Making Cost-Based Query Optimization Asymmetry-Aware

Daniel Bausch
bausch@dvs.tu-darmstadt.de

Ilia Petrov
petrov@dvs.tu-darmstadt.de

Alejandro Buchmann
buchmann@dvs.tu-darmstadt.de

Technische Universität Darmstadt
Department of Computer Science
Databases and Distributed Systems Group
Hochschulstraße 10
64289 Darmstadt, Germany

ABSTRACT

The architecture and algorithms of database systems have been built around the properties of existing hardware technologies. Many such elementary design assumptions are 20–30 years old. Over the last five years we witness multiple new I/O technologies (e.g. Flash SSDs, NV-Memories) that have the potential of changing these assumptions. Some of the key technological differences to traditional spinning disk storage are: (i) asymmetric read/write performance; (ii) low latencies; (iii) fast random reads; (iv) endurance issues.

Cost functions used by traditional database query optimizers are directly influenced by these properties. Most cost functions estimate the cost of algorithms based on metrics such as sequential and random I/O costs besides CPU and memory consumption. These do not account for asymmetry or high random read and inferior random write performance, which represents a significant mismatch.

In the present paper we show a new asymmetry-aware cost model for Flash SSDs with adapted cost functions for algorithms such as external sort, hash-join, sequential scan, index scan, etc. It has been implemented in PostgreSQL and tested with TPC-H. Additionally we describe a tool that automatically finds good settings for the base coefficients of cost models. After tuning the configuration of both the original and the asymmetry-aware cost model with that tool, the optimizer with the asymmetry-aware cost model selects faster execution plans for 14 out of the 22 TPC-H queries (the rest being the same or negligibly worse). We achieve an overall performance improvement of 48% on SSD.

1. INTRODUCTION

Database systems, their architecture and algorithms are built around the I/O properties of the storage. In contrast to Hard Disk Drives (HDD), Flash Solid State Disks (SSD) exhibit fundamentally different characteristics: high random and sequential throughput, low latency and power consumption [3]. SSD throughput is asymmetric in contrast to magnetic storage, i.e. reads are significantly faster than writes. Random writes exhibit low performance, which also degrades over time. Interestingly enough many of those properties also apply to other novel I/O technologies such as NV-Memories [4].

Precise cost estimation for query processing algorithms is of elementary importance for robust query processing and predictable database performance. Cost estimation is the basis for many query optimization approaches. The selection of the 'best' query execution plan correlates directly with database performance.

Taking hardware properties properly into account is essential for cost estimation and query optimization, besides the consideration of data properties (data distribution, ratios, access paths) and intra- and inter-query parallelism. Cost functions were built on assumptions about hardware properties which are now 20–30 years old. Some of these change fundamentally with the advent of new I/O technologies.

Due to the symmetric read/write characteristics and the high random I/O cost of traditional spinning disks, the I/O behavior is approximated by counting sequential and random storage accesses and weighting them with different factors. Flash SSDs as well as other new I/O technologies exhibit: read/write asymmetry (different for sequential and random I/O); and very good random read performance, i.e. random and sequential read costs converge. Hence the following factors can yield incorrect cost estimation: (i) not distinguishing between reads and writes; (ii) not accounting for random writes; (iii) not accounting for I/O parallelism and read operations.

In a previous study [1], we found that the optimal plan to answer a query indeed can depend on the used storage technology. In the present paper we report about our efforts to improve query optimization in respect to storage technologies. We show incremental improvements to the cost model of the open source DBMS PostgreSQL. The improvements are derived from observable behavior of the query processing algorithms of that DBMS and are supported by theoretical considerations. Besides sequential and random patterns the new model distinguishes reading and writing making it *asymmetry-aware*. Additionally a tool based on an iterative heuristic was built to automatically find good base coefficients for the configurable parameters of cost models. To simulate the situation of data-intensive systems under heavy load the test system was setup with tight memory settings relative to the size of the data. With tuned param-

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA
Copyright 2012 ACM 978-1-4503-1445-9 ...$10.00.

eters the DBMS with the new model is able to perform a TPC-H derived workload faster than a vanilla PostgreSQL with equally intensively tuned cost model parameters.

The rest of the paper is organized as follows: after a discussion of related work (Section 2) we describe the details of the asymmetry-aware optimizer cost model (Section 3). The adapted cost functions for Sorting and Hash-Join are presented in Sections 3.3.1 and 3.3.2 respectively. The asymmetry-aware model is implemented in PostgreSQL[1] and tested with an open source benchmark build around a TPC-H schema and data [16, 13]. The experimental setup, its results, and their discussion follows in Section 4.

2. RELATED WORK

Query optimization has been a research topic in database systems right from the beginning [15]. There has been a very large body of literature on the topic; some of the survey works are [8, 2, 7]. All these works provide the basis for the present paper.

The work by Pelley et al. [14] treats topics similar to the ones considered in our previous work [1]. Pelley et al. measure different scan and join algorithms' performance with varying query selectivity. Their results show only a small selectivity range where the optimal algorithm for an HDD is sub-optimal for an SSD. From that observation they extrapolate the generalized conclusion that optimizers do not need to be made SSD-aware for practical means. Their work also relates to [6]. In [1] we explored a similar problem setting but arrived at the conclusion that different query execution plans are best suited for different hardware. The model presented in the present paper features additional degrees of freedom to represent properties of asymmetric storage devices.

3. ASYMMETRY-AWARE COST MODEL

As a basis the open source DBMS PostgreSQL is used. PostgreSQL features a cost-based query optimizer. Cost functions in that optimizer calculate frequencies of expected operations, weight them, and are summed up to a scalar cost value per alternative plan. The plan which received the lowest cost value is executed.

3.1 Behavior of the Query Executor

Before discussing PostgreSQL's cost model and the suggested changes, we briefly introduce the behavior of the query execution algorithms in this section, while we focus on the I/O behavior.

3.1.1 Scanners

PostgreSQL features four ways to scan data, which produce read-only access patterns of different distributions.

