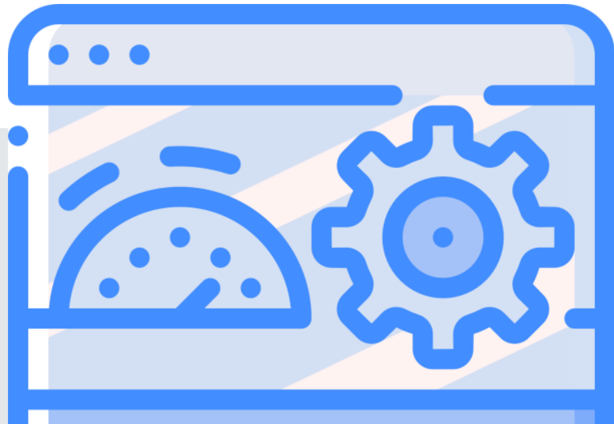
Lynt SERVICES

Relační databáze efektivně

Jan Smitka
jan.smitka@lynt.cz
@jansmitka
Lynt services s.r.o.



Optimalizovat?

Minulý rok zde zazněla přednáška „Optimalizace optimalizací“. Její hlavní myšlenkou bylo, abyste si spočítali, zda se vyplatí investovat vývojářský čas do optimalizace, nebo koupit další hardware.
„Koupit HW“ ale v databázích není tak jednoduché:

1. Pokud nemáte databázové servery připravené na replikaci, může být přidání dalšího DB serveru složité.
2. U komerčních databází je nutné připočíst cenu licencí – koukněte na ceník Oracle.
3. Úzké hrdlo bývá často disk a ten se škáluje špatně.

Oproti tomu základní optimalizace DB bývá snadná a pokud to před vámi někdo zanedbal, lze dotazy urychlit o desítky až stovky procent.

Icon made by Smashicons from www.flaticon.com

Vývojáři

Databáze se ale musí optimalizovat na míru konkrétní aplikaci. Musí to dělat někdo, kdo jí rozumí. Proto je tato přednáška určena především vývojářům, co se s databází někdy již setkali. Nečekejte úvod do problematiky, ani tipy, jak správně nastavit servery. Zaměříme se hlavně na praxi návrhu databáze a optimalizaci dotazů.

Icon made by Smashicons from www.flaticon.com

Databáze

V přednášce budu používat pojem „databáze“. Nebude tím myšlena nějaká konkrétní datová sada, ale samotný databázový software, nebo také databázový engine.
Není ale možné popsat všechny druhy databází, proto postupy zde uvedené budou do jisté míry obecné a společné pro většinu řádkových databází. Vaše databáze se může chovat jinak a umět něco navíc, proto si vše vyzkoušejte a zkonzultujte dokumentaci.
V přednášce budou také použity nekonkrétní pojmy „velký“, „malý“ a podobně. To je záměrně: vše záleží na konkrétní situaci a struktuře dat.

Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

Praktické příklady a postupy optimalizace si ukážeme na MySQL a PostgreSQL – dvou nejrozšířenějších open-source databázích.

Lynt SERVICES

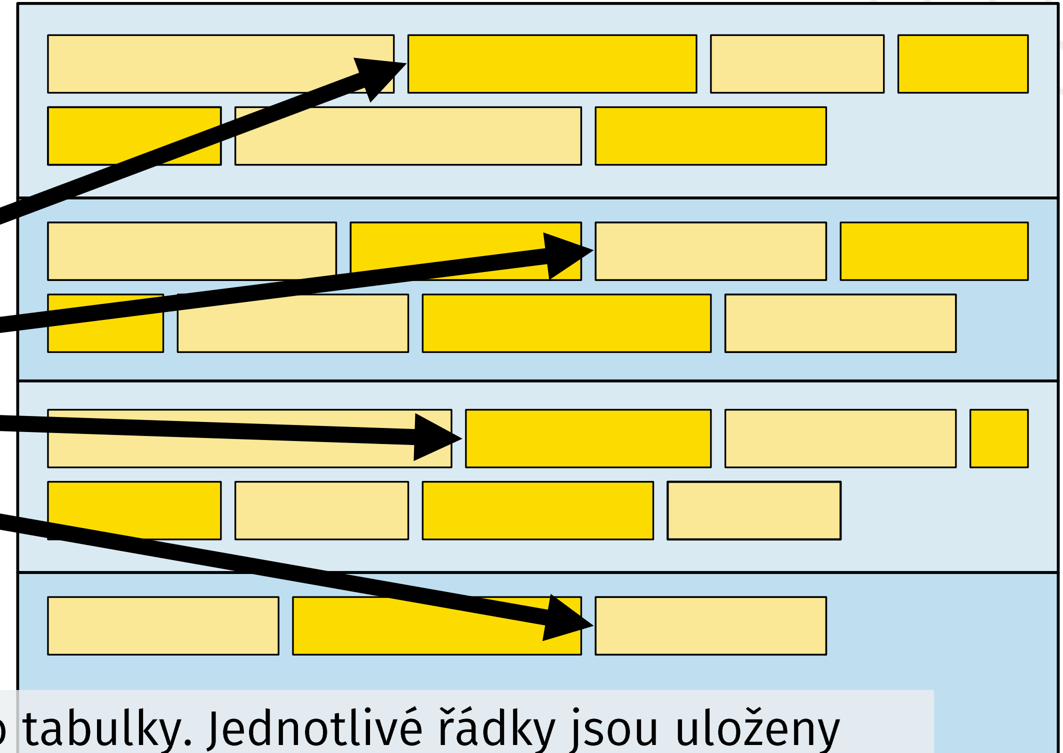
ULOŽENÍ TABULEK NA DISKU

Lynt SERVICES

Fyzické uložení tabulek

id	user_id	text	created_at
377	5	Mrznu ☹ ❄	2019-01-05
610	15	Konečně ☕	2019-09-01
987	15	Hacknuto.	2019-09-25
1597	28	Přednáška!	2019-10-08

posts



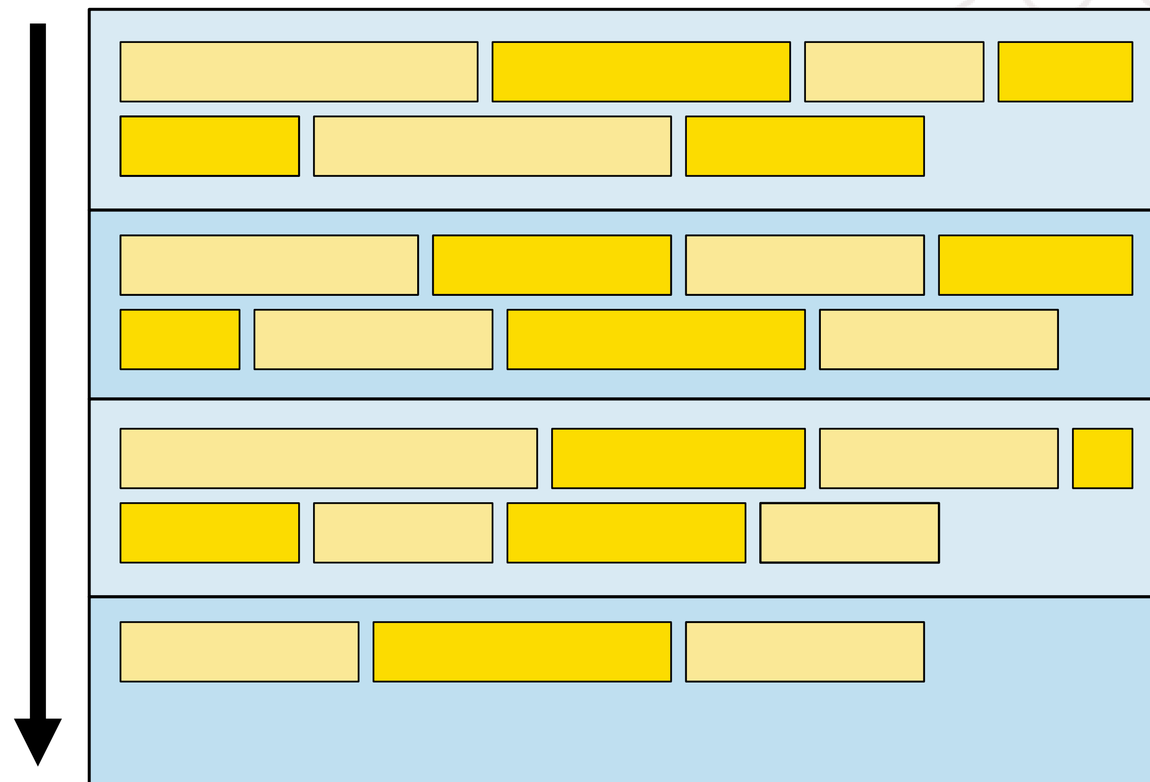
Databáze tabulky ve skutečnosti neukládají jako tabulky. Jednotlivé řádky jsou uloženy sekvenčně za sebou a rozdělené do bloků fixní velikosti. Když nějaký řádek smaže, vznikne v bloku prázdné místo, kam může v budoucnu uložit jiný řádek. Když dostupné bloky dojdou, alokuje si na disku nový.

Lynt SERVICES

Výběr všech dat

```
SELECT *
FROM posts
```

Pokud si vyžádáme celou tabulku, musí databáze všechny bloky sekvenčně přečíst a z nich načíst všechny řádky. Ty pak pošle ve výsledku.



