On The Performance Of Database Query Processing Algorithms On Flash Solid State Disks

Daniel Bausch, Ilia Petrov, Alejandro Buchmann
Databases and Distributed Systems Group
Technische Universität Darmstadt
{bausch, petrov, buchmann}@dvs.tu-darmstadt.de

***Abstract*—Flash Solid State Disks induce a drastic change in storage technology that impacts database systems. Flash memories exhibit low latency (especially for small block sizes), very high random read and low random write throughput, and significant asymmetry between the read and write performance. These properties influence the performance of database join algorithms and ultimately the cost assumptions in the query optimizer.**

In this paper we examine the performance of different join algorithms available in PostgreSQL on SSD and magnetic drives. We observe that (a) point queries exhibit the best performance improvement of up to fifty times; (b) range queries benefit less from the properties of SSDs; (c) join algorithms behave differently depending on how well they match the properties of solid state disks or magnetic drives.

I. Introduction

Flash Solid-State Disks (SSDs) are revolutionizing the storage technology and have the potential to change established principles of DBMS architectures. Both Flash SSDs (by "SSDs" we mean enterprise-class, Flash NAND SLC SSD) and Hard Disk Drives (HDD) support the same block device interface standards. This makes substitution easy. SSDs exhibit low latency and very high random throughput, both 10 to 100 times better than the respective HDD values. The sequential throughput of an enterprise SSD is also high. Therefore, one might be tempted to believe that all the database performance issues are solved by simply replacing HDDs by SSDs. However, an intrinsic characteristic of Flash SSDs is strongly asymmetric throughput. This is especially true for random operations and there is a strong block size dependence. Furthermore, the random write performance degrades over time. All that combined with the low Flash SSD latency requires a re-evaluation of data processing algorithms.

In this paper we study the performance impact of the Flash SSD characteristics on different data processing algorithms and query plans executing on data stored on flash storage. We investigate past assumptions regarding the degree to which existing algorithms match the characteristics of SSDs. Some of these change, which clearly influences their cost function. For example, in addition to the changing random-to-sequential ratio, we need to account explicitly for random read and random write operations.

There are several works that investigate similar aspects in database query processing [1], [2], [3]. Graefe et al. [2], [3] focus on the impact of SSD characteristics on query processing in relational databases and especially on join processing. [2], [3] explore the impact of new page layouts – more suitable for SSDs and propose *FlashJoin* and *RARE-join* algorithms. Do and Patel [1] perform significant preliminary study on the influence of SSD performance on join algorithms. They consider block nested-loop join, sort-merge join, *Grace Hash join* and *Hybrid Hash join* on a single threaded embedded storage manager. In addition, they investigate the impact of the buffer size and page size on the performance. In the present paper we pursue a similar goal, however, we base our study on PostgreSQL and examine several different types of queries (point, range, and different join queries).

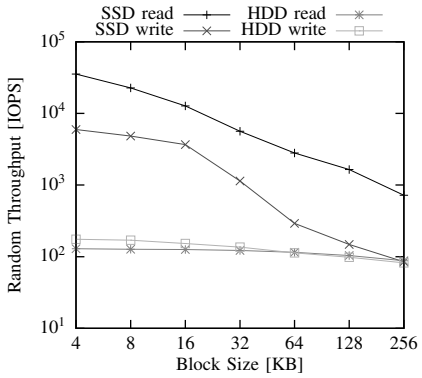
The contributions of this paper are: (a) we observe that not all algorithms benefit equally from the characteristics of Flash SSDs. Most of the algorithms for point queries have a huge performance gain, while range queries benefit less; (b) join algorithms exhibit a speed-up on SSDs of between five and fifteen times; (c) finally, we observe that over the set of join algorithms that was analyzed there is a non-uniform improvement, i.e certain algorithms are relatively faster on SSDs, while a few are relatively faster on HDDs.