The *Sequential Scan* algorithm implements the "full table scan". Disk is accessed in natural order, i.e. sequentially.

The *Index Scan* algorithm accesses tables indirectly by looking up the tuples' locations within an index first. Conditions specified in the query are used to narrow the range of relevant index and table portions. This may produce random patterns depending on index-to-table correlations and/or the order the keys are looked up in a complex plan.

Then there is *Bitmap Scan*, a variant of *Index Scan* which first records all potentially matching locations as a bitmap. After that only those portions are scanned in on-disk order. Bitmaps originating from different conditions or different indexes on the same table can be bit-logically combined prior to scanning. The amount of randomness induced by this algorithm depends on the number of holes in the bitmap.

Finally, the *Tuple-ID Scan* algorithm handles the special case when tuples are explicitely requested by conditions of the form "ctid = ..." and "WHERE CURRENT OF" expressions. Its access pattern is unpredictable.

3.1.2 Sorting and Hash Join

The sort and hash-joining algorithms produce mixed write and read I/O, if their temporary data does not fit in the main memory slice available to a single algorithm.

PostgreSQL's sort algorithm is basically an implementation of a combination of *Heap Sort* and *Polyphase Merge Sort With Horizontal Distribution*, both found in [10]. Its first partitioning phase produces mainly sequential writes[2]. The multiple merge phases tend to produce more randomly targeted read and write accesses because space occupied by read pages is directly reused for new sorted runs. The amount of randomness, however, depends on whether the input data is pre-sorted or not. We will experimentally show this in Section 3.3.1.

Hash Join in PostgreSQL means *Hybrid Hash Join. Hybrid Hash Join* first splits the data of the inner table into mulitple batches which can be processed in RAM at a time. The first batch is held in RAM so it can be processed directly after splitting. The tuples going to secondary batches are appended to multiple temporary files, one for each batch, in parallel. The hash table for each batch is created in memory when the batch is processed. Tuples of the outer table whose join field hashes to secodary batches are postponed and written to temporary files, too. Although the tuples are strictly appended to the temporary batches, this produces a lot of random writes as we will experimentally show in Section 3.3.2.

3.1.3 Materialization and Re-Scanning

If the identical result of an execution plan sub-tree is needed multiple times by a parent node, it is materialized, i.e. temporarily stored. If the intermediate result fits in the memory slice available for an algorithm, it is held in RAM, otherwise it is stored sequentially to temporary on-disk storage. When the data is used again, it is sequentially streamed from disk.

3.2 Original Model

PostgreSQL's cost model is organized parallel to its individual query processing algorithms so there is a one-to-one relationship between algorithms and elementary cost functions. A complete mathematical transcript of the cost model and its changes can be found in Appendix A. We focus on the I/O part of the functions as the computational part was unchanged. Some functions are notated slightly different to the calculations in the program code and some technical specialities are left out for easier understanding. In this section we give an idea of how the shown behavior of the executor has been translated into a cost model by the PostgreSQL developers.

[1]`http://www.postgresql.org`

[2]The cost for sourcing the data is accounted using the appropriate formulas respective to the sourcing plan.

The distinction of I/O access patterns is implemented as configurable weight factors. As of writing PostgreSQL's cost model accounts for sequential and random accesses using two different factors.

The cost functions for the scan algorithms estimate the number of pages to be read for the different tasks within the algorithms. These numbers are multiplied with one of the two configurable weight factors depending on whether the respective operations are expected to produce sequential or random accesses. For *Sequential Scan* the model multiplies the number of pages of the scanned relation with the factor for sequential accesses; for *Index Scan* sophisticated computations are performed to convert the estimated selectivity and a statistics value about the index-to-table correlation into the number of required page reads and a fraction for the randomness of these accesses; for *Bitmap Scan* the number of holes which would lead to skips is approximated based on the selectivity, i.e. small result fractions are assumed to produce more random accesses; the unpredictability of the *Tuple-ID Scan* algorithm is treated by the worst case assumption: every tuple expected to be accessed is counted as a random page read; and, finally, *re-scanning* a previously materialized intermediate result is counted by multiplying the expected size with the parameter for sequential accesses. Interestingly, the same cost formula models the *materialization* itself.

The I/O cost of a *sort* operation is modeled by a typical $O(n \log n)$ formula. The number of input pages times 2 (for write and read) is multiplied with the expected number of sort phases, which is the logarithm to base of the merge order[3] of the number of expected intial sorted runs. This estimates the total number of page accesses, which is then weighted with $\frac{3}{4}$ sequential and $\frac{1}{4}$ random.

The I/O cost for writing and further processing of additional external batches of data in the *Hash Join* algorithm is accounted by multiplying two times the relevant data size with the parameter for sequential accesses.

3.3 Modifications

The proposed *asymmetric cost model* provides four configuration parameters regarding I/O instead of only two. These allow to distinguish not only sequential and random operations but also whether data is read or written. Each of the old parameters is split into a parameter representing read operations of the given access pattern and another parameter that represents write operations performed with the same pattern. In the cost functions the old parameters are replaced with the new ones according to the behavior of the algorithms. For easier comparison we restrict our model modifications to parameter replacements which allow us to reproduce the cost values of the old model by the new model using certain parameter settings.

The scanners perform pure read loads while *materialization* performs a pure write load; however, in both cases the associated cost functions can be converted to the new parameter set by substituting old parameter variables with new ones: "sequential read" and "random read" instead of just "sequential" and "random" for the scanners and "sequential write" instead of "sequential" for materialization.

The cost functions for the *sort* and *hashjoin* algorithms need more attention because their algorithms perform read loads as well as write loads while their original cost functions do not model those loads separately. Sections 3.3.1 and 3.3.2 will show in-detail how this was resolved.

3.3.1 Adapted cost function for sort

As we focus on the parts of the cost functions representing storage accesses we will look at the external sort only.