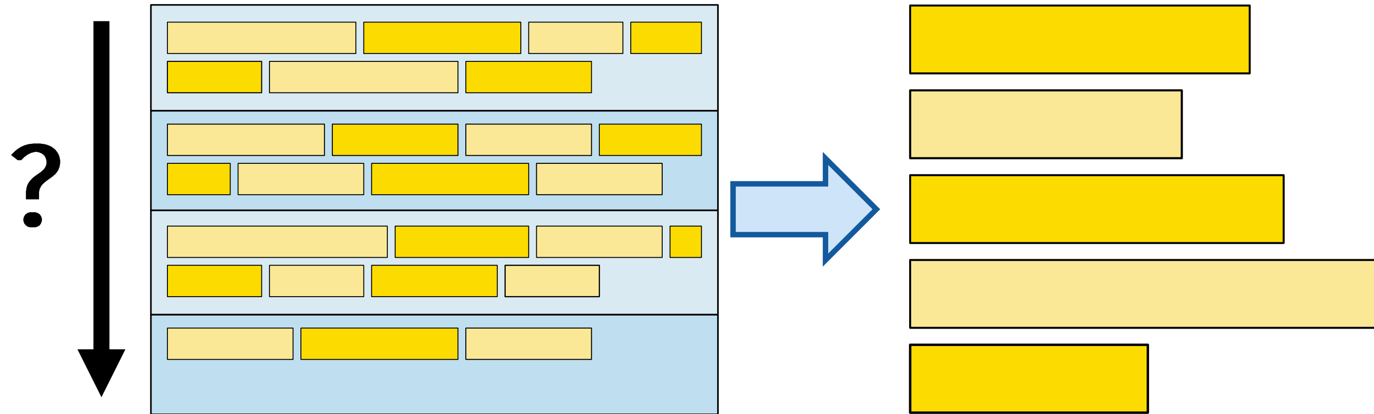
Lynt SERVICES

```sql
SELECT *
FROM posts
WHERE user_id = 42
ORDER BY created_at DESC
LIMIT 10
```

Co když ale budeme chtít něco složitějšího? Uvedený dotaz má podmínku, nastavené řazení a limit na prvních 10 řádků.

Lynt SERVICES

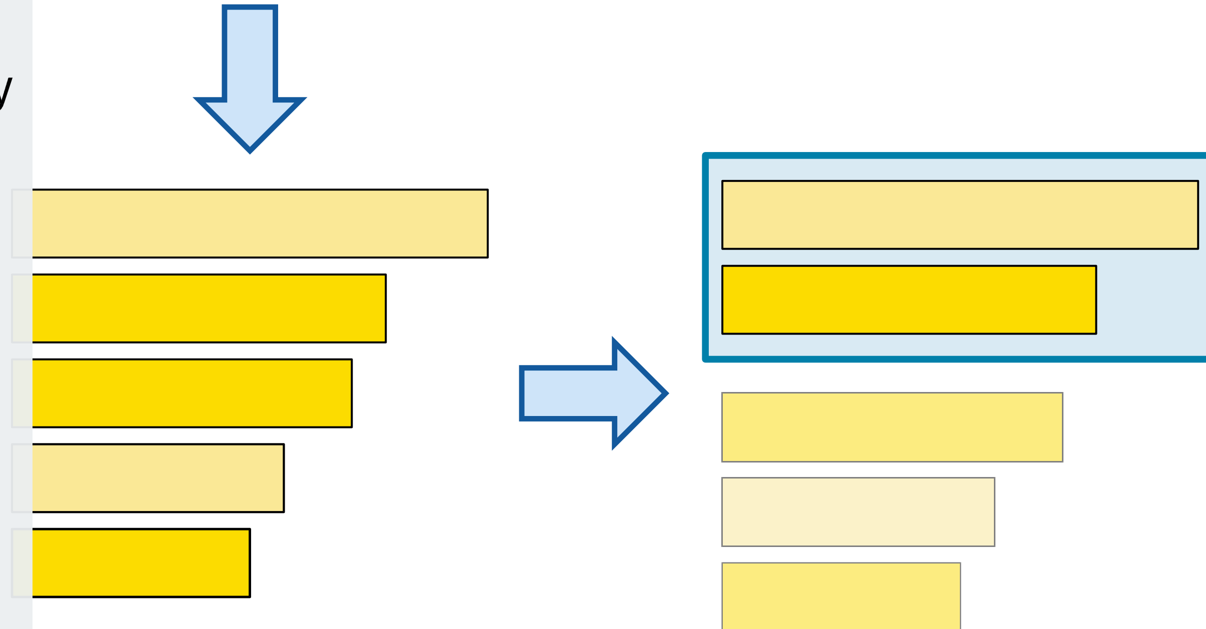
Sequential Scan (Full Table Scan)



Databáze zase musí načíst všechny řádky a otestovat, zda vyhovují podmínce. Řádky v paměti pak musí seřadit, vzít prvních 10 a celý zbytek pak zahodit.

Operaci načítání celé tabulky se říká sekvenční sken a u větších tabulek je pomalý.

Pokud tento dotaz poté provedeme znovu, bude pravděpodobně podstatně rychlejší.



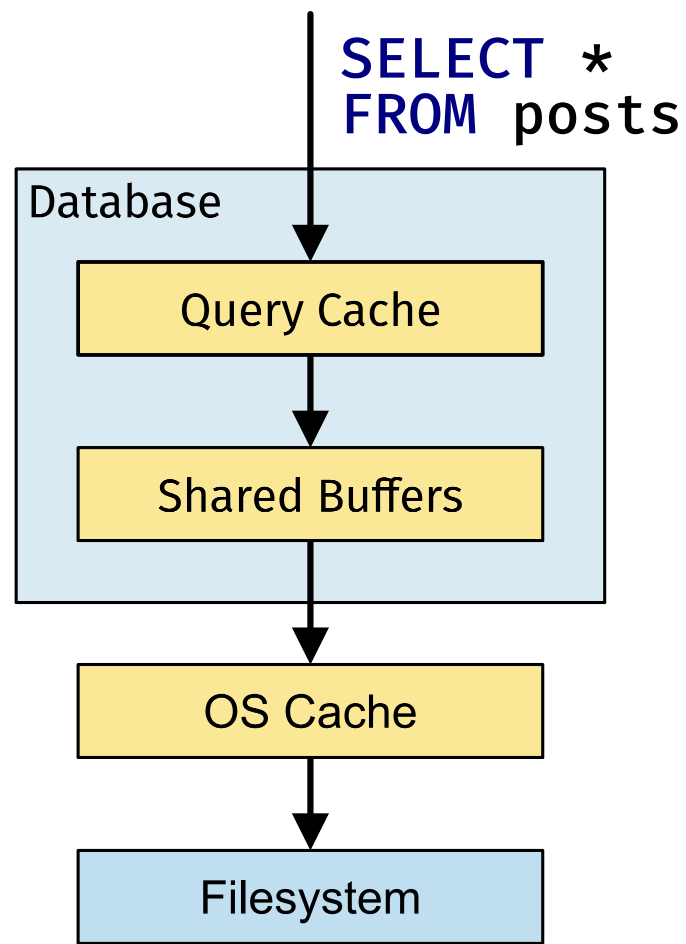
Lynt SERVICES

Cache

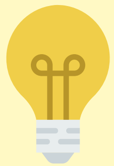
Mohou za to různé cache, které má databáze i operační systém.
DB nejprve kontroluje Query Cache, zda v ní nemá připravený již kompletní výsledek na zadaný dotaz.
Při načítání dat z disku jí pak pomáhají tzv. Shared Buffers, kde ukládá načtené bloky z tabulek.

Mimo kontrolu databáze pak je cache samotného operačního systému, který využívá nepotřebnou paměť k tomu, aby nemusel používané soubory opětovně číst z disku. Některé databáze ale své soubory otevírají v režimech, kde se cache systému nepoužívá a spoléhají pouze na svoji cache.

Pokud při testech budou vaše data v cache, dostanete lepší výsledek.



Lynt SERVICES



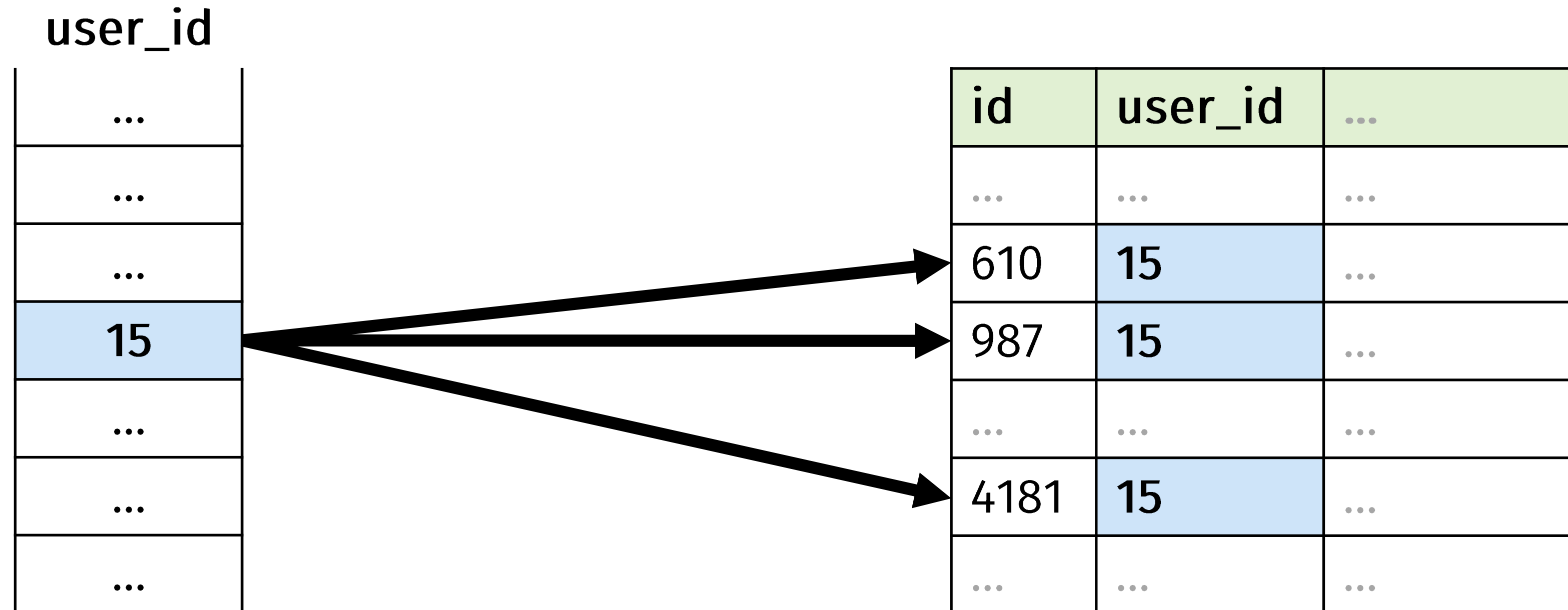
Zajistěte, aby vaše měření neovlivňovaly cache.

Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

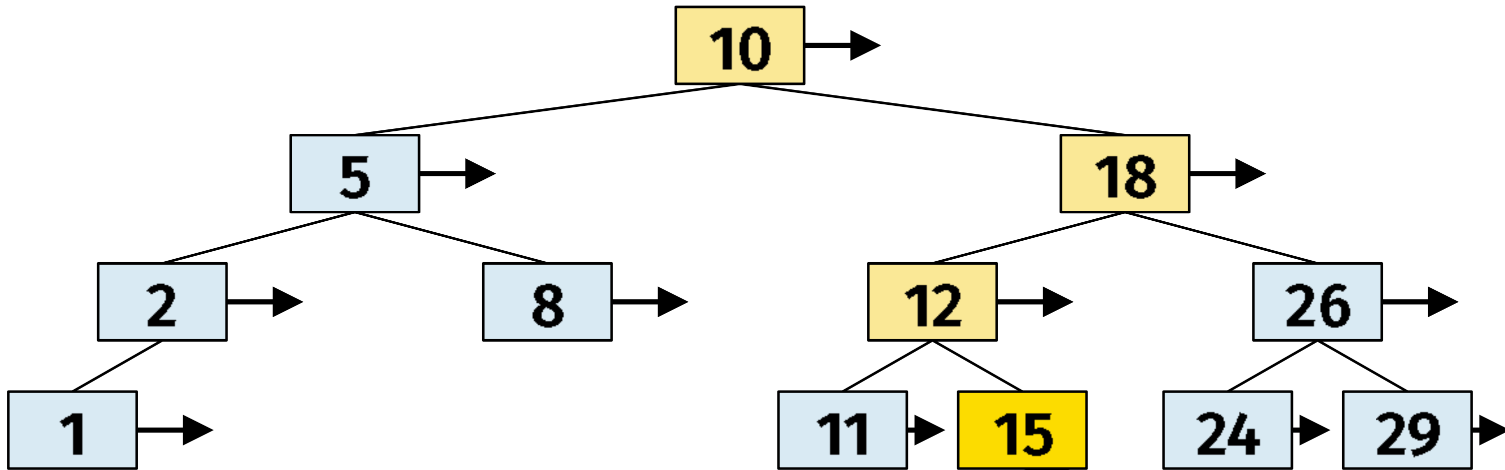
INDEXY

Index



Pokud chceme čtení urychlit nezávisle na cache, musíme použít indexy. Vytváří se nad jedním či více sloupci v tabulce a jde o datovou strukturu, která databázi umožňuje rychle najít řádky s odpovídající hodnotou.

Binární vyhledávací strom

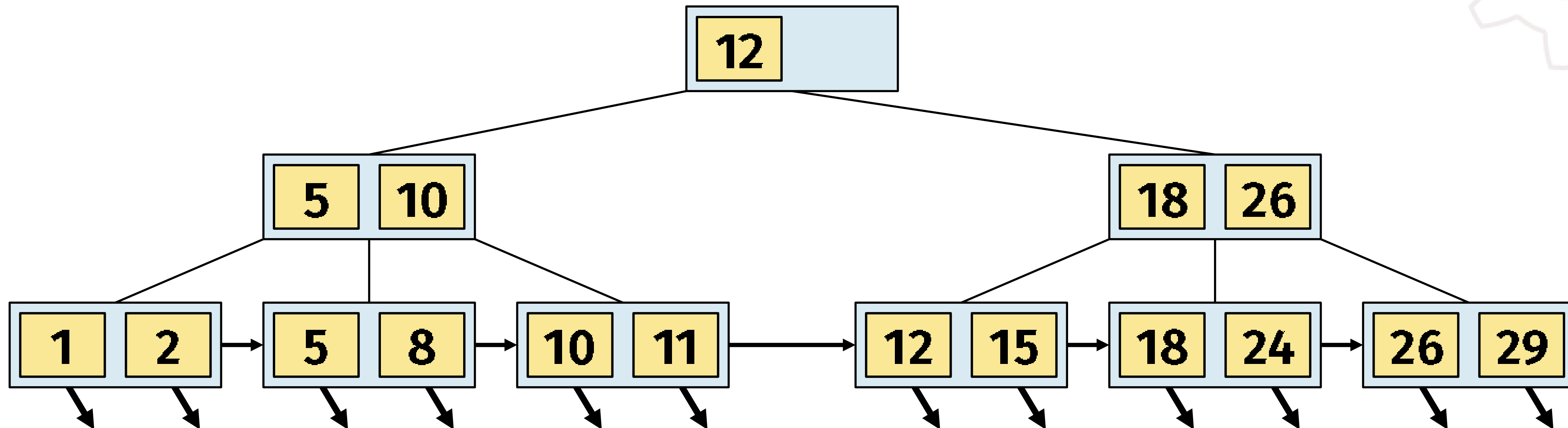


id	user_id	...
...	...	...
...	...	...
987	15	...
...	...	...
...	...	...
...	...	...

Tyto struktury jsou většinou odvozené od vyhledávacích stromů.
Nejjednodušším je binární vyhledávací strom. Vyhledávání v něm má logaritmickou náročnost.
V každém uzlu je uložena konkrétní hodnota a odkaz na řádek tabulky.
Princip hledání je popsán například na wiki:
https://cs.wikipedia.org/wiki/Bin%C3%A1rn%C3%AD_vyhled%C3%A1vac%C3%AD_strom

Lynt SERVICES

B+TREE



Pro databáze má ale lepší vlastnosti B+strom. V jednom uzlu je několik hodnot a může mít několik potomků. Odkazy do tabulky jsou uloženy v nejnižší úrovni, kde jsou navíc seřazeny a uloženy podobně, jako spojový seznam. To umožňuje načtení řádků již seřazených podle zadané hodnoty.

Lynt SERVICES

VYHLEDÁVÁNÍ POMOCÍ INDEXŮ

Lynt SERVICES

```
WHERE user_id = 15
WHERE user_id = 15 OR user_id = 25
WHERE user_id IN(15, 25, 30)
WHERE user_id <= 15 AND user_id >= 5
WHERE user_id BETWEEN 5 AND 15 OR
      user_id BETWEEN 25 AND 30
WHERE text LIKE 'data%'
```

```
WHERE user_id + 5 <= 15
WHERE ABS(user_id) <= 15
WHERE text LIKE '%data%'
```

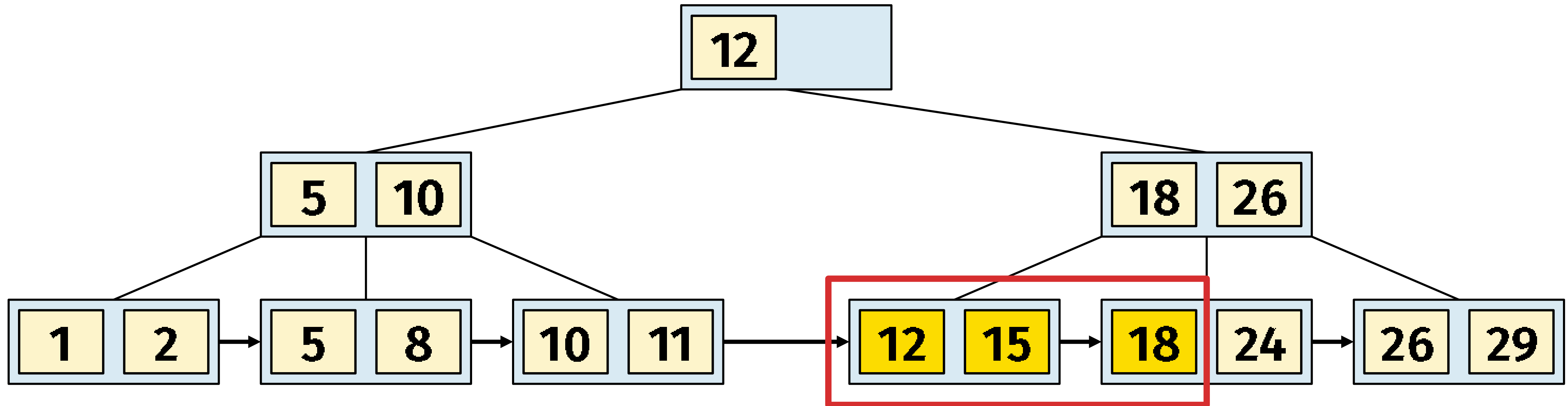
Index lze efektivně použít pro různé podmínky porovnávající s konstantami a pro vyhledávání podle prefixu řetězce.

Pokud ale sloupec v indexu přidáme do výrazu (byť jen přičtením konstanty), tak již index použít nelze. Nelze také vyhledávat uprostřed či na konci řetězce.

Lynt SERVICES

Využití indexů pro vyhledávání

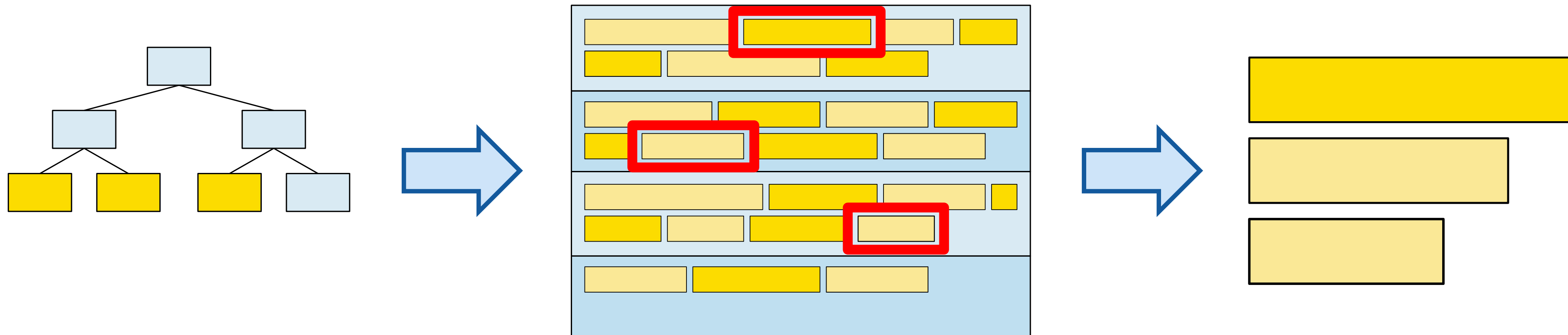
```
SELECT * FROM posts
WHERE user_id BETWEEN 12 AND 18
```



Při vyhledávání databázi stačí v indexu najít uzly s hodnotami odpovídající podmínce.

