Towards more efficient query plans

PostgreSQL 11 and beyond

Alexander Kuzmenkov
a.kuzmenkov@postgrespro.ru

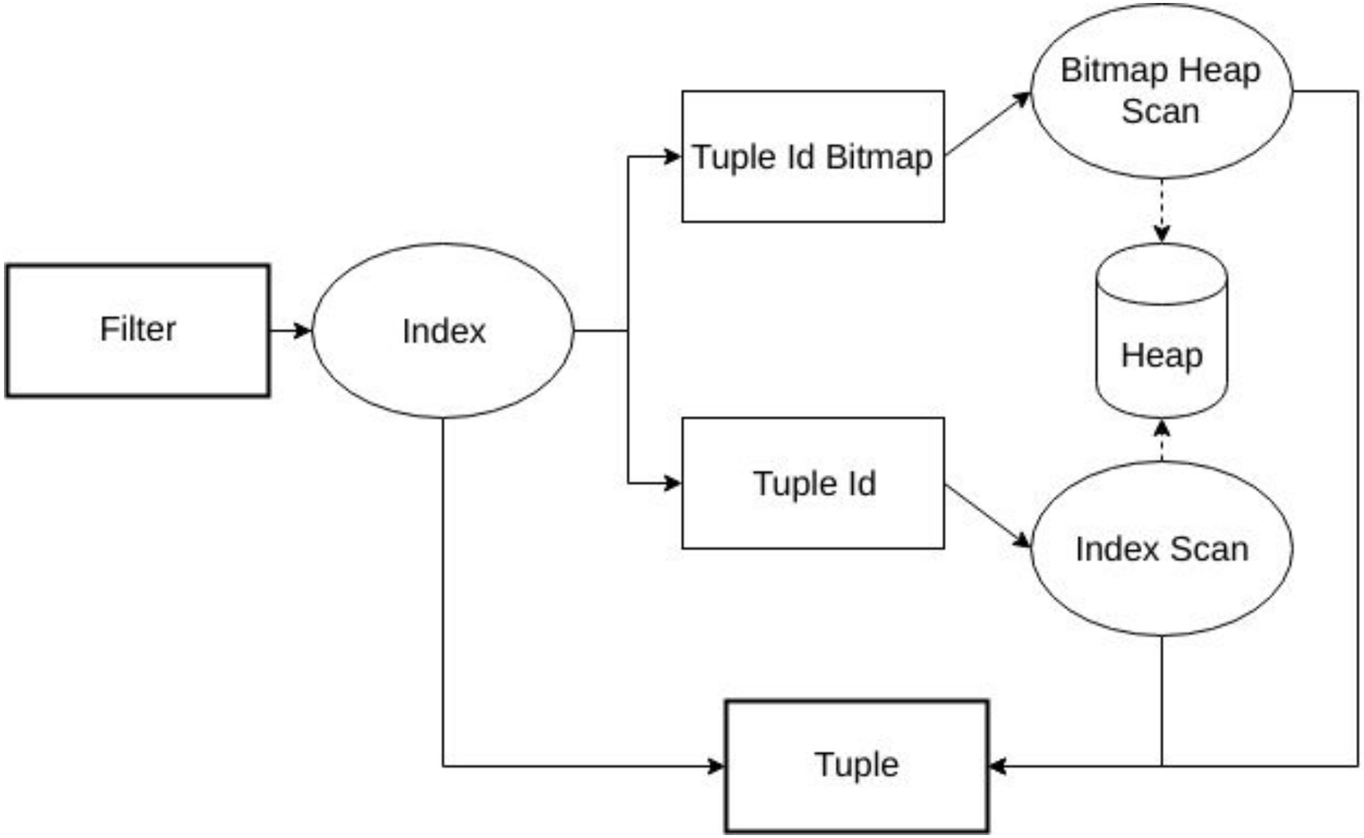
PROFESSIONAL
Postgres

What's a plan?

- SQL is a declarative language: “what”, not “how”
- Optimizer decides how to execute queries based on statistics about data and available resources

- A plan is a tree of simple building blocks
    - Scan
        - Table
        - Index
        - Function
        - Subquery
    - Join
        - Merge
        - Nested Loop
        - Hash
    - Sort/Group/Unique
    - etc.