The original cost function for this algorithm assumed $\frac{1}{4}$ of the block accesses as random and $\frac{3}{4}$ as sequential, together representing all the reads and writes happening for this external sort. It is obvious that everything that is written to the temporary storage is read later again. Therefore we assume that half of the accesses were originally counted as reads and the other half as writes.

To get a realistic assigment for the random-to-sequential ratio, we traced the requests on the block layer of the operating system using *blktrace*[4]. This revealed the first line of the statistics shown in Table 1 for an external sort by the `l_partkey` column of the LINEITEM table of a TPC-H data set. In these statistics, a request is counted as sequential, if its start address is immediately following the end address of the preceding request.

Often query plans include sort operations carried out on data that is already stored in the requested order. The second row of Table 1, therefore, shows the statistics for a sort of the LINEITEM table by `l_orderkey`. In freshly loaded data, the LINEITEM table physically contains monotonically ascending order keys. Sorting by that column the shares of sequential and random operations exchange. Where the sorting of unordered data showed a high random share, there is now a high sequential share. We conclude, there is a high data dependency for the sort algorithm. As a compromise we assume that $\frac{1}{2}$ of the requests are sequential and $\frac{1}{2}$ are random. However, we count the first partitioning phase as only sequential writes, because in that phase the algorithm appends all new sorted runs to the end of the temporary file only.

Sorting in a join context.

A single execution of TPC-H query number 12 reveals the third line of Table 1 when it is answered using a sort-merge join with external sort. These numbers are similar to the ordered case above, and indeed, the main data portion that is sorted here is already ordered on disk. Only the smaller table (ORDERS) involved in the join really needs the sort operation while the other bigger table (LINEITEM) is already stored physically in the requested order. This further supports the assumptions we made for the simple sort case.

3.3.2 Adapted Cost Function for HashJoin

A second algorithm featuring mixed read and write operations is the *Hash Join* algorithm.

To determine the cost function for the *hashjoin* algorithm we conducted block tracing experiments very similar to the ones we did for the *sort* algorithm. The join from TPC-H query number 12 performed with PostgreSQL's hash join algorithm shows access patterns which can be summarized to the statistics shown in line four of Table 1. These numbers show a strong random write tendency and a fair sequential read tendency.

[3]The implementable merge order depends on the memory slice available to the algorithm.

[4]`http://git.kernel.dk/?p=blktrace.git`

Table 1: Temporary I/O Access Patterns

	write		read	
	sequential	random	sequential	random
external sort of unordered data	30.42%	69.57%	5.47%	94.52%
external sort of ordered data	77.15%	22.84%	95.89%	4.10%
sort-merge join	75.40%	24.59%	91.71%	8.28%
hash join	5.61%	94.38%	71.40%	28.59%

So partitioning produces random writes mostly. This can be explained as the filesystem (ext3) keeps the temporary batch files separated from each other by pre-allocation of space. This way individual batches are stored mainly consecutive, what in turn explains why reading them produces sequential reads most of the time.

As an approximation we therefore treat all read operations in the join phase as sequential accesses. The writes in the partitioning phase we treat as random as clearly indicated by the statistics. The resulting typical $O(n + m)$ formula is shown in the appendix.

4. EXPERIMENTAL ANALYSIS

In this section we will present the experiments we conducted to show the efficiency of the new model. In Section 4.1 we will define the system configuration and the used benchmark. Section 4.2 will illustrate the way we compare the different models. The results of the experiments are presented in Section 4.3 and discussed in Section 4.4.

4.1 System and load

For our experiments we used two identical computer systems. One of them was dedicated to a PostgreSQL installation using the modified cost model, while the other was installed with a vanilla PostgreSQL for reference. The used base version of PostgreSQL was 9.0.1. The systems were equipped with 1GB of main memory, a common 7200RPM hard disk for operating system and DBMS software, and a 64GB Intel X25-E SSD on which a partition of 40GB contained the database data including temporary files. Initially PostgreSQL was configured with *pgtune*[5] using the datawarehousing profile with up to 30 sessions (required for loading the data).

As workload we used the DBT-3 benchmark, which is an open source implementation of the TPC-H specification [16]. It is not validated by the *Transaction Processing Performance Council (TPC)*, so its results are not guarranteed to be comparable with validated TPC-H results. For the purposes of our experimental analysis the benchmark's well-defined schema and data set, as well as the standardized set of queries suffice. TPC-H models a data warehousing or decision support scenario, so most of the queries are highly demanding. We partially modified the original benchmark control scripts to fit the needs of our testing procedure.

Data was generated for scale factor 5. That is about 5 GB raw data. When loaded into the PostgreSQL database this inflates to about 13 GB through the addition of indexes and statistics. So less than 10% of that database fits in memory, which is a typical ratio in large scale analytical systems. We used the default page size of 8KB.

[5]`http://pgfoundry.org/projects/pgtune`

As performance metric we use the sum of the execution times of the benchmark's read only queries. However, we exclude the time to process query 15, because it contains a view creation, for which PostgreSQL cannot output a plan description, which is needed for our optimization. This sum of the 21 execution times will be called "total time" in the rest of the paper. We do not use TPC-H's original geometric mean based metric, because it is not suitable for speed-up calculation and generally harder to compare (see also [5]).

4.2 Calibration

The new cost model provides an extended set of configurable coefficients within its functions. A fair way to demonstrate the improved configurability and its positive impact on the processing performance is to test both models with their individual optimal configuration. Unfortunately, the optimal parameter settings cannot be computed analytically because they depend on algorithmic operations as well as on unknown operating system and hardware properties. Additionally, there is also a virtually infinite space in which the settings have to be found, so an exhaustive evaluation is impossible, too. To still find very good settings for both models, we used a heuristic based on *simulated annealing*[9]. We instructed that search algorithm to find the configuration that minimizes the "total time" of the DBT-3 benchmark.