Lynt SERVICES

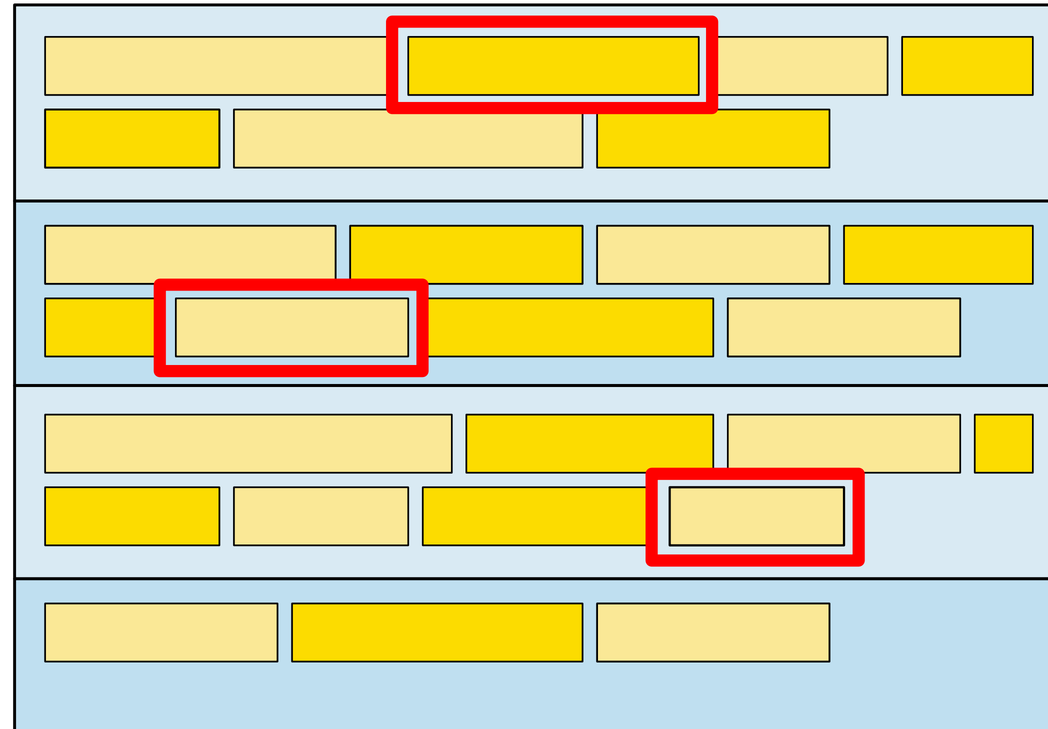
Odpovídající řádky pak jen načte z tabulky a výsledek se rovnou může poslat aplikaci.



Index Scan

Lynt SERVICES

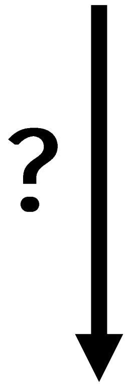
Náhodný přístup k datům

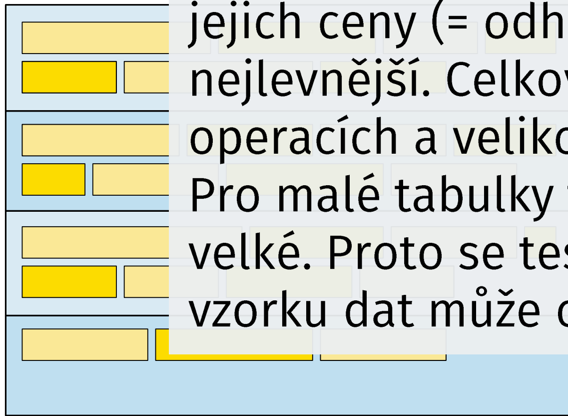


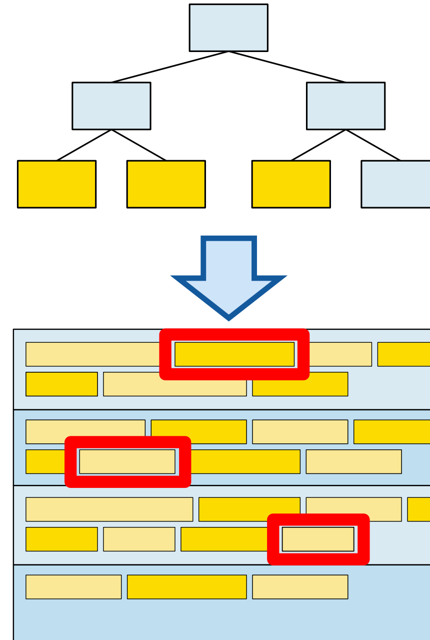
Trochu problematické je ale načítání příslušných řádek z disku. Databáze v tomto případě data načítá z různých míst na přeskáčku (= pro disk náhodně). I na dnešních rychlých SSD je tato operace pomalejší, než načíst celý soubor sekvenčně, a někde může být ve výsledku lepší provést sekvenční sken. K tomu dochází zvláště u malých tabulek.

Plánování dotazu: Cost-Based

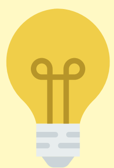
Vyhodnocení, co je lepší, má na starosti plánovač. Ten většinou pracuje na základě odhadu ceny: sestaví možné plány, vypočte jejich ceny (= odhadne náročnost) a vybere ten nejlevnější. Celková cena závisí na použitých operacích a velikosti i rozložení dat na disku. Pro malé tabulky tak vyjdou jiné ceny, než pro velké. Proto se test provedený na testovacím vzorku dat může od skutečnosti zásadně lišit.







Cost: X ?< Y



Testujte na datech, která co nejvíce odpovídají skutečnosti.

Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 12 AND 18
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	posts	**range**	user_id	**user_id**	NULL	39	**100**	**Using index condition**

https://dev.mysql.com/doc/refman/8.0/en/explain-output.html

Abychom si ověřili, který plán databáze zvolí, můžeme použít příkaz EXPLAIN. Výstupem je v MySQL tabulka, ve které budou rozepsány operace pro jednotlivé tabulky. Vše je dobře popsáno v dokumentaci. Zaměřte se zejména na slouce:
type: způsob přístupu k tabulce. Zde range = načtení rozsahu z indexu.
key: použitý index.
rows a **filtered:** odhad počtu načtených řádků a odhad, kolik % se použije ve výsledku.
Extra: další informace o postupu zpracování.

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 12 AND 18
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	posts	**ALL**	NULL	**NULL**	NULL	**1635**	**11.11**	**Using where**

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	posts	**range**	user_id	**user_id**	NULL	**39**	**100**	**Using index condition**

MySQL™

Pokud by se index neexistoval, ve sloupci **type** by bylo ALL (načtení celé tabulky), sloupec **key** by byl prázdný, **rows** by odpovídal odhadu řádků v celé tabulce a ve sloupci **filtered** by byla hodnota menší než 100, která indikuje, že databáze při načítání dat musí provádět dodatečné filtrování. Hodnota 11.11% je špatná, hodnoty < 1% jsou už opravdu velmi špatné.

```
EXPLAIN FORMAT=TREE SELECT * FROM posts …
```

-> Filter: (posts.user_id between 12 and 18)
  -> **Table scan** on posts

-> **Index range scan** on posts using user_id, with index condition:
  (posts.user_id between 12 and 18)

>= 8.0

Formát tabulky je nepřehledný a určitě budete často koukat do dokumentace, co která hodnota znamená. Proto vývojáři v MySQL 8.0 přidali možnost přepnout na stromový výpis, který je pro člověka mnohem čitelnější. I bez dokumentace poznáte, kdy se používá index, a kdy se provádí sken celé tabulky.

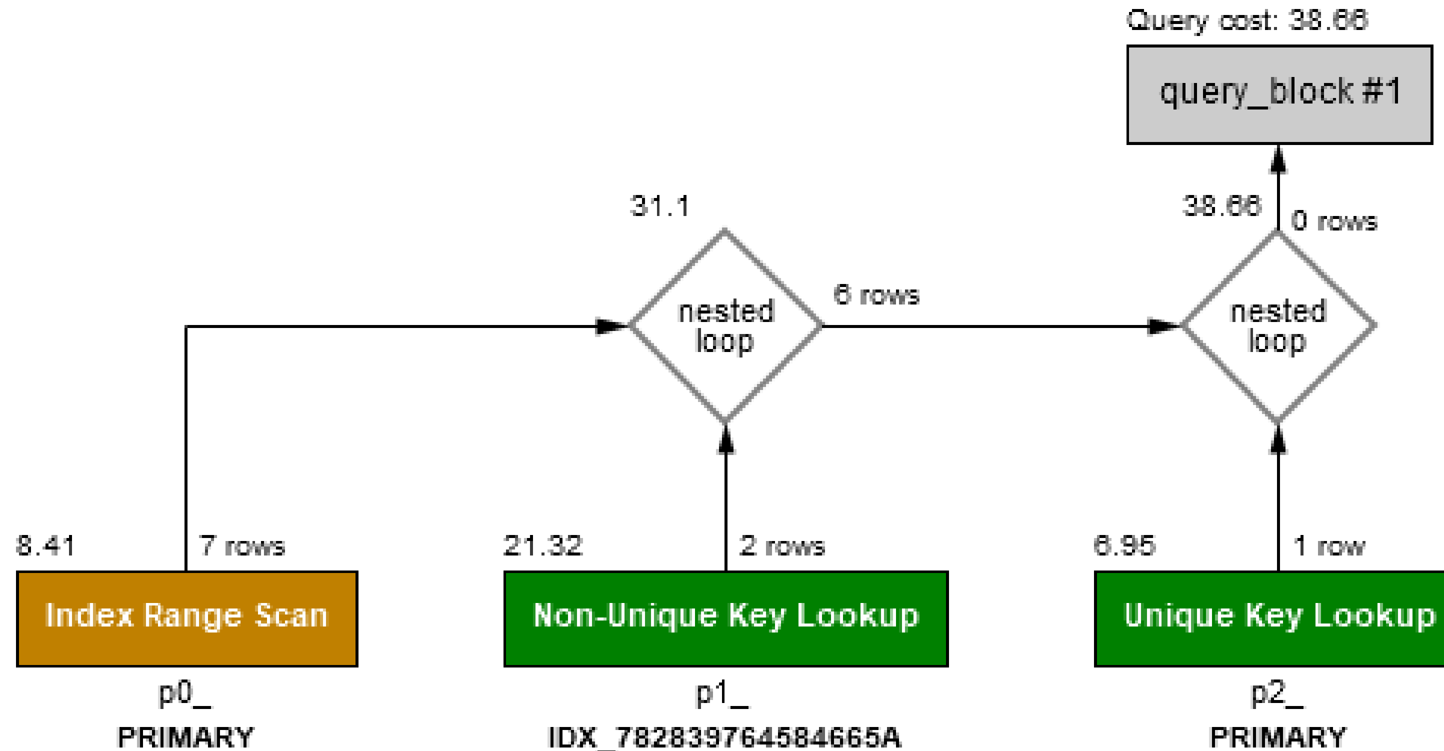
EXPLAIN FORMAT=JSON SELECT * FROM posts …