The rest of the paper is organized as follows. We briefly introduce some basic characteristics of Flash SSDs in the next section. In Section IV we describe the experimental search space. Sections III and V describe the benchmark used and contain detailed discussions on the experimental results, respectively. Section VI summarizes our findings

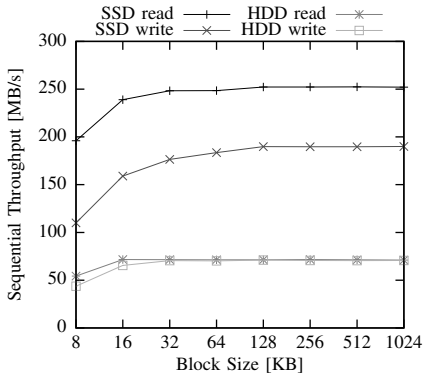
II. Enterprise Flash SSDs

The performance exhibited by Flash SSDs is significantly better than that of HDDs. Flash SSDs are not merely a faster alternative to HDDs; just replacing them does not yield optimal performance. The performance of Flash SSDs is characterized through: asymmetry, very high random throughput; high sequential performance; low latency; low power consumption. The basic characteristics of the Flash SSDs are well documented [4], [5], [6]. These can be summarized as follows: (a) asymmetric read/write performance

– the read performance is significantly better than the write performance, up to an order of magnitude. This is due to the internal organization of the NAND memory and FTL algorithms. (b) excellent random read throughput (IOPS) – especially for small block sizes (Figure 1(a). (c) acceptable random write throughput – small random writes are 5x to 10x slower than the random reads. The random throughput also deteriorates over the time. (d) very good sequential read/write transfer. Although it is still commonly assumed that HDDs have higher sequential throughput, however newer generations of SSDs perform significantly better (Figure 1(b)). (e) IO Parallelism and Command Queuing (CQ) allows several IO requests to be executed asynchronously and in parallel.



(a) Random throughput (IOPS)



(b) Sequential throughput (MB/s)

Figure 1. Random and Sequential throughput of an Intel X25-E SSD and a 7200 RPM HDD

III. Benchmark

The database created for these experiments contains two tables, connected through a foreign key relationship. One with random address records and one with pets that the people in the first table own. For complete coverage of the experimental search space, different layouts of the same database are used, in which the key columns are indexed or not. Unclustered B-Tree indices are used. The key values of the primary key of table "addresses" are monotonically increasing numbers. The values of the foreign key column in table "pets" are created randomly. To put a reasonable load on the system we created ten million address records and five million pets amounting to 1.7 GB.

Database schema:

ADDRESSES(id, name, str, no, town);

PETS(id, owner, animal, name);

Three classes of queries are defined and executed against the database (SQL in the appendix):

point query: This query computes a random value and selects a single record from the ADDRESSES table with this primary key. These keys were generated as SERIAL and can be indexed (unclustered B^+-tree) or not.

range query: This query selects full records for a random range of foreign keys from the PETS table. These were inserted as random values but may be indexed or not. On average this returns 25% of the PETS tuples.

join query: This query selects all columns from both tables where the pet is a dog—that is one third of the pets. The tables are joined on the primary key of the ADDRESSES table and the foreign key in the PETS table. The join type is "inner join".

IV. Exploring the Search Space

Our goal is to examine the performance of different data processing algorithms. Unfortunately there is no direct way to instruct PostgreSQL to execute a given algorithm. Our methodology is to define representative types of queries: point, range and join queries against a simple database (see Section III).

We generate a plethora of different execution plans in a controlled manner: the PostgreSQL configuration contains nine boolean variables (named "enable_*"), which can be used to suppress the use of certain alternatives (such as *indexscan* or *seqscan*) when set to "off". Internally this assigns a "high" cost value to the disabled algorithms. A tool explores all the 512 enable bit combinations for a given query. It utilizes PostgreSQL's "EXPLAIN" SQL command to query which plan it would choose and what are its associated costs. The outcome of the selection process depends on multiple factors, including the memory settings, the concrete queries (including contained constants), the presence of indices and the buffer sizes.

The complete search space, described by the 512 bit combinations, is significant, however there is an inherent room for reduction since not all bit-combinations result in distinct plans. We reduce the complete set to a minimal set of plans by employing the following criteria: (i) *Group by a hash value (md5) of the plan-definition.* This identifies the distinct plans. (ii) *Minimize the estimated costs.* That avoids bit combinations containing "disabled" bits. (iii) *Minimize the number of enable bits switched to "on"*. This raises the correlation of the enabled bits and the selected algorithms, and simplifies interpretation.