The implemented search algorithm repeatedly changes the configuration settings and runs the load; it observes a response time, and if that is lower than the previous current execution time, it accepts the new settings; if the new response time is higher it randomly chooses whether to accept it nevertheless or discard it. The probability for the acceptance of an inferior setting decreases over time as well as the average strength of setting modification. To prevent the algorithm from getting caught in a non-global local optimum, the modification strength level and acceptance probability is reset after a fixed number of iterations and the algorithm starts again with the currently best known settings while having the chance to escape it with a big step. Figure 3 in the appendix shows the parameter settings as they vary over time and the corresponding total execution time for the first cycles of calibration. The repeated increase of variance in the curves is caused by the mentioned restarts. Modification of the configuration settings is performed by multiplication with a logarithmic normal-distributed random variable to respect the totally different magnitudes of the parameter values and to prevent autonomous drift.

For the final comparison of the models we executed the benchmark with 25 varying random seeds different from the seed used for calibration. The seed value essentially modifies the selectivity of the queries and sub-queries. Therefore the optimizer may consider different plans as being the optimal strategy to answer the given query templates. The

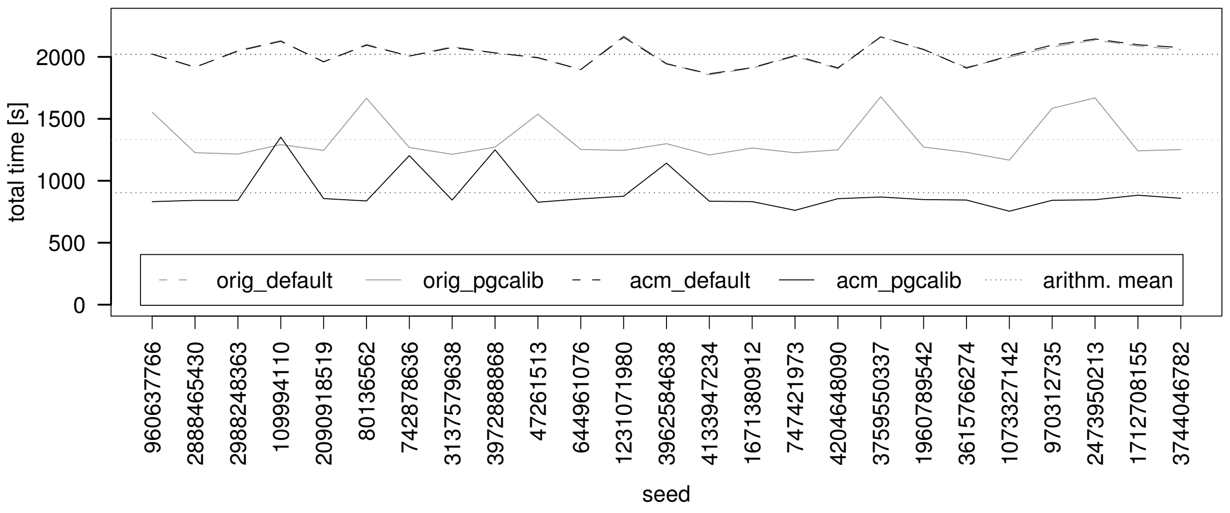

Figure 1: Total execution time of DBT-3 benchmark (excluding query 15) [orig = original model, default = initial pgtune'd configuration, acm = new asymmetric cost model, pgcalib = calibrated configuration that delivered minimal totaltime for a given model and the workload]. Different random seeds lead to different selectivities of the qualifications in the queries.

same 25 seeds are used to test (i) the original model with our data warehousing default configuration ('orig_default'), (ii) the calibrated original model ('orig_pgcalib'), (iii) the modified new model with a default configuration containing a converted set of the original weight factor settings ('acm_default'), and (iv) the new model with calibrated parameter settings ('acm_pgcalib').

4.3 Results

Figure 1 shows the total execution time for the different seeds, settings, and models. Finer grained information is shown in Figure 2 compares only the two calibrated configurations and shows the geometric mean of the speed-ups each query template received.

4.4 Discussion

From the chart in Figure 1 it is easily visible that the calibrated models both perform better than the same models with their initial ("default") configuration. Furthermore, the system with the calibrated new *asymmetric cost model* ("acm") completes the benchmark in even less time. There is a single case where the system with the new model seems to be a little bit slower and there are three additional peaks where the difference is not as prominent. Wilcoxon-Mann-Whitney's U-test [17, 12] computes a very very low probability (0.00001788%) that the results of both calibrated models might originate from the same distribution. The uncalibrated models, however, look very congruent and for these numbers the U-test delivers a probability of 20.99%. So one can not refute that they originate from the same distribution with a typical significance level of 5%.

The speed-up values seen in Figure 2 show that 14 out of 21 tested individual queries receive a positive speed-up while the performance of the others is only a litte bit decreased. The speed-up for the various query templates is very non-uniform. While query 21 gains more than 500% speed-up there are a lot of queries with much lower speed-up values and even one query whose speed is now 4% lower. Of the 21 queries (Q15 is excluded, see above), there are 10 queries that gain at least 5%. The average speed-up with respect to the total execution time of the 21 queries is 48%.

We did an in depth analysis of query 21 which had the highest relative speed-up and query 9 whose runtime difference was the biggest. Their detailed timing data reveals that in both cases the faster plan is doing index-nested-loop joins where the join attribute values change in the same order as the corresponding tuples are stored in the index-accessed table of the inner plan tree. In such a case the index scan results in sequential storage accesses. Although the slower plans used the same index, the join attribute values do not change in table order. In this case storage accesses are randomized what results in a reduced cache-hit rate. So the faster plan exploits a data-dependent inter-table correlation. Problematically, such inter-table correlations are not respected in PostgreSQL's optimizer at all. It only accounts for index-to-table correlations which are only relevant for range scans. Other reasons for speed-up are much less prominent or are hard to grasp because of unstable plan choices. Clearly device related reasons are thus hidden and it may be possible that the additional degrees of freedom got abused to compensate general optimizer deficiencies.