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": { "query_cost": "17.81" },
    "table": {
      "table_name": "posts",
      "access_type": "range",
      "possible_keys": ["user_id"],
      "key": "user_id",
      "used_key_parts": ["user_id"],
      // ...
      "filtered": "100.00",
      "index_condition": "(`test`.`posts`.`user_id` between 12 and 18)",
      "cost_info": { ... },
      "used_columns": [ ... ]
    }
  }
}
```

Ještě podrobnější je formát JSON. Ten už obsahuje i výsledné ceny plánů, ale je poměrně ukecaný a pro lidské konzumenty jej nedoporučuji.

Lynt SERVICES

MySQL Workbench



MySQL

Strojům ale sedí mnohem lépe, proto v nástroji MySQL Workbench najdete pěknou vizualizaci celého plánu, včetně použitých indexů, pořadí zpracování a všech cen.

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 15 and 18
```

QUERY PLAN
Seq Scan on posts (cost=0.00..31.33 rows=33 width=18)
Filter: ((user_id >= 12) AND (user_id <= 18))

QUERY PLAN
Index Scan using posts_user_id_idx on posts (cost=0.28..8.56 rows=14 width=18)
Index Cond: ((user_id >= 15) AND (user_id <= 18))

V PostgreSQL je výchozí formát podobný stromu v MySQL, jen je ještě o kousek podrobnější: obsahuje i všechny ceny a počty zpracovaných řádků.

PostgreSQL

```
EXPLAIN (ANALYZE, TIMING) SELECT * FROM posts
WHERE user_id BETWEEN 12 AND 18
```

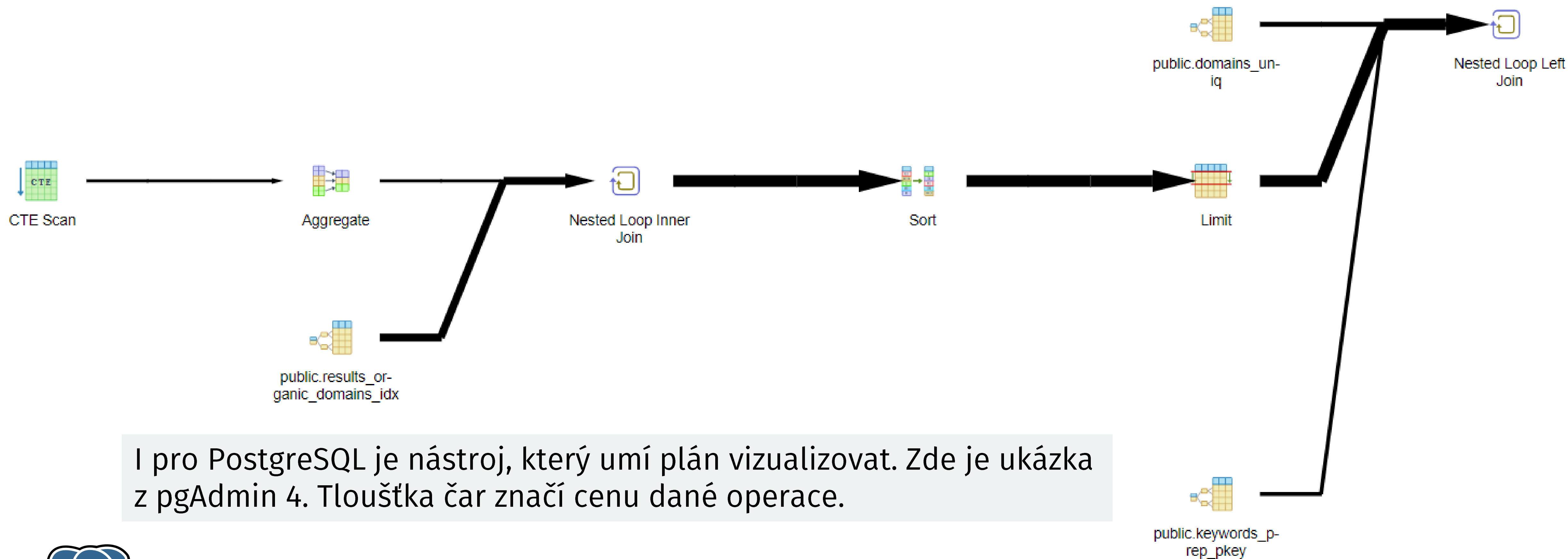
QUERY PLAN
Index Scan using posts_user_id_idx on posts (...) **(actual time=0.010..0.014 rows=35 loops=1)**
Index Cond: ((user_id >= 12) AND (user_id <= 18))
Planning Time: 0.078 ms
Execution Time: 0.024 ms

Jdou také zapnout další možnosti výstupu. Uvedením ANALYZE databáze dotaz vykoná a vypíše skutečný čas a počty řádků.

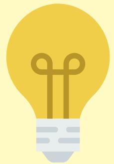
Volba TIMING pak vypíše i čas strávený plánováním a vykonáváním. Povšimněte si, že u tohoto jednoduchého dotazu plánování zabralo více než trojnásobek doby vykonávání. U složitých schémat a dotazů může doba plánování růst, většinou to ale nebude problém.

PostgreSQL

pgAdmin 4



I pro PostgreSQL je nástroj, který umí plán vizualizovat. Zde je ukázka z pgAdmin 4. Tloušťka čar značí cenu dané operace.

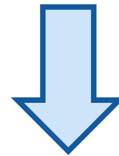
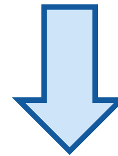
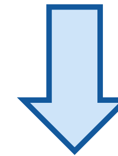


Neodhadujte, použijte EXPLAIN k ověření, jak se dotaz skutečně vykonává.

Icon made by Smashicons from www.flaticon.com

TABULKY S VÍCE INDEXY

```sql
SELECT * FROM people
WHERE last_name = 'Novák' AND
      gender = 'male' AND
      date_of_birth = '1990-01-25'
```

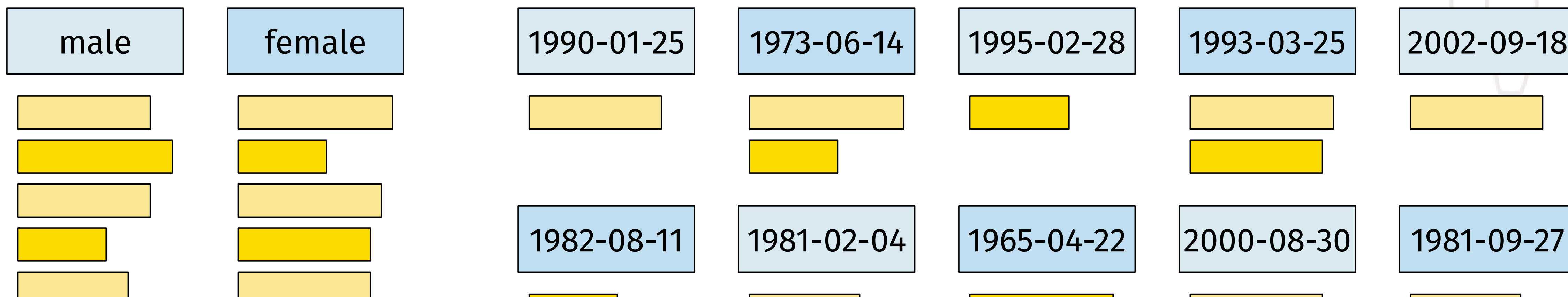




id	first_name	last_name	gender	date_of_birth
6765	Josef	Novák	male	1990-01-25
10946	Petra	Nováková	female	1973-06-14
17711	Jana	Nováková	female	1995-02-28
28657	Michal	Novátor	male	1993-03-25
46368	Petr	Nový	male	2002-09-18

people

V jedné tabulce můžete mít více indexů. Zde máme indexy nad sloupci last_name, gender, date_of_birth. Databáze většinou neumí efektivně využít více indexů najednou, proto pokud v dotazu uvedeme podmínky pro více sloupců s indexy, musí si některý z nich vybrat. Jak se ale rozhodnout, který bude lepší?

Kardinalita a selektivita indexu



Databáze si ke sloupcům v indexů udržují statistiky. Jednou z nich je i kardinalita: počet unikátních hodnot ve sloupci či indexu. Pokud unikátních hodnot bude mnoho, nalezením té správné v indexu se tak výrazně zúží množina řádků k prohledání.

Pokud použijeme pohlaví, zredukujeme to zhruba na 50% celku, pokud datum narození, tak na jednotky řádků (samozřejmě podle toho, jaká máme data).

Indexy s vysokou kardinalitou tak mají i vysokou selektivitu = lépe se podle nich hledá.

Lynt SERVICES

SHOW INDEXES FROM people

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	...
people	0	PRIMARY	1	id	A	48696	...
people	1	last_name	1	last_name	A	103	...
people	1	gender	1	gender	A	1	...
people	1	date_of_birth	1	date_of_birth	A	18429	...