Index scan


Bitmap Heap Scan
Tuple Id Bitmap
Filter
Index
Heap
Tuple Id
Index Scan
Tuple

Covering indexes

Index-only scan can return INCLUDEd columns, but these columns:

- do not participate in UNIQUE constraint
- do not require btree operators (e.g. point type)

```
# table t(a int = 1..1kk, b int in [0, 100), c text(60));
# create unique index idx_t on t(a) include (b);
# select a, b from t where a > 100000 (1/10 of the rows);
```

Index	**Plan**	**Time, ms**
unique on t(a)	Index Scan	30
unique on t(a), plain on t(a, b)	Index Only Scan	20
unique on t(a) include (b)	Index Only Scan	20

Index-only Bitmap Scan for count(*)

- for indexes that do not support index-only scan (e.g. GIN)
- don't fetch the tuples when we only need to count them
- fast and precise pagination, no need for the EXPLAIN trick
- needs adequate work_mem to fit the bitmap
- works only on vacuumed pages

```
# create index pglist_gin on pglist(fts) using gin;
# select count(*) from pglist
  where body_tsvector @@ to_tsquery('rebase');
```

Conditions	**Buffers: shared hit**	**Time, ms**
not vacuumed	95k	160
vacuumed	50	90

Bitmap	
Tuple ID	**?**
Page 1 Tuple 1	1
Page 1 Tuple 2	0
....	
Page N Tuple M	1
....	

Loose index scan

DEV

- Fast DISTINCT using a btree index
- Now done with Unique over sorted input

```
# create table t(a int), 100k ints [0, 500);
# create index idx_t_a on t(a);
# select distinct a from t;
```

Plan	**Time, ms**
Loose index scan	6
Unique over Index scan	97
Unique over Sort	160

Incremental sort

- Sort partially sorted input
- Reuse one index for similar ORDER BY queries or joins
- Read less rows with LIMIT

Who needs sorted output?

- ORDER BY
- DISTINCT
- GROUP BY
- window functions
- merge joins

```
# table t(a int, b int); 100k random ints in [1, 1000]
 -- groups of 100 rows with same 'a'
# index on t(a);
# select * from t order by a, b limit 1001;
```

Plan	**Rows read**	**Time, ms**
Incremental Sort over Index Scan	1101 (11 groups of 100 + 1)	3.2
Sort (top-N heapsort) over Seq Scan	100k	6.5

Estimate sort costs for GROUP BY

DEV

- Make sort cost accord for cardinality and order of columns
- Choose cheapest sort order for GROUP BY
- Example
  - “p” — high cardinality, cheap to compare
  - “v” — low cardinality, expensive to compare

Sort keys	**Sort time, ms**
p, v	800
v, p	1500

```
# select i/2 as p, format('%60s', i%2) as v into btg from
generate_series(1, 1000000) i;
# select count(*) from btg group by p, v;
```

Joins

Join types

- Inner
- Outer
- Semi/Anti

Optimizations

- Transitive equality
- Join strength reduction
- Join removal

How to choose the order of joins?

- System R
  - Finds the best join for 2 tables
  - Combines the best joins it found for N-1 tables to find the best ones for N
  - Too many combinations to try. Only used when N < geqo_threshold
- Genetic algorithm
  - Used when N >= geqo_threshold
  - A heuristic algorithm that doesn't try all the permutation

DEV

Multicolumn join selectivity

- Poor selectivity estimates for multicolumn join on correlated columns
- CREATE STATISTICS not helpful for joins
- Solution: create single–column statistics on composite values
- Do it automatically — there is probably and index on these columns

```
# create table t (a, b in [0, 10k)), 1M rows;
# select * from t t1, t t2 where t1.a = t2.a and t1.b = t2.b;
```

Real number of join rows	**Normal stats**	**Multicolumn index stats**
10M	100 (4 orders off!)	9.97M

Joins with a unique inner side

- On the inner side, at most one row matches the join clauses
- Proved by unique index for table or GROUP BY for subquery

Semi join

- WHERE EXISTS
- Like Inner, but:
  - Doesn't output inner columns
  - Doesn't output duplicates
- Can be reduced to inner join when the inner side is unique [10]

Skip materialization in merge joins

- Each inner tuple only used once => don't have to materialize the inner side [10]

Self join on primary key

DEV

- Frequent in ORM-generated queries
- Also happens when reusing complex views
- Can be replaced with a scan with combined filters

```
# create table t(id int primary key, x text);
# select * from t t1 join t t2 where t1.id = t2.id and t1.x like 'a%';
```

or

```
# create view v as select * from t where x like 'a%';
# select * from t where exists (select * from v where id = t.id);
```

Baseline	Join over scans on `t` and `v`
Optimized	Scan on `t` where `x like 'a%'`

Outer join

- Output all outer rows, nulls for inner rows when none match
- Less freedom for planning