Table I
MINIMAL SET OF PLANS

#	query	seqscan	indexscan	bitmapscan	sort	nestloop	mergejoin	hashjoin	cost	
1	point			•					9.370000×10^0	
2	point		•						9.360000×10^0	
3	point	•							2.485714×10^5	
4	range		•						2.767864×10^6	
5	range	•							1.089996×10^5	
6	range	•		•					7.133092×10^4	
7	range			•					7.352792×10^4	
8	join	•			•		•		2.190674×10^6	
9	join	•						•	7.405776×10^5	
10	join		•			•			3.273128×10^7	
11	join	•				•			4.723343×10^{11}	
12	join	•	•		•		•		7.526443×10^5	
13	join		*	•		•			1.003332×10^{10}	*
14	join		•					•	1.909402×10^7	
15	join	•	•			•			1.453751×10^7	
16	join		•				•		1.871930×10^7	
17	join	•		•		•			1.512598×10^7	
18	join †	*	•					•	1.000090×10^{10}	*
19	join †	*	•		•		•		1.000074×10^{10}	*
20	join †	*	*			•			4.977020×10^{11}	*
21	join ‡	•	•			•			1.191672×10^8	
22	join ‡	•	•		*		•		1.002026×10^{10}	*
23	join ‡	*	•		•		•		1.002016×10^{10}	*
24	join ‡	*	•					•	1.001893×10^{10}	*
25	join ‡	*	*			•			5.875931×10^{11}	*
26	join ‡	•		•		•			1.263072×10^8	

[•] enabled algorithm; [∗] "disabled" bit used algorithm(s); [†] only primary key of first table is indexed; [‡] only foreign key in second table is indexed

The minimal set of plans comprises 26 experiments: 3 variants of the point queries, 4 variants of the range queries, and 19 variants of the join queries. Table I shows the settings along with the calculated costs.

Each row describes one experiment, with the cost value being an average of multiple executions. The columns marked with a bullet represent the elementary algorithms that are enabled in this experiment originally. As PostgreSQL does not guarantee that only these will be selected perfectly, we have consulted the logged plans and marked the additionally selected disabled algorithms with an asterisk. The concrete plans used for each of the experiments are listed in the appendix as they are printed by the PostgreSQL "EXPLAIN" command.

V. EXPERIMENTAL RESULTS

The final result set of the preliminary experiments was obtained on an Intel Core2Duo at 3GHz. As storage devices we used a conventional desktop 7200 RPM hard disk and an X25-E SSD from Intel. We use Ubuntu Linux with kernel 2.6.27-17-server and PostgreSQL version 8.4 with an I/O concurrency level of 8. Furthermore we repeated the experiments with a PostgreSQL 9.0 release candidate. Regardless of the extensions introduced there the present subset of results is unchanged. We performed experiments with varying amount of main memory and database buffer. Out of space reason we report the experimental results with a total of 384MB memory out of which 192MB were allocated to PostgreSQL. Such configuration is very I/O-intensive and loads the I/O subsystem, while correlating well to the total database size.

As part of each experiment we execute ten queries with randomized parameters for the point and the range query classes. For each join algorithm we perform three execution runs. Before each query run OS page cache is dropped. The average run times of the real execution of the queries are displayed in Table II and Figure 2.

All algorithms run faster on the SSD, but not all of them benefit equally, as can be seen in Figure 4. This would not influence the quality of the decision, as long as the best algorithm for a given task stays the best. However, this is not the case in this setup. For the range queries algorithm 5 is best for both HDD and SSD and for the point queries there is only a marginal difference. However, for the join queries algorithms 8 and 9 exchange their rank very prominently. Figure 3 shows an excerpt of the most interesting instances of such join algorithm behavior and their execution times on linear scale.