V MySQL můžeme použít příkaz SHOW INDEXES. Ve výstupu najdeme statistiky pro jednotlivé indexy a jejich sloupce v tabulce. Pozor, vše jsou pouze odhady, počítat vše přesně by bylo náročné!

MySQL

Lynt SERVICES

```
SELECT * FROM pg_stats WHERE tablename = 'people'
```

attname	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	correlation
id	-1	NULL	NULL	{1,490,1032, ...}	1
first_name	198	{Rudolf,Marian,...}	{0.0065,0.006,...}	{Adam,Adéla,...}	0.001
last_name	100	{Kadlec,Polák,...}	{0.012,0.0115,...}	NULL	0.004
gender	2	{female,male}	{0.501,0.499}	NULL	0.504
date_of_birth	-0.34208	{1959-10-24,...}	{0.00036,...}	{1948-10-05,...}	0.003

https://www.postgresql.org/docs/11/view-pg-stats.html

V PostgreSQL existuje systémový pohled pg_stats. Obsahuje statistiky pro všechny sloupce (nejen ty s indexy). Zajímá nás hlavně sloupec n_distinct: kladná hodnota značí, že sloupec pravděpodobně obsahuje konečnou množinu hodnot a s přibývajícím počtem řádků se nebude zvětšovat. Záporná čísla jsou vydělena celkovým počtem řádků. ID (primární klíč) zde poto má -1, protože je unikátní.
Pohled dále obsahuje i například histogram a nejčastější hodnoty.

PostgreSQL

Lynt SERVICES

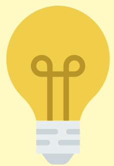
```
SELECT * FROM pg_stats WHERE tablename = 'people'
```

attname	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	correlation
id	-1	NULL	NULL	{1,490,1032, ...}	1
first_name	198	{Rudolf,Marian,...}	{0.0065,0.006,...}	{Adam,Adéla,...}	0.001
last_name	100	{Kadlec,Polák,...}	{0.012,0.0115,...}	NULL	0.004
gender	2	{female,male}	{0.501,0.499}	NULL	0.504
date_of_birth	-0.34208	{1959-10-24,...}	{0.00036,...}	{1948-10-05,...}	0.003

https://www.postgresql.org/docs/11/view-pg-stats.html

Zajímavý je i sloupec correlation: určuje závislost hodnoty a fyzického umístění řádku na disku. Pokud je vysoká, znamená to, že řádky s blízkou hodnotou jsou i blízko na disku. Databáze tak ví, že čte-li řádky se stejnou nebo blízkou hodnotou, nebude mít náhodný přístup tak velkou režii.

PostgreSQL

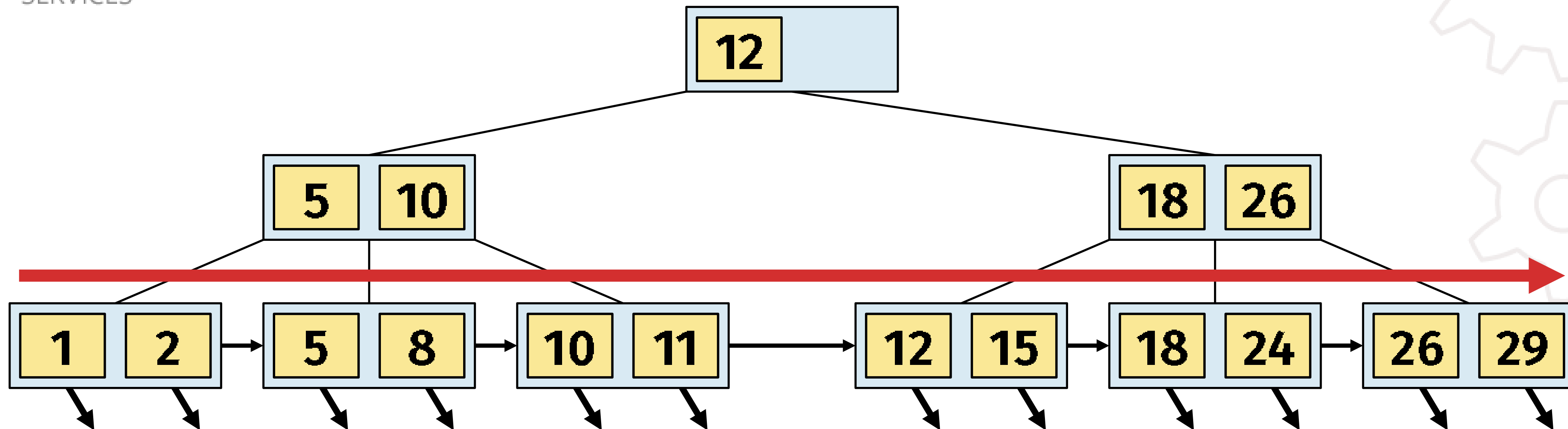


Indexy vytvářejte nad sloupci s vysokou selektivitou.

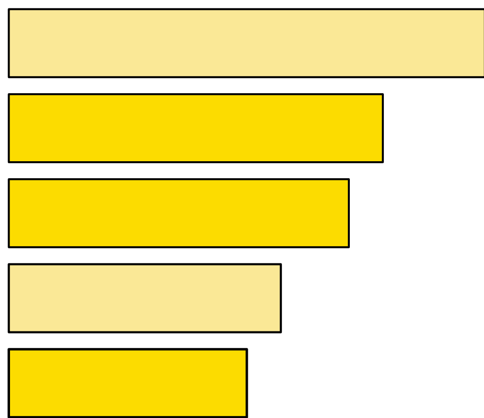
Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

ŘAZENÍ POMOCÍ INDEXŮ



B+Tree Index má také tu výhodu, že pokud databáze najde určitou množinu řádků, může je načíst již seřazené podle dané hodnoty.



Pořadí operací

1. Filtrace
2. Řazení

Obě operace musejí využívat **stejný index**

Má to ale i omezení: databáze nejprve filtrují a až poté řadí řádky. Pokud tedy jeden index použije k filtraci, musí stejný index použít i pro řazení, nebo načtené řádky dodatečně seřadit. Pokud filtrujete a řadíte podle jiných sloupců a na obou je index, dostane vždy přednost index pro filtraci.

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 12 AND 18
        ORDER BY user_id ASC
```

id	select_type	table	type	possible_keys	**key**	ref	rows	filtered	**Extra**
1	SIMPLE	posts	range	user_id	user_id	NULL	39	100	Using index condition

V dotazu řadíme podle stejného sloupce, podle kterého i filtrujeme. Pokud MySQL používá pro řazení index, z tabulky příkazu EXPLAIN se to přímo nedozvíme...

MySQL™

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 12 AND 18
        ORDER BY id ASC
```

id	select_type	table	type	possible_keys	**key**	ref	rows	filtered	**Extra**
1	SIMPLE	posts	range	user_id	user_id	NULL	39	100	Using index condition; **Using filesort**

… pokud ale index nepoužívá, ve sloupci Extra bude položka „Using filesort“. To vždy značí dodatečné řazení po načtení výsledků. Pro malé množství řádků to nevadí, pokud je ale řádků hodně, něco je asi špatně.

MySQL™

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 12 AND 18
        ORDER BY user_id ASC
```

QUERY PLAN
Index Scan using posts_user_id_idx on posts (cost=0.28..8.94 rows=33 width=18)
Index Cond: ((user_id >= 15) AND (user_id <= 18))

Stejná situace je i u PostgreSQL. Ve výpisu EXPLAINu nic o řazení podle indexu není.

PostgreSQL

Lynt SERVICES

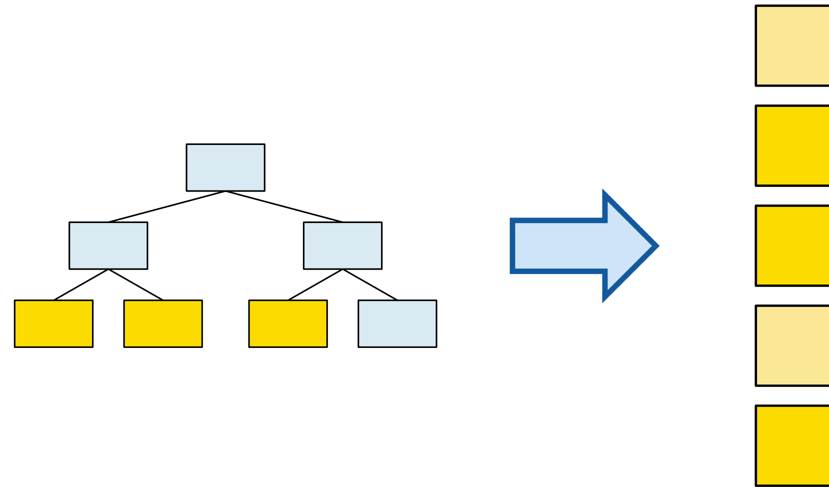
```sql
EXPLAIN SELECT * FROM posts
        WHERE user_id BETWEEN 12 AND 18
        ORDER BY id ASC
```

QUERY PLAN
Sort (cost=9.77..9.85 rows=33 width=18)
Sort Key: id
-> Index Scan using posts_user_id_idx on posts (cost=0.28..8.94 rows=33 width=18)
Index Cond: ((user_id >= 12) AND (user_id <= 18))

Pokud ale index použít nelze, tak se to dozvíte podle operace Sort.

PostgreSQL

```
SELECT user_id FROM posts
WHERE user_id BETWEEN 12 AND 18
ORDER BY user_id ASC
```



Speciální kategorií jsou dotazy, které nejen používají stejný sloupec s indexem pro filtraci a/nebo řazení, ale zároveň je tento sloupec jediný, který vybíráme. V takovém případě může databáze použít tzv. Index Only Scan. Při něm všechna data čerpá jen z indexu a ze samotné tabulky nic načítat nemusí.

Index Only Scan

```
EXPLAIN SELECT user_id FROM posts
        WHERE user_id BETWEEN 12 AND 18
        ORDER BY user_id ASC
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	posts	range	user_id	user_id	NULL	39	100	Using where; **Using index**

V MySQL se to dozvíme z položky „Using index" ve sloupci Extra.