We performed a quick cross check on common HDDs. In a very time constrained experiment comprising only one cooling cycle of 100 iterations the calibration could not provoke a significant speed-up difference between the two models. With calibrated settings both systems perform the benchmark about 120% faster than with default settings. However, using the calibrated settings originally obtained for the SSDs, query 21 runs about 25 times faster (280s instead of 7227s) indicating that the optimal parameters for the HDDs were not found during this short calibration. With the same settings the whole benchmark is performed in 11895s using the old model and in 4975s using the new model – in both cases faster than with the quickly calibrated settings.

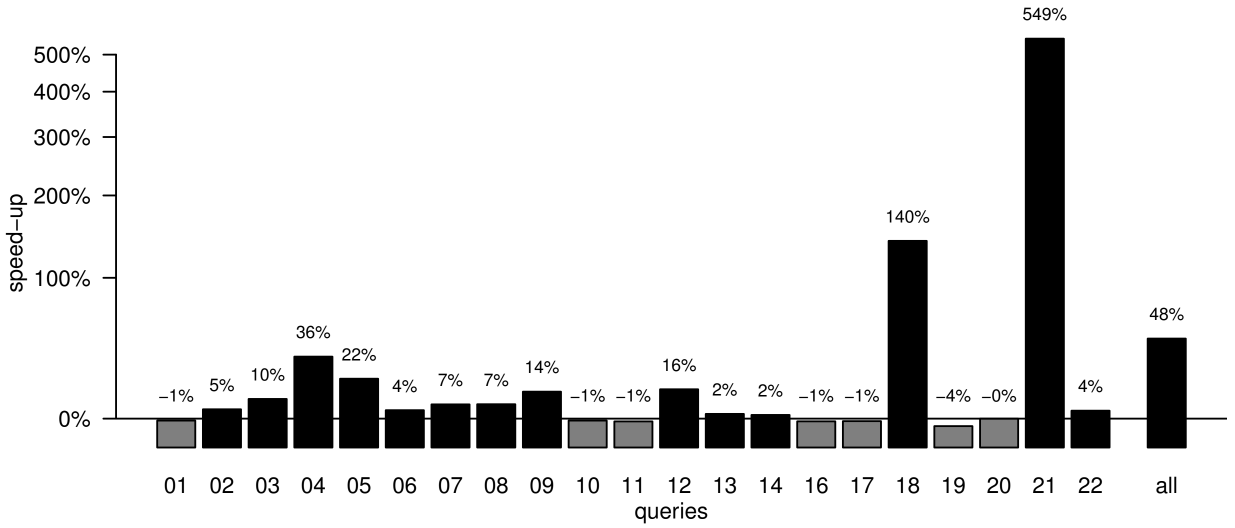

Figure 2: Speed-up from optimized old model to optimized new model per TPC-H query. [black = system with new model performs better, gray = system with old model performs better]

5. CONCLUSION

We presented a new cost model for PostgreSQL's cost-based query optimizer that distinguishes read and write operations. From our experimental results we conclude that this model can indeed deliver a higher performance if its weight parameters are configured to reflect the system and data properties. A tool was built and described that automates the configuration process.

We see significant performance improvements of an application class benchmark using the new model with calibrated parameters. However, we cannot relate the concrete plan changes observed in the experiments conducted so far to original SSD properties. The additional degrees of freedom available in the new model may be even useful to better tune the optimizer on systems based on common hard disks. With a simpler workload like the ones used in [1, 14] containing only data-dependencies respected by the used optimizer there might be a chance to explicitely demonstrate asymmetry-awareness. Such experiments are planned as a future work.

6. ACKNOWLEDGEMENTS

This work has been partially supported by the DFG project Flashy-DB. We thank Steven Pelley and Thomas Wenisch for their detailed and constructive critique of our work.

7. REFERENCES

[1] D. Bausch, I. Petrov, and A. Buchmann. On the performance of database query processing algorithms on flash solid state disks. In *FlexDBIST'11*, 2011.

[2] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS '98*, pages 34–43. ACM, 1998.

[3] 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*, pages 181–192, 2009.

[4] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CDIR'11*, Asilomar, CA, Jan. 2011.

[5] A. Crolotte. Issues in benchmark metric selection. *LNCS*, 5895:146–152, 2009.

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

[7] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.

[8] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28:121–123, March 1996.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[10] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[11] L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: A validated I/O model. *ACM Trans. Database Syst.*, 14(3):401–424, 1989.

[12] H. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[13] OSDLDBT Project. Database test 3 (dbt-3). `http://osdldbt.sourceforge.net`.

[14] S. Pelley, T. F. Wenisch, and K. LeFevre. Do query optimizers need to be ssd-aware? In *ADMS'11*, 2011.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. SIGMOD '79*, pages 23–34. ACM, 1979.

[16] Transaction Processing Performance Council. TPC benchmark H – standard specification. `http://www.tpc.org/tpch`.

[17] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

APPENDIX

A. TRANSCRIPT OF COST MODEL

This appendix contains a mathematical transcript of the cost model. These formulas are not required to understand the presented work as there are enough explanations in the main matter. Nevertheless, we include them for completeness or a brief overview. Table 2 displays the meaning of the used mathematical symbols.