Table II
AVG. EXECUTION TIMES AND SPEED-UP

#	query	algorithm	HDD [s]	SSD [s]	speed-up
1	point	bitmapscan	8.10×10^{-2}	1.91×10^{-3}	42.35
2	point	indexscan	8.26×10^{-2}	1.70×10^{-3}	48.37
3	point	seqscan	2.35×10^1	6.02×10^0	3.90
4	range	indexscan	1.60×10^3	2.16×10^2	7.42
5	range	seqscan	7.41×10^0	2.11×10^0	3.50
6	range	bit/seq	3.22×10^1	1.30×10^1	2.48
7	range	bitmapscan	3.94×10^1	1.78×10^1	2.21
8	join	sort-merge	8.22×10^1	3.12×10^1	2.63
9	join	hash	1.61×10^2	2.06×10^1	7.79
10	join	nestloop	1.26×10^4	1.73×10^3	7.31
11	join	nestloop	*t/o* (> 4h)	*t/o* (> 4h)	
12	join	sort-merge	1.39×10^2	7.97×10^1	1.74
13	join	nestloop	1.26×10^4	1.63×10^3	7.77
14	join	hash	1.02×10^4	1.04×10^3	9.85
15	join	nestloop	2.64×10^4	2.09×10^3	12.60
16	join	merge	1.09×10^4	1.48×10^3	7.37
17	join	nestloop	2.64×10^4	2.09×10^3	12.60
18	join	hash	2.45×10^2	2.27×10^1	10.79
19	join	sort-merge	1.69×10^2	1.26×10^2	1.33
20	join	nestloop	*t/o* (> 8h)	*t/o* (> 4h)	
21	join	nestloop	1.03×10^4	1.32×10^3	7.80
22	join	sort-merge	1.01×10^4	1.42×10^3	7.10
23	join	sort-merge	1.01×10^4	1.44×10^3	6.99
24	join	hash	1.04×10^4	1.02×10^3	10.23
25	join	nestloop	*t/o* (> 8h)	*t/o* (> 4h)	
26	join	nestloop	1.03×10^4	1.38×10^3	7.46

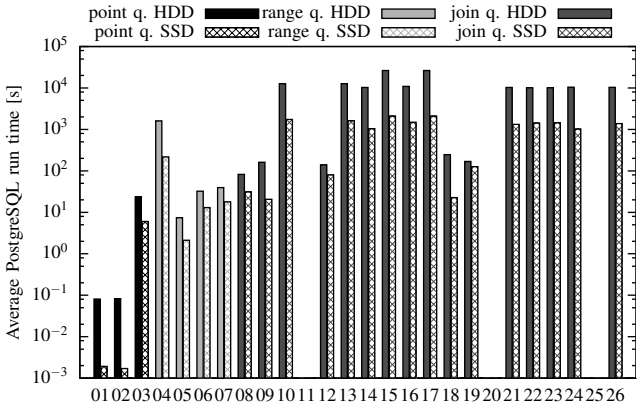


Figure 2. Average run times for one query

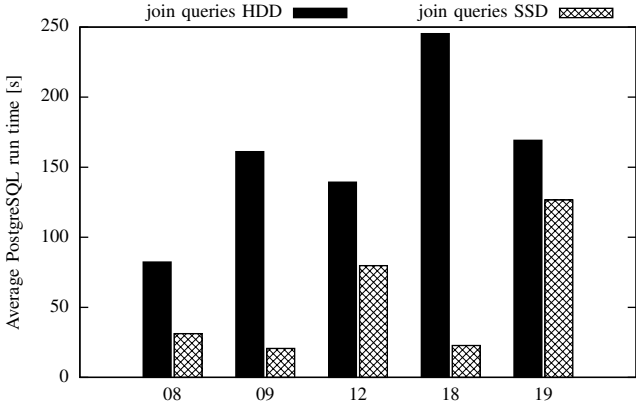


Figure 3. Close-up of the best join algorithms

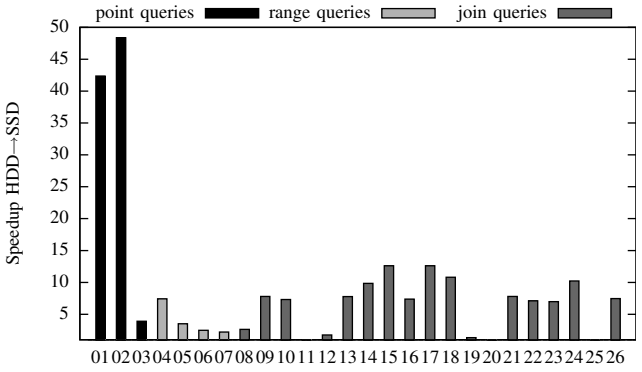


Figure 4. Speed-up of the average run times

Alternatively, consider algorithms 8 (sort-merge join) and 9 (hash join) in Figure 3. Both represent valid execution plans for the join query. Interestingly enough on SSDs the hash join execution plan (algorithm 9) has better performance and a worse one HDDs, whereas the sort-merge join (algorithm 8) has better relative performance on HDDs. We call this phenomenon an *exchange*. Explanations for such behavior will be given for the rest of the section. For reasons of brevity whenever we say that an algorithm is faster or slower we mean its relative performance depending on the device.