MySQL™

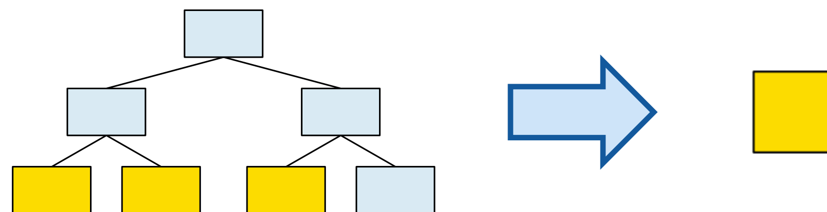
```sql
EXPLAIN SELECT user_id FROM posts
        WHERE user_id BETWEEN 12 AND 18
        ORDER BY user_id ASC
```

QUERY PLAN
Index Only Scan using posts_user_id_idx on posts (cost=0.28..8.94 rows=33 width=4)
Index Cond: ((user_id >= 12) AND (user_id <= 18))

V PostgreSQL uvidíme přímo Index Only Scan.

PostgreSQL

```sql
SELECT created_at FROM posts
ORDER BY created_at DESC LIMIT 1
```



Pozn.: následujících 5 slidů bylo v přednášce z časových důvodů vynecháno.
Někdy nám stačí nejen jediný sloupec, ale dokonce i jediný řádek – například pokud chceme znát datum, kdy někdo naposledy publikoval příspěvek. Ke zjištění této informace můžeme zkombinovat ORDER BY a LIMIT 1. Bez indexů by tento dotaz byl naprostá katastrofa – musela by se proskenovat celá tabulka, celá seřadit a pak by se vše, až na jeden řádek, zahodilo.

Lynt SERVICES

```
EXPLAIN SELECT created_at FROM posts
        ORDER BY created_at DESC LIMIT 1
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	posts	index	NULL	created_at	NULL	1	100	Backward index scan; Using index

S indexem je vše podstatně lepší: MySQL nám řekne, že používá index a skenuje jej odzadu (v dotazu máme ORDER BY ... DESC).

MySQL™

```sql
EXPLAIN SELECT created_at FROM posts
        ORDER BY created_at DESC LIMIT 1
```

QUERY PLAN
Limit (cost=0.28..0.34 rows=1 width=8)
-> **Index Only Scan Backward** using posts_created_at_idx on posts (cost=0.28..85.60 rows=1422 width=8)

Stejně tak i PostgreSQL.

PostgreSQL

Lynt SERVICES

EXPLAIN SELECT *MAX*(created_at) FROM posts

id	select_type	table	type	possible_keys	key	ref	rows	filtered	**Extra**
1	SIMPLE	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Select tables optimized away

V MySQL jde dotaz ještě trochu optimalizovat: místo ORDER BY a LIMIT jen použít MAX(…). Dotaz je tak ještě kratší a čitelnější, je na první pohled jasné, o co nám jde.

MySQL v tomto konkrétním případě zjistí, že mu z indexu stačí načíst první/poslední hodnotu a použije přístup optimalizovaný přesně pro tuto variantu. Ohlásí to položkou Select tables optimized away ve sloupci Extra.

Lynt SERVICES

EXPLAIN SELECT *MAX*(created_at) FROM posts

QUERY PLAN
Result (cost=0.34..0.35 rows=1 width=8)
InitPlan 1 (returns $0)
-> Limit (cost=0.28..0.34 rows=1 width=8)
-> Index Only Scan Backward using posts_created_at_idx on posts (cost=0.28..89.16 rows=1422 width=8)
Index Cond: (created_at IS NOT NULL)

PostgreSQL

PostgreSQL tuto optimalizaci nemá, tak vygeneruje plán, který na první pohled vypadá trochu složitěji. Má jen nepatrně vyšší cenu, poběží ale stejně rychle, jako varianta s ORDER BY a LIMIT. I přes to v této situaci raději používejte MAX(...), dotaz bude čitelnější.

Lynt SERVICES

INDEXY S VÍCE SLOUPCI

Lynt SERVICES

N-tice v indexu

(user_id, created_at)

...
(15, 2019-09-01)
(15, 2019-09-25)
(28, 2019-09-02)
(28, 2019-09-21)
(28, 2019-10-08)
...

...	user_id	...	created_at
...	...	...	...
...	15	...	2019-09-01
...	28	...	2019-09-02
...	...	...	...
...	28	...	2019-09-21
...	15	...	2019-09-25
...	28	...	2019-10-08

Index můžeme vytvořit i nad více sloupci. Jejich hodnoty budou uloženy ve formě n-tic. Tyto n-tice pak budou seřazeny podle prvního sloupce, pokud u několika řádků v prvním sloupci bude stejná hodnota, tak podle druhého sloupce. Pokud v prvním i v druhém bude stejná hodnota, tak podle třetího a tak dále.

Lynt SERVICES

Vyhledávání

```
WHERE user_id = 15
WHERE user_id = 15 AND
      created_at BETWEEN '2019-01-01'
                     AND '2019-01-31'
```

```
WHERE created_at = '2019-01-15'
WHERE user_id = 15 OR created_at = '2019-01-15'
```

Při vyhledávání musíme dávat pozor na pořadí. Nemusíme využít všechny sloupce, ale musíme je používat „od začátku“: lze filtrovat podle prvního, podle prvního a druhého, ale nikoliv jen podle druhého. Mezi podmínkami pro různé sloupce navíc musí být operátor AND.

Lynt SERVICES

Řazení

```
ORDER BY user_id ASC, created_at ASC
ORDER BY user_id DESC, created_at DESC
```

```
ORDER BY user_id ASC, created_at DESC
```

Při razení platí stejné pravidlo. Pokud jsme navíc při vytváření indexu neuvedli jinak, musíme použít stejný směr: všechny sloupce musejí mít buď ASC, nebo DESC. Pamatujte, že v indexu jsou uloženy jako sloupce, takže databáze musí být schopná řádky přečíst v tom pořadí, v jakém jsou v indexu (nebo v pořadí přesně opačném).

WHERE	= konst.	= konst.	rozsah	---	---
	A	B	C	D	E
ORDER BY	---	---	ASC	ASC	---

```
WHERE A = 1 AND B = 2 AND C > 4
    ORDER BY C ASC, D ASC
```

Při filtraci a řazení stačí dodržet 3 pravidla. Uvažujme dotaz na tabulkou s indexem nad sloupci A až E:

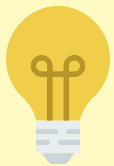
1. Řadit podle po sobě jdoucích sloupcích: řadíme podle sloupců C a D ve směru ASC.
2. Všechny sloupce před sloupci pro řazení musejí mít podmínku rovnosti s konstantou: sloupce A a B mají podmínku na rovnost s čísly 1 a 2.
3. Sloupec může mít podmínku na rozsah, je-li posledním sloupcem s podmínkou a prvním sloupcem pro řazení: u sloupce C je podmínka > 4, zároveň je ale prvním sloupcem, podle kterého řadíme. Další sloupce podmínku nemají.

V tomto případě jsou všechny podmínky splněny a databáze může index použít.

WHERE	= konst.	= konst.	rozsah	---	---
	A	B	C	D	E
ORDER BY	---	---	ASC	ASC	---

Následující dotazy však index použít nemohou:
První nemá žádnou podmínku nad sloupci před sloupcem C.
Druhý má podmínku rozsahu nad sloupcem, podle kterého se neřadí (to by šlo ale spravit, pokud bychom upravili řazení na A ASC, B ASC):

```
ORDER BY C ASC
WHERE A > 4 ORDER BY B ASC
```



Indexy vytvářejte podle nejčastějších dotazů tak, aby databáze nemusela zbytečně filtrovat a řadit.

Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

INDEXY A JOINY

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        INNER JOIN users
        ON posts.user_id = users.id
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	users	ALL	NULL	NULL	NULL	405	100	NULL
1	SIMPLE	posts	ALL	NULL	NULL	NULL	52409	10	Using where; Using join buffer (Block Nested Loop)

Pokud provádíme JOIN nad tabulkami podle sloupců, na kterých není index (nebo jsou součástí indexů s více sloupci a nelze ke použít – platí stejné podmínky, jako při filtraci), musí databáze načítat obě tabulky a pak je v paměti spojovat. To je velmi neefektivní. V MySQL to poznáme podle ALL ve sloupci **type** a NULL ve sloupci **key**.

MySQL™

Lynt SERVICES

```sql
EXPLAIN SELECT * FROM posts
        INNER JOIN users
        ON posts.user_id = users.id
```

QUERY PLAN
Hash Join (cost=7.63..2019.82 rows=66787 width=36)
Hash Cond: (posts.user_id = users.id)
-> **Seq Scan on posts** (cost=0.00..1093.87 rows=66787 width=19)
-> Hash (cost=4.50..4.50 rows=250 width=17)
-> **Seq Scan on users** (cost=0.00..4.50 rows=250 width=17)

V PostgreSQL uvidíme Seq Scan.

PostgreSQL

Lynt SERVICES

```
CREATE TABLE posts (
    -- ...
    user_id INT
    -- ...
);

CREATE TABLE users (
    id BIGINT
    -- ...
);
```

Pro JOIN chceme, aby alespoň jeden sloupec byl v indexu, ideálně oba – databáze si může vybrat nejefektivnější způsob, jak tabulky spojit.

Oba sloupce ale musí být přesně stejného typu. Pokud budeme spojovat sloupce INT a BIGINT, index se nepoužije.

Oba sloupce v indexu chceme i kvůli cizím klíčům. Pokud máme nastavený cizí klíč posts.user_id -> users.id, tak se při smazání záznamu z tabulky users provede kontrola, jestli neexistuje v tabulce posts ještě nějaký záznam, který se na něj odkazuje.

Lynt SERVICES

```sql
CREATE TABLE posts (
    -- ...
    user_id VARCHAR(50)
    -- ...
);

CREATE TABLE users (
    id VARCHAR(60)
    -- ...
);
```

Při spojování podle indexů záleží i na délce VARCHARů a podobných typů. Pokud budou mít rozdílné maximální délky, index se opět nepoužije.

Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
       INNER JOIN users
       ON posts.user_id = users.id
```

id	select_type	table	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	posts	ALL	NULL	NULL	NULL	52409	100	NULL
1	SIMPLE	users	eq_ref	PRIMARY	PRIMARY	posts.user_id	1	100	NULL

Po doplnění indexu nad users.id vše vypadá lépe. U tabulky users vidíme type eq_ref (hledání rovnosti se sloupci v jiné tabulce) a používá se na to klíč PRIMARY, který před tím chyběl.