Table 2: Notation

notation	meaning
c_{io}	compound cost value for I/O operations
$\hat{c}_{\mathrm{io}}$	compound I/O cost specific to index type
$\mathbf{c_s}$	cost weight factor for sequential accesses
$\mathbf{c_r}$	cost weight factor for random accesses
$\mathbf{c_{sw}}$	cost weight factor for sequential page writes
$\mathbf{c_{rw}}$	cost weight factor for random page writes
$\mathbf{c_{sr}}$	cost weight factor for sequential page reads
$\mathbf{c_{rr}}$	cost weight factor for random page reads
$\dot{\mathbf{c}}_{\mathbf{cpu}}$	CPU cost for loading a single tuple
$\hat{\mathbf{c}}_{\mathbf{cpu}}$	additional CPU cost for index based acceses
$\mathbf{c_{op}}$	CPU cost for applying an operator
$\mathbf{e}$	total cache available to database
$\left\Vert\cdot\right\Vert_{\mathrm{t}}$	number of tuples in an object
$\left\Vert\cdot\right\Vert_{\mathrm{p}}$	number of pages required to store an object
$\left\Vert\cdot\right\Vert_{\mathrm{l}}$	number of entries in the argument array
$\mathcal{R}$	all physical tables of database
R	base relation associated with algorithm
O	sibling outer relation in a nested loop join
P_i	relation produced by the inner deeper path
P_o	relation produced by the outer deeper path
S	to be sorted data
$\hat{r}$	correlation coefficient of index to table
$\hat{s}$	selectivity of index associated quals
$\hat{s}_B$	selectivity of index boundary quals
m	the *merge order* of the *polyphase merge sort*
n	the expected number of initial sorted runs

variable names shown in **bold** face are user configurable

For *sequential scan*, *index scan*, *bitmap scan*, *tuple-ID scan*, and *materialization* we show only the formulas of the original model and a replacement map (Table 3) because these formulas do not change structurally, i.e. only symbol names change. For *sort* and *hash join* contrarily we will show both, the old and the new function, at length.

Table 3: Parameter replacements in cost functions

function	original	new	affected equations
seqscan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(1)
indexscan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(4)
	$\mathbf{c_r}$	$\mathbf{c_{rr}}$	(3), (4), (16)
bitmapscan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(13)
	$\mathbf{c_r}$	$\mathbf{c_{rr}}$	(13), (16)
tidscan	$\mathbf{c_r}$	$\mathbf{c_{rr}}$	(25)
material	$\mathbf{c_s}$	$\mathbf{c_{sw}}$	(31)
rescan	$\mathbf{c_s}$	$\mathbf{c_{sr}}$	(32)
sort	*more complex, see Section 3.3.1*		
hashjoin	*more complex, see Section 3.3.2*		

A.1 Sequential Scan

Cost of a sequential scan is trivial: sequential page (read) cost times number of pages.

$$c_{\mathrm{io}}(\mathit{seqscan}) = \mathbf{c_s} \left\Vert R\right\Vert_{\mathrm{p}} \tag{1}$$

A.2 Index Scan

I/O cost of the index scan is calculated by a rather complex system of formulas. Trivially speaking, they interpolate between the sequential and random cost factors based on selectivity and index-to-table order correlation.

$$c_{\mathrm{io}}(\mathit{indexscan}) = \hat{c}_{\mathrm{io}} + \overline{c}_{\mathrm{io}} + \hat{r}^2\left(\underline{c}_{\mathrm{io}} - \overline{c}_{\mathrm{io}}\right) \tag{2}$$

$$\overline{c}_{\mathrm{io}} = \mathbf{c_r} \cdot \hat{p}(i_o t) \cdot i_o^{-1} \tag{3}$$

$$\underline{c}_{\mathrm{io}} = \begin{cases} \mathbf{c_r} \cdot \hat{p}\Big(\left\lceil i_o \hat{s} \left\Vert R\right\Vert_{\mathrm{p}}\right\rceil\Big) \cdot i_o^{-1} & \text{if } i_o > 1 \\ \mathbf{c_r} + \mathbf{c_s} \cdot \Big(\left\lceil \hat{s} \left\Vert R\right\Vert_{\mathrm{p}}\right\rceil - 1\Big) & \text{else} \end{cases} \tag{4}$$

$$t = \mathrm{round}\big(\max\big(1.0,\, \hat{s}\left\Vert R\right\Vert_{\mathrm{t}}\big)\big) \tag{5}$$

$$i_o = \max\big(\left\Vert O\right\Vert_{\mathrm{t}},\, 1\big) \tag{6}$$

To estimate the number of page fetches required for the index scan the function rooted at Equation (7) from [11] is used. (In the original paper $\hat{p}$ is named Y_{APP}. PostgreSQL replaces the product Dx by `tuples_fetched` which is t in our symbol symbol system.)

$$\hat{p}(t) = \begin{cases} \min\Big(p_t,\, \left\Vert R\right\Vert_{\mathrm{p}}\Big) & \text{if } \left\Vert R\right\Vert_{\mathrm{p}} \le e_R \\ p_t & \text{else if } t \le p_e \\ e_R + (t - p_e)\,\frac{\left\Vert R\right\Vert_{\mathrm{p}} - e_R}{\left\Vert R\right\Vert_{\mathrm{p}}} & \text{else} \end{cases} \tag{7}$$

$$p_t = \frac{2\left\Vert R\right\Vert_{\mathrm{p}} t}{2\left\Vert R\right\Vert_{\mathrm{p}} + t} \tag{8}$$

$$p_e = \frac{2\left\Vert R\right\Vert_{\mathrm{p}} e_R}{2\left\Vert R\right\Vert_{\mathrm{p}} - e_R} \tag{9}$$

$$e_R = \mathbf{e} \cdot \frac{\left\Vert R\right\Vert_{\mathrm{p}}}{\left\Vert \mathcal{R}\right\Vert_{\mathrm{p}} + \left\Vert I\right\Vert_{\mathrm{p}}} \tag{10}$$

A.3 Bitmap (Index) Scan

The I/O cost function of the *bitmap (index) scan* treats off the random and sequential factors based on the fraction of pages estimated to be needed from the relation. The function $\hat{p}(\cdot)$ from Equation (7) is reused in this context.