Point Queries. Considering the execution times of the point queries (Table II) we observe that the first two algorithms (algorithms 1 and 2) are comparably fast and profit by the use of the SSD with performance increases of over 40 times. The third algorithm exhibits a 5x performance increase. These differ in that the former algorithms use an index, while the latter (algorithms 3) performs a sequential scan. Index lookups are random read accesses, for which an SSD is a lot faster than an HDD, while the sequential reads of the third algorithm are fast on an HDD as well.

Range Queries. As for the range queries the pure *indexscan* (algorithm 4) logic is the slowest, but has the highest speed-up on SSDs. Fastest plan for this query is the sequential scan (algorithm 5) that ignores the index completely. This holds for HDD and SSD equally. Therefore while a single index lookup is much faster than a full sequential scan, a large amount of index lookups is slower than the full sequential scan.

Algorithms 6 and 7 use the bitmap index scan logic, which is slower than the pure index scan for the point query. In this case, however, the selectivity of the query is significantly higher (25% return on average) and the values are distributed randomly over the column. The pure index scan reads the index sequentially and the corresponding data tuples where the condition matches on the index producing a random read pattern on the table. The bitmap index scan scans the index and creates a bitmap in table order where matching tuples are marked. After that the table is scanned by only accessing marked tuples. Because of the high selectivity, this results in a nearly full sequential scan of the table. The difference of the runtimes between algorithms 5 and 7 is caused by reading the index and the creation of the bitmap. Algorithm 6 is special because whenever both *bitmapscan* and *indexscan* are enabled, the planner chooses based on the selectivity. If the number of tuples expected to be returned exceeds a certain level, it chooses a *seqscan*, otherwise it chooses the *bitmapscan*. We think, this strategy should be indeed better, but it depends on the choice of the threshold. The *seqscan* should be superior for even lower selectivities than the planner currently assumes.

Join Queries. The join query reveals 19 distinct plans. PostgreSQL provides three fundamental ways to do a join: nested loop join (*nestloop*), hash join (*hashjoin*) and (sort)-merge join (*mergejoin*). These basic join variants are complemented by the scan operators already discussed.

Nested Loop Joins. Considering the speed-up (Figure 4), we distinguish two groups of nested loop joins; one with a speed-up of about 7.5x (plans 10, 13, 21, 26), and another with a speed-up of about 12.5 (plans 15 and 17). While the latter two are more accelerated by the SSD, they are still slower than the others. The former group accesses the smaller “pets” table by index while the other group accesses

the bigger "addresses" table by index. Because of a bigger tuple size in "addresses" this results in more cache misses and therefore a greater number of random accesses, which are accelerated more strongly by replacing the HDD with an SSD. Unfortunately, PostgreSQL does not implement *block nested loop join*, which conceptually and according to [1] should exhibit a significant performance improvement.

Sort-Merge Joins. The merge joins can also be subdivided into two groups. Group one consists of the plans 08, 12, and 19. This group sorts by the randomized "pets.owner" column and accesses "addresses.id" by *seqscan* or *indexscan*, but both variants result in sequential accesses as the data in "addresses.id" was generated serially. As PostgreSQL does not use the provided parallelism of the SSD, this results only in a low speed-up. The other group consists of plans 16, 22, and 23. These plans scan pets by the foreign key index what results in random reads and so get accelerated by the use of an SSD.

Hash Joins. The hash joins have all good speed-up of about 8x to 12x. A first group of plans 9 and 18 performs a *seqscan* on the "pets" table and is very fast compared to another group (plans 14 and 24) which resort to an *indexscan* and are about two orders of magnitude slower. This is again caused by the random accesses from the foreign key scan.

The better performance of SSDs regarding random operations is also the cause for the exchange of the best algorithms. *Hashjoin* produces random writes which are more efficiently performed on SSDs than on HDDs, boosting the good plans of this algorithm in front of the good plans of *sort-merge join*, whose write pattern is more sequential, but which incorporate more expensive calculations.

