaj M-2 blokov **menšej** relácie R do RAM. 1 blok rezervuj pre vstup S a 1 blok pre výstup. Postupne prečítaj všetky bloky S, a pre každý blok aplikuj join záznamov R v RAM na záznamy v práve načítanom bloku S. Ak joinovacia podmienka *c* platí, zapíš výsledný záznam do výstupného bloku. Ak je výsledný blok plný, zapíš ho na disk

Ak R má viac než M-2 blokov, prečítaj nasledujúcich M-2 blokov R do RAM a zopakuj postupné čítanie celej S. Atď. (počet opakovaných čítaní S je $\lceil B(R) / (M-2) \rceil$).

cost = $B(R) + \lceil B(R) / (M-2) \rceil * B(S)$

Toto nezahŕňa počet zápisov na disk, veľkosť joinu závisí od joinovacej podmienky a obsahu relácií. Špeciálnym prípadom je kartézsky súčin, kde joinovacia podmienka splnená pre každú dvojicu záznamov

Vybrané fyzické operátory: nested-loop join

Iterator **NestedLoopJoin(R,S)**

```
private tuple r,s;
open(R,S) {
  R.open();
  S.open();
  s=S.next();
}
close(R,S) {
  R.close();
  S.close();
}
```

```
next(R,S) {
  if (s==EOF)
    return EOF;
  do {
    r=R.next();
    if (r=EOF)
      R.close();
      s=S.next();
      if (s==EOF)
        return EOF;
      R.open();
  } while not (r and s join);
  return join of r and s;
}
```

```
oscon-# select * from pg_foo join pg_namespace on
(pg_foo.pronamespace=pg_namespace.oid);
                        QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=1.05..39920.17 rows=5867 width=68)
   Join Filter: ("outer".pronamespace = "inner".oid)
   -> Seq Scan on pg_foo  (cost=0.00..13520.54 rows=234654 width=68)
   -> Materialize  (cost=1.05..1.10 rows=5 width=4)
        -> Seq Scan on pg_namespace  (cost=0.00..1.05 rows=5 width=4)
```

- Joins two tables (two input sets)
- USED with INNER JOIN and LEFT OUTER JOIN
- Scans 'outer' table, finds matches in 'inner' table
- No startup cost
- Can lead to slow queries when not indexed, especially when functions are in the select clause

Vybrané fyzické operátory: merge join

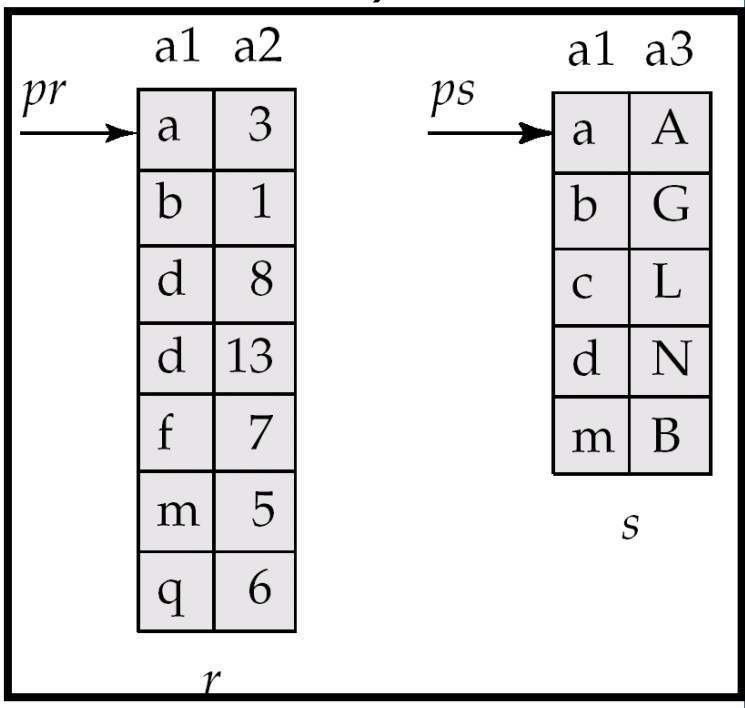
Algoritmus join (merge join)

Tento algoritmus dovoľuje len joinovaciu podmienku na rovnosť (equijoin, resp. natural join) a predpokladá, že obe relácie sú utriedené

cost = B(R) + B(S)

(Cena písania na disk tu nie je zahrnutá, veľkosť joinu závisí od joinovacej podmienky a obsahu relácií.)

Podobný 2. fáze merge-sort algoritmu. Ale nie celkom rovnaký: rozdiel je v spracovaní duplikátov!



Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Iterator **IndexScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.index().next();
}
```

```
oscon=# explain select oid from pg_proc where oid=1;
                              QUERY PLAN
------------------------------------------------------------------------
 Index Scan using pg_proc_oid_index on pg_proc  (cost=0.00..5.99
rows=1 width=4)
   Index Cond: (oid = 1::oid)
```

IndexScan je pri veľkých reláciách oveľa rýchlejší ako SeqScan