$$c_{\mathrm{io}}(\mathit{bitmapscan}) = \overbrace{\hat{c}_{\mathrm{io}}}^{\text{startup}} + pc' \tag{11}$$

$$p = \begin{cases} \min\left(\frac{\hat{p}\left(t\left\Vert O\right\Vert_{\mathrm{t}}\right)}{\left\Vert O\right\Vert_{\mathrm{t}}}, p_R\right) & \text{if } \left\Vert O\right\Vert_{\mathrm{t}} > 1 \\ \min\left(\frac{2p_R t}{2p_R + t}, p_R\right) & \text{else} \end{cases} \tag{12}$$

$$c' = \begin{cases} \mathbf{c_r} - (\mathbf{c_r} - \mathbf{c_s})\sqrt{\frac{p}{p_R}} & \text{if } p \ge 2 \\ \mathbf{c_r} & \text{else} \end{cases} \tag{13}$$

$$p_R = \max\Big(\left\Vert R\right\Vert_{\mathrm{p}}, 1\Big) \tag{14}$$

$$t = \mathrm{round}\big(\max\big(1.0,\, \hat{s}\left\Vert R\right\Vert_{\mathrm{t}}\big)\big) \tag{15}$$

A.4 General B$^+$-Tree Functions

PostgreSQL supports different kinds of index structures. Therefore the cost functions for both index based algorithms call an abstract function to calculate the cost for accessing only the index structures. In the used version of PostgreSQL, all included index types base their cost function on a generic index cost function. Therefore we only present a cost function for the default B$^+$-tree index as it contains everything relevant for the other types of index, too.

$$\hat{c}_{\text{io}} = \begin{cases} \frac{p\mathbf{c_r}}{i_o} & \text{if } i > 1 \\ p_I\,\mathbf{c_r} & \text{else} \end{cases} + p_I \frac{\mathbf{c_r}}{100000} \tag{16}$$

$$i_o = \max\left(\|O\|_{\text{t}}, 1\right) \tag{17}$$

$$i = i_{sa} \cdot i_o \tag{18}$$

$$i_{sa} = \prod_{a \in \hat{Q}_{sa}} \|a\|_{\text{l}} \tag{19}$$

$$\hat{Q}_{sa} := \{q \in \hat{Q} \mid q \text{ is a scalar array operator}\} \tag{20}$$

$$p_I = \begin{cases} \left\lceil t_I \frac{\|I\|_{\text{p}}}{\|I\|_{\text{t}}} \right\rceil & \text{if } \|I\|_{\text{p}} > 1 \wedge \|I\|_{\text{t}} > 1 \\ 1 & \text{else} \end{cases} \tag{21}$$

$$t_I = \begin{cases} 1 & \begin{array}{l}\text{if uniqe index } \wedge \\ \text{only `=' quals } \wedge \\ \text{no scalar array op } \wedge \\ \text{no null test}\end{array} \\ \text{round}\left(\hat{s}_B \frac{\|R\|_{\text{t}}}{i_{sa,B}}\right) & \text{else} \end{cases} \tag{22}$$

$$i_{sa,B} = \prod_{a \in \hat{Q}_{sa,B}} \|a\|_{\text{l}} \tag{23}$$

$$\hat{Q}_{sa,B} := \{q \in \hat{Q}_{sa} \mid q \text{ is an index boundary qual}\} \tag{24}$$

A.5 Tuple-ID Scan

The cost function for *tuple-ID scan* counts for every tuple access one page access. However, if it is combined with a "scalar array operator", e.g. an "in (...)" expression, then it counts a page access for every array entry.

$$c_{\text{io}}(\textit{tidscan}) = t\mathbf{c_r} \tag{25}$$

$$t = \sum_{q \in Q} \begin{cases} \|q\|_{\text{l}} & \text{if } q \in Q_{sa} \\ 1 & \text{if } q \in Q_{co} \cup Q_{ctid} \\ 0 & \text{else} \end{cases} \tag{26}$$

$$Q : \text{all quals associated to the tidscan} \tag{27}$$

$$Q_{sa} := \{q \in Q \mid q \text{ is a scalar array qual}\} \tag{28}$$

$$Q_{co} := \{q \in Q \mid q \text{ is a 'current of' expr}\} \tag{29}$$

$$Q_{ctid} := \{q \in Q \mid q \text{ is a 'ctid = ...' expr}\} \tag{30}$$

A.6 Materialization and Re-Scan

Writing out an intermediate result and re-reading it from disk is counted with the same formula in the original model. However, after applying the replacements from Table 3 *materialization* uses the "sequential write" factor while *re-scan* uses the "sequential read" factor.

$$c_{\text{io}}(\textit{material}) = \|P_i\|_{\text{p}}\,\mathbf{c_s} \tag{31}$$

$$c_{\text{io}}(\textit{rescan}) = \|P_i\|_{\text{p}}\,\mathbf{c_s} \tag{32}$$

A.7 Sort

The I/O cost of PostgreSQL's external sort is computed by a typical $O(n \log n)$ formula. The leading factor in its original form accounts for writing plus reading. In the new asymmetry-aware (rw) form this is more differentiated.

The term $\lceil log_m n \rceil$ estimates the number of required sort phases, where m is the merge order and depends on the available memory. The number of expected initial sorted runs n was previously calculated under the assumption that the initial sorted runs grow to two times the memory slice available to the algorithm. They are generated by a heap sort. As all tuples need to be sorted before the algorithm can output the first, everything is counted as *startup cost.*

$$c_{\text{io}}(\textit{sort}) = \overbrace{2\,\|S\|_{\text{p}} \lceil log_m n \rceil \left(\frac{3}{4}\mathbf{c_s} + \frac{1}{4}\mathbf{c_r}\right)}^{\text{startup}} \tag{33}$$

$$\begin{aligned} c_{\text{io}}(\textit{sort})_{rw} = & \overbrace{\|S\|_{\text{p}} \left(\mathbf{c_{sw}} + \left(\lceil log_m n \rceil - 1\right) \frac{\mathbf{c_{sw}} + \mathbf{c_{rw}}}{2}\right.}^{\text{startup}} \\ & \underbrace{\left. + \lceil log_m n \rceil \frac{\mathbf{c_{sr}} + \mathbf{c_{rr}}}{2}\right)}_{\text{startup}} \end{aligned} \tag{34}$$

A.8 Hash Join

The cost function accounts for writing and reading all data one time each. In the original form accesses are counted as sequential, which does not match the real behavior. Our asymmetry-aware form corrects that and counts the write operations for batching as random writes.