VI. Conclusions

In this paper we examined the performance of different join algorithms available in PostgreSQL on SSD and magnetic drives. Firstly, we observe that point queries exhibit the best performance improvement of up to fifty times. Secondly, range queries benefit less from the properties of SSDs. The way they are evaluated at present takes little advantage of the good random properties and parallelism of SSDs. The best speed-up is observed from the *indexscan* algorithm, although that remains the slowest. Finally, the different join algorithms investigated here match the device (SSD or HDD) properties to a varying extent. We observe several algorithms having better relative performance on an HDD but which exhibit lower relative performance on an SSD or vice versa. For the tested query hash joins have better relative performance on SSDs and an inferior on HDDs, whereas the converse is true for sort-merge joins. Even in absence of block nested-loop approaches, nested-loop joins achieve a significant speed-up, if they incorporate *indexscan* in their inner loop. Variants using *seqscan* in the inner loop were still extremely slow, so we had to abort those experiments. Future research should target achieving more uniform speed-up across the different algorithms and more realistic cost estimates.

Acknowledgement

This work has been partially supported by the DFG project Flashy-DB.

References

[1] J. Do and J. M. Patel, "Join processing for flash SSDs: remembering past lessons," in *Proc. DaMoN 2009*, 2009.

[2] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query processing techniques for solid state drives," in *Proc. SIGMOD 2009*, 2009, pp. 59–72.

[3] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe, "Fast scans and joins using flash drives," in *Proc. DaMoN 2008*, 2008, pp. 17–24.

[4] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. of SIGMETRICS '09*, 2009, pp. 181–192.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX 2008*, 2008, pp. 57–70.

[6] V. Hudlet and D. Schall, "SSD != SSD – an empirical study to identify common properties and type-specific behavior," in *Proc. of BTW 2011*, 2011, pp. 430–441.

APPENDIX

PLANS OF POINT QUERIES

Query: `SELECT * FROM addresses WHERE id = ???;`

01	bitmapscan	`Bitmap Heap Scan on addresses`
		`  Recheck Cond: (id = ???)`
		`  -> Bitmap Index Scan on addresses_pkey`
		`  Index Cond: (id = ???)`

02	indexscan	`Index Scan using addresses_pkey on addresses` `  Index Cond: (id = ???)`
03	seqscan	`Seq Scan on addresses` `  Filter: (id = ???)`

PLANS OF RANGE QUERIES

Query: `SELECT * FROM pets WHERE owner >= ??? AND owner <= ???;`

04	indexscan	`Index Scan using pets_owner_idx on pets` `  Index Cond: ((owner >= ???) AND (owner <= ???))`
05	seqscan	`Seq Scan on pets` `  Filter: ((owner >= ???) AND (owner <= ???))`

06	bitmapscan	`Bitmap Heap Scan on pets`
		`  Recheck Cond: ((owner >= ???) AND (owner <= ???))`
		`  -> Bitmap Index Scan on pets_owner_idx`
		`  Index Cond: ((owner >= ???) AND (owner <= ???))`
		—or—
		`Seq Scan on pets`
		`  Filter: ((owner >= ???) AND (owner <= ???))`

07	bitmapscan	`Bitmap Heap Scan on pets`
		`  Recheck Cond: ((owner >= ???) AND (owner <= ???))`
		`  -> Bitmap Index Scan on pets_owner_idx`
		`  Index Cond: ((owner >= ???) AND (owner <= ???))`

PLANS OF JOIN QUERIES

Query:
```
SELECT * FROM addresses JOIN pets
  ON pets.owner = addresses.id AND species = 'dog';
```

08	mergejoin	`Merge Join` `  Merge Cond: (addresses.id = pets.owner)`
	sort	`  -> Sort` `  Sort Key: addresses.id`
	seqscan	`  -> Seq Scan on addresses`

	sort	`  -> Materialize`
		`  -> Sort`
		`  Sort Key: pets.owner`

	seqscan	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`
09	hashjoin	`Hash Join` `  Hash Cond: (pets.owner = addresses.id)`
	seqscan	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`
	seqscan	`  -> Hash` `  -> Seq Scan on addresses`
10	nestloop	`Nested Loop`
	indexscan	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (species = 'dog'::animal)`
	indexscan	`  -> Index Scan using addresses_pkey on addresses` `  Index Cond: (addresses.id = pets.owner)`
11	nestloop	`Nested Loop` `  Join Filter: (addresses.id = pets.owner)`
	seqscan	`  -> Seq Scan on addresses`

	seqscan	`  -> Materialize`
		`  -> Seq Scan on pets`
		`  Filter: (species = 'dog'::animal)`