- Can be reduced to inner join
  - when it follows from WHERE clause that some inner column is not null [before 10]
- Can be removed
  - Inner side is not used and is unique [before 10]
  - Inner side is not used and the result is made unique by GROUP BY or DISTINCT [DEV]

```
# create table t1 (id int primary key);

# create table t2 (id int primary key, b int not null);

# select t1.* from t1 left join t2 on t1.id = t2.id;

# select distinct t1.* from t1 left join t2 on t1.id = t2.b;

# select t1.id from t1 left join t2 on t1.id = t2.b group by t1.id;
```

DEV

Merge join on inequality

- Can do full joins
- Faster than Nested Loop

Plan	**Time, ms**
Merge Join over Sort	100
Nested Loop	880

```
# create table t(a) as select generate_series(1, 10000);

# select * from t t1 full join t t2 on t1.a < t2.a;
  ERROR:  FULL JOIN is only supported with merge-joinable
  or hash-joinable join conditions

# select * from t t1 join t t2 on t1.a < t2.a and t2.a < 1000;
```

DEV

Merge join on range overlap

- Normally performed with Nested Loop
- Order ranges by comparison operator
- Perform Merge Join on range overlap (&&)

```
# tables s, r(ir int4range) with
  r.ir = (g, g+10),
  s.ir = (g+5, g+15),
  g = 1..100k;

# gist(ir) on s and r;

# select * from s join r on s.ir && r.ir;
```

Plan	**Time, s**
Nested Loop over Seq Scan and Index Only Scan	15.7
Merge Join over Sort	4.3
Merge Join over btree Index Scan	2.8

Transform join to union

DEV

- Useful for aggregation over star schema joins

```
# create temp view denorm as
    select f.*, d1.t t1, d2.t t2
    from fact f
    left join dim d1 on f1=d1.id
    left join dim d2 on f2=d2.id;
```

```
# select count(*) from denorm
  where '1' in (t1,t2);
```

```
Aggregate
  -> Hash Join on (f.f2 = d2.s)
    -> Hash Join on (f.f1 = d1.s)
Execution time: 5.8 ms
```

```
# select count(*) from
(select * from denorm where '1'=t1
 union
 select * from denorm where '1'=t2);
```

```
Aggregate
 -> Hash Aggregate
  -> Append
   -> Nested Loop on (f.f2 = d2.s)
   -> Nested Loop on (f.f1 = d1.s)
Execution time: 0.6 ms
```

Precalculate stable and immutable functions

DEV

1. Cache stable functions in expressions at execution time

```
# select count(*) from messages where body_tsvector @@
to_tsquery('postgres');
```

- Calculate `to_tsquery` only once in Recheck step of Bitmap Heap Scan
- 1.5 s precalculated / 2.3 s baseline

2. Inline immutable functions in FROM list at planning time

```
# select count(*) from messages m, to_tsquery('english',
'postgres') qq
where m.body_tsvector @@ qq;
```

- Bitmap Heap scan instead of Nested Loop over Function Scan + Bitmap Heap scan
- No join => faster planning, better cost estimates

Support the development

- Review the patches you need
- No need to know Postgres internals or C programming
- Read “Reviewing a Patch” at the wiki
- Usability review
  - Is the feature actually implemented?
  - Do we want it?
  - Are there dangers?
- Feature test
  - Does it work as advertised?
  - Are there any corner cases?
- Performance review
  - Are there any slowdowns?
  - If the patch claims to improve the performance, does it?

Thank you!

Alexander Kuzmenkov
a.kuzmenkov@postgrespro.ru

PROFESSIONAL
Postgres

References

Loose index scan
https://www.postgresql.org/message-id/flat/707b6f68-16fa-7aa7-96e5-eeb4865e6a30%40redhat.com

Incremental sort
https://commitfest.postgresql.org/20/1124/

Estimate sort costs for GROUP BY
https://commitfest.postgresql.org/20/1706

Multicolumn join selectivity
https://www.postgresql.org/message-id/flat/3fcfd5e5-6849-34e6-22ab-1b62d191bedb%402ndquadrant.com#d61504c511d4b437505a05fa50047019

Self join on primary key
https://commitfest.postgresql.org/20/1712/

Unique outer join with GROUP BY
https://www.postgresql.org/message-id/flat/CAKJS1f96XNrS68NZy9s=Xkq+RAj6RE5CrCvDcy_uB-V=U4+YRw@mail.gmail.com

Merge join on inequality
https://commitfest.postgresql.org/19/1141/

Merge join on range overlap
https://commitfest.postgresql.org/17/1449/

Transform join to union
https://www.postgresql.org/message-id/flat/7f70bd5a-5d16-e05c-f0b4-2fdfc8873489@BlueTreble.com

Precalculate stable and immutable functions

1. https://commitfest.postgresql.org/20/1648/
2. https://commitfest.postgresql.org/19/1664/