$$c_{\text{io}}(\textit{hashjoin}) = \overbrace{\|P_i\|_{\text{p}}\,\mathbf{c_s}}^{\text{startup}} + \left(\|P_i\|_{\text{p}} + 2\,\|P_o\|_{\text{p}}\right)\mathbf{c_s} \tag{35}$$

$$\begin{aligned} c_{\text{io}}(\textit{hashjoin})_{rw} = & \overbrace{\|P_i\|_{\text{p}}\,\mathbf{c_{rw}}}^{\text{startup}} + \|P_i\|_{\text{p}}\,\mathbf{c_{sr}} \\ & + \|P_o\|_{\text{p}} \left(\mathbf{c_{rw}} + \mathbf{c_{sr}}\right) \end{aligned} \tag{36}$$

B. CALIBRATED PARAMETER SETTINGS

Table 4 shows the parameters settings used for the experiments. They were obtained using the calibration algorithm described in Section 4.2.

Table 4: Optimal settings determined by calibration

old model	new model
$\mathbf{c_s} = 1.00000$	$\mathbf{c_{sr}} = 1.00000$
	$\mathbf{c_{sw}} = 49.91840$
$\mathbf{c_r} = 6.77405$	$\mathbf{c_{rr}} = 5.62724$
	$\mathbf{c_{rw}} = 19.08421$
$\dot{\mathbf{c}}_{\mathbf{cpu}} = 0.00121$	$\dot{\mathbf{c}}_{\mathbf{cpu}} = 0.00003$
$\dot{\hat{\mathbf{c}}}_{\mathbf{cpu}} = 0.03658$	$\dot{\hat{\mathbf{c}}}_{\mathbf{cpu}} = 0.01608$
$\mathbf{c_{op}} = 0.00016$	$\mathbf{c_{op}} = 0.00008$

For symbol explanation see Table 2.

Appendix B.1 shows a graphical plot of the first 1000 calibration cycles. The parameters shown in Table 4, however, are (one of) the best seen configurations after about 11000 cycles.

B.1 Calibration Dynamics

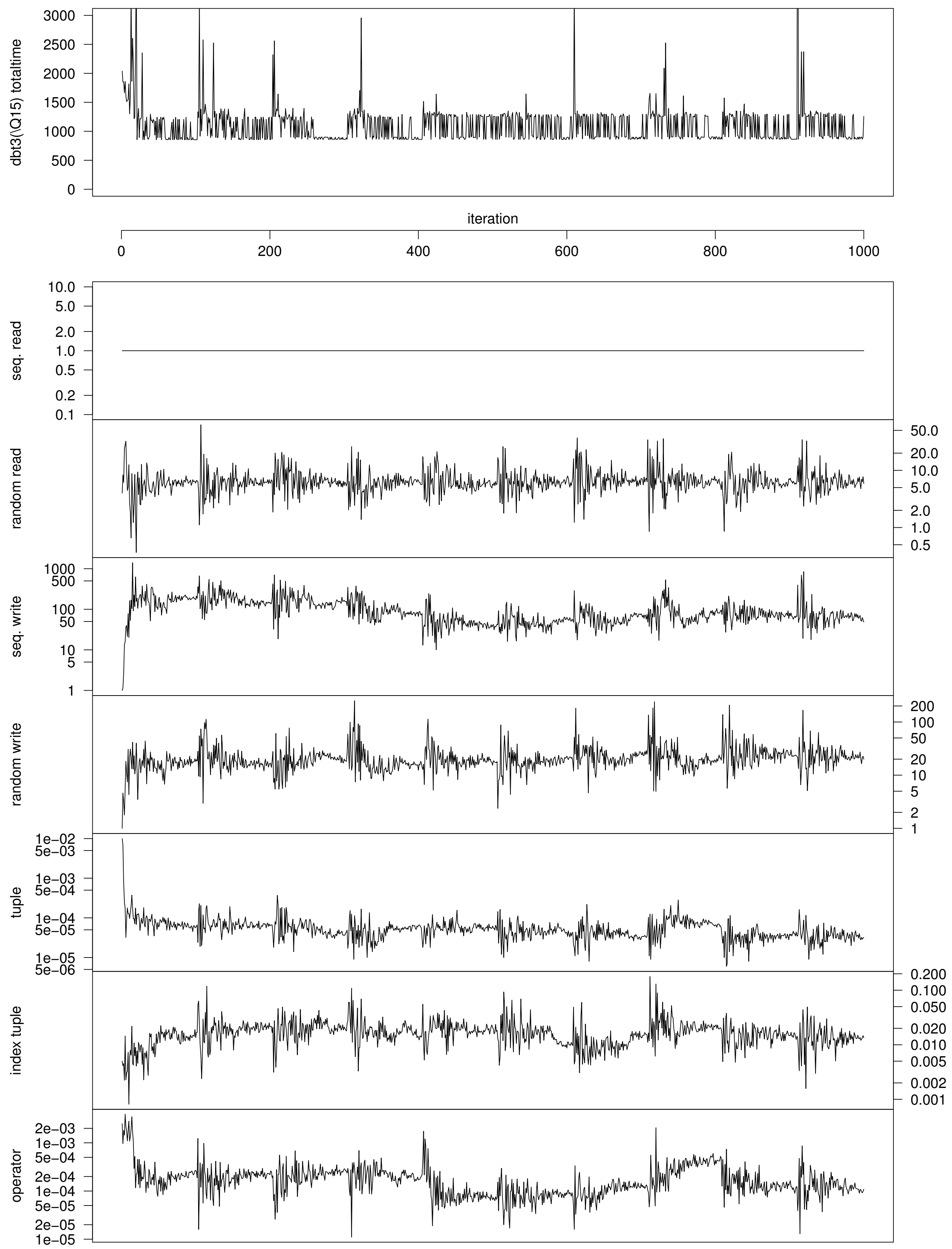

Figure 3: First 1000 iterations of calibration of new model