12	mergejoin	`Merge Join` `  Merge Cond: (addresses.id = pets.owner)`
	indexscan	`  -> Index Scan using addresses_pkey on addresses`

	sort	`  -> Materialize`
		`  -> Sort`
		`  Sort Key: pets.owner`

	seqscan	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`
13	nestloop	`Nested Loop`
	indexscan*	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (species = 'dog'::animal)`

	bitmapscan	`  -> Bitmap Heap Scan on addresses`
		`  Recheck Cond: (addresses.id = pets.owner)`
		`  -> Bitmap Index Scan on addresses_pkey`
		`  Index Cond: (addresses.id = pets.owner)`

14	hashjoin	`Hash Join` `  Hash Cond: (pets.owner = addresses.id)`
	indexscan	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (species = 'dog'::animal)`
	indexscan	`  -> Hash` `  -> Index Scan using addresses_pkey on addresses`
15	nestloop	`Nested Loop`
	seqscan	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`
	indexscan	`  -> Index Scan using addresses_pkey on addresses` `  Index Cond: (addresses.id = pets.owner)`
16	mergejoin	`Merge Join` `  Merge Cond: (addresses.id = pets.owner)`
	indexscan	`  -> Index Scan using addresses_pkey on addresses`
	indexscan	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (pets.species = 'dog'::animal)`
17	nestloop	`Nested Loop`
	seqscan	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`

	bitmapscan	`  -> Bitmap Heap Scan on addresses`
		`  Recheck Cond: (addresses.id = pets.owner)`
		`  -> Bitmap Index Scan on addresses_pkey`
		`  Index Cond: (addresses.id = pets.owner)`

18	hashjoin	`Hash Join` `  Hash Cond: (pets.owner = addresses.id)`
	seqscan*	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`
	indexscan	`  -> Hash` `  -> Index Scan using addresses_pkey on addresses`
19	mergejoin	`Merge Join` `  Merge Cond: (pets.owner = addresses.id)`
	sort	`  -> Sort` `  Sort Key: pets.owner`
	seqscan*	`  -> Seq Scan on pets` `  Filter: (species = 'dog'::animal)`
	indexscan	`  -> Index Scan using addresses_pkey on addresses`
20	nestloop	`Nested Loop` `  Join Filter: (addresses.id = pets.owner)`
	indexscan*	`  -> Index Scan using addresses_pkey on addresses`

	seqscan*	`  -> Materialize`
		`  -> Seq Scan on pets`
		`  Filter: (species = 'dog'::animal)`

21	nestloop	`Nested Loop`
	seqscan	`  -> Seq Scan on addresses`

	indexscan	`  -> Index Scan using pets_owner_idx on pets`
		`  Index Cond: (pets.owner = addresses.id)`
		`  Filter: (pets.species = 'dog'::animal)`

22	mergejoin	`Merge Join` `  Merge Cond: (pets.owner = addresses.id)`
	indexscan	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (species = 'dog'::animal)`

	sort*	`  -> Materialize`
		`  -> Sort`
		`  Sort Key: addresses.id`

	seqscan	`  -> Seq Scan on addresses`
23	mergejoin	`Merge Join` `  Merge Cond: (addresses.id = pets.owner)`
	sort	`  -> Sort` `  Sort Key: addresses.id`
	seqscan*	`  -> Seq Scan on addresses`
	indexscan	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (pets.species = 'dog'::animal)`
24	hashjoin	`Hash Join` `  Hash Cond: (pets.owner = addresses.id)`
	indexscan	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (species = 'dog'::animal)`
	seqscan*	`  -> Hash` `  -> Seq Scan on addresses`
25	nestloop	`Nested Loop` `  Join Filter: (addresses.id = pets.owner)`
	indexscan*	`  -> Index Scan using pets_owner_idx on pets` `  Filter: (species = 'dog'::animal)`
	seqscan*	`  -> Materialize` `  -> Seq Scan on addresses`
26	nestloop	`Nested Loop`
	seqscan	`  -> Seq Scan on addresses`

	bitmapscan	`  -> Bitmap Heap Scan on pets`
		`  Recheck Cond: (pets.owner = addresses.id)`
		`  Filter: (pets.species = 'dog'::animal)`
		`  -> Bitmap Index Scan on pets_owner_idx`
		`  Index Cond: (pets.owner = addresses.id)`

* originally disabled algorithm