MySQL™

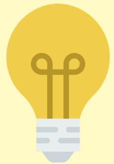
Lynt SERVICES

```
EXPLAIN SELECT * FROM posts
        INNER JOIN users
        ON posts.user_id = users.id
```

QUERY PLAN
Merge Join (cost=6474.65..7456.74 rows=66787 width=44)
Merge Cond: (users.id = posts.user_id)
-> **Index Scan using users_pk on users** (cost=0.42..6687.17 rows=200250 width=25)
-> Sort (cost=6445.94..6612.91 rows=66787 width=19)
Sort Key: posts.user_id
-> Seq Scan on posts (cost=0.00..1093.87 rows=66787 width=19)

PostgreSQL

I v PostgreSQL zmizel jeden sekvenční sken. Doplněním indexu nad posts.user_id by v závislosti na velikosti obou tabulek mohl zmizet i ten druhý.



V JOINech používejte sloupce stejných typů s indexy.

Icon made by Smashicons from www.flaticon.com

Slow Query Log

start_time	query_time	rows_sent	rows_examined	database	sql_text
04.10.2019 13:34	0:00:11	36	10882	test	SELECT * FROM posts ...
04.10.2019 12:49	0:00:24	998	106491	test	SELECT * FROM posts ...
04.10.2019 12:28	0:00:12	182	18383	test	SELECT * FROM users ...
04.10.2019 11:54	0:00:38	195	63986	test	SELECT * FROM users ...

Nyní už víme, jak indexy použít pro urychlení dotazů, jak ale vybrat, na které dotazy se zaměřit? Jedním vodítkem může být tzv. Slow Query Log. Ten obsahuje dotazy, které buď trvaly déle, než nastavený minimální čas (lze nastavit), nebo vůbec nevyužívaly indexy (nutno zapnout).

Ale pozor na to, že databáze může být zatížena velkým množstvím malých dotazů, které nepřekračují limit a částečně využívají indexy, ale mohly by být optimalizovány.

Na to jsou potřeba buď lepší nástroje (PostgreSQL má rozšíření pg_stat_statements, v cloudu Amazon Web Services je k dispozici Performance Insights), nebo lepší monitorování aplikace, např. Application Performance Monitoring (APM), které vám umožní měřit, co jaké dotazy aplikace provádí a kolik zabírají času. Zkuste najít, co máte k dispozici pro vaši platformu.

Lynt SERVICES

TEMNÁ STRANA INDEXŮ

https://lynt.cz @jansmitka

Lynt SERVICES

Velikost

table	table_size	index_size
domains	**294 MB**	**329 MB**
domains_meta	**1216 MB**	**1060 MB**
keywords	745 MB	209 MB
keywords_history	911 MB	0 bytes
results	29 GB	6466 MB

Stejně jako síla, i indexy mají svoji „temnou stranu“. Často se například zapomíná, že i index musí být někde uložen a ve výsledku mohou indexy zabírat více místa, než samotná tabulka. Zde příklad z jedné naší produkční databáze. Pokud se index nevejde do paměti, nemusí být tak efektivní.

Lynt SERVICES

```
CREATE INDEX posts_user_idx ON posts (user_id);
```

(user_id, id)

...	...
15	18ccbc80-3274-4b1f-b127-8498b77e9f89
15	c6104eb6-3551-4584-8ab2-d1860407ff42
28	69fff292-889c-4d3e-897c-1f59ca81a7f6
15	7abfa836-0a27-467a-81f2-5645ece8c87c
28	c90b32a7-cf86-4502-a45b-34fabc6cff25
28	d2618370-0ae6-4536-8ffc-b66e98c68a1e

V MySQL, konkrétně v jeho výchozím úložišti InnoDB, číhá ještě jedna zrada – odkaz do tabulky je v indexech realizován pomocí hodnoty primárního klíče. Je-li primární klíč velký (např. UUID či jiný dlouhý náhodný identifikátor), budou i všechny indexy velké, i když ho vytvoříte jen nad jedním sloupečkem typu INT. Snažte se proto nepoužívat dlouhé identifikátory jako primární klíče.

MySQL™

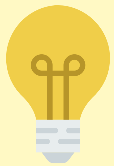
Lynt SERVICES

Aktualizace dat

```
INSERT INTO posts VALUES(...)

UPDATE posts SET ...
```

Pokud aktualizujete data v tabulce, musí se kromě samotné tabulky aktualizovat i indexy. To samozřejmě zabírá čas a aktualizace tabulek jsou pomalejší. U tabulek, do kterých se spíše zapisuje, proto nevytvářejte moc indexů.



U tabulek s častým zápisem omezte indexy.

Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

Autocommit

```
INSERT INTO posts VALUES(...);
INSERT INTO posts VALUES(...);
INSERT INTO posts VALUES(...);
INSERT INTO users VALUES(...);
INSERT INTO users VALUES(...);
INSERT INTO users VALUES(...);
```

U aktualizací ještě zůstaneme. Pokud při spojení nenastavíme jinak, pracuje většina databází v tzv. Autocommit řežimu. V tom každý příkaz obalí do samostatné transakce a po jeho provedení provede potvrzení (COMMIT) transakce. Při této operaci databáze zajistí, že jsou data bezpečně zapsána na disk a nově aktualizované řádky zpřístupňuje pro ostatní klienty. To má velkou režii, proto jsou tyto samostatné aktualizace neefektivní.

Lynt SERVICES

Explicitní transakce

```
START TRANSACTION;
    INSERT INTO posts VALUES(...);
    INSERT INTO posts VALUES(...);
    INSERT INTO posts VALUES(...);
    INSERT INTO users VALUES(...);
    INSERT INTO users VALUES(...);
    INSERT INTO users VALUES(...);
COMMIT;
```

Pouhým uzavřením příkazů do transakce všechny operace zrychlíme, protože potvrzování transakce a celá režie s tím spojená proběhne jen jednou.

Lynt SERVICES

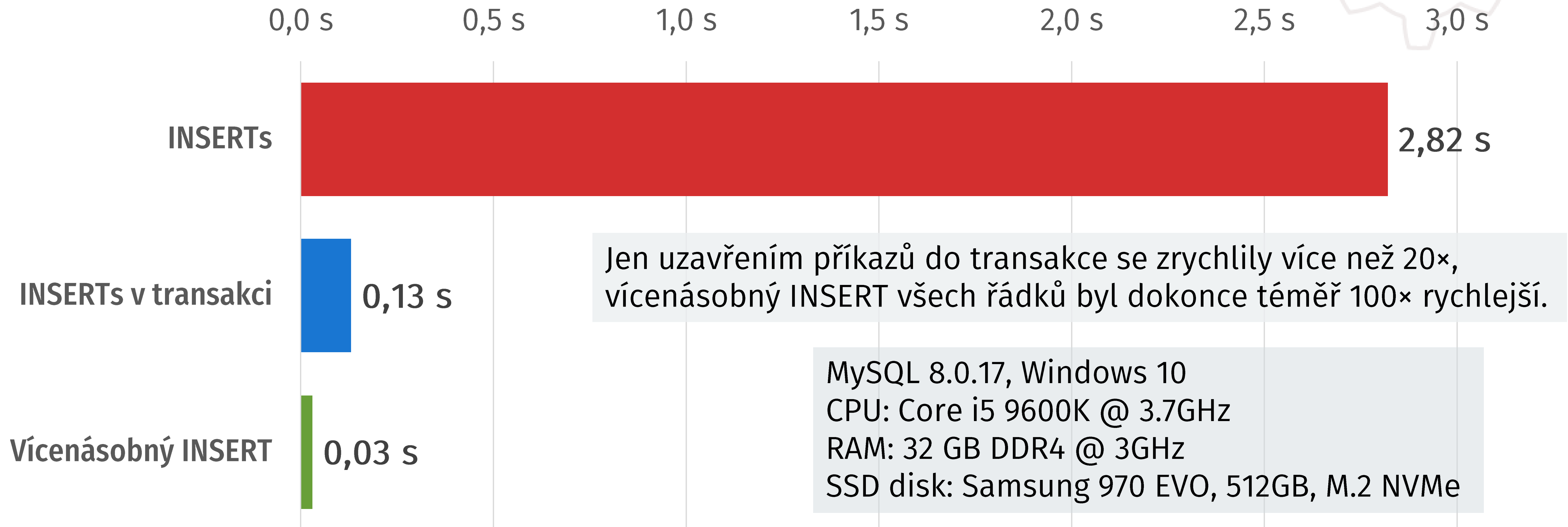
Vícenásobné operace

```
START TRANSACTION;
    INSERT INTO posts
        VALUES(...),
        VALUES(...),
        VALUES(...);
    INSERT INTO users
        VALUES(...),
        VALUES(...),
        VALUES(...);
COMMIT;
```

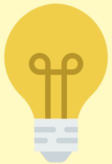
U INSERTů je ještě lepší používat hromadné inserty, kdy jedním příkazem vkládáme více řádků. Jen pozor na to, že některé databáze omezují maximální velikost příkazu (např. MySQL má ve výchozím nastavení 4MB).

Lynt SERVICES

Srovnání rychlosti



Lynt SERVICES



Pro zápis vždy používejte explicitní transakce a pokud to jde, tak i hromadné operace.

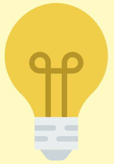
Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

ZÁVĚR

Lynt SERVICES

Klíčové tipy



1. Vytvářejte indexy podle předpokládaných dotazů, ne obráceně.
2. Testujte na reálných datech, používejte EXPLAIN.
3. Používejte transakce a hromadné operace.

Icon made by Smashicons from www.flaticon.com

Otázky?

Icon made by Smashicons from www.flaticon.com

Lynt SERVICES

Díky za pozornost!

https://lynt.cz @jansmitka