A Critical Look at the TAB Benchmark for Physical Design Tools

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Abstract

There has recently been considerable research on physical design tuning algorithms. At the same time, there is only one published methodology to evaluate the quality of different, competing approaches: the TAB benchmark. In this paper we describe our experiences with TAB. We first report an experimental evaluation of TAB on our latest prototype for physical design tuning. We then identify certain weakness in the benchmark and briefly comment on alternatives to improve its usefulness.

1 Introduction

Lately there has been considerable effort in the database community on reducing the total cost of ownership of database installations. Specifically, physical design tuning has become relevant, and most vendors nowadays include automated tools to tune database physical designs as part of their products (e.g., [3, 10, 14]). Given a query workload W and a storage budget B, these tools find the set of physical structures (or configuration) that fits in B and results in the lowest cost for W (see Figure 1).

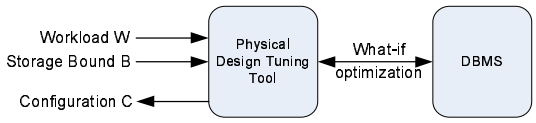


Figure 1: Architecture of Physical Design Tools.

Although there has been considerable research in new algorithms to find good configurations and extensions to newer physical structures (e.g., [2, 4, 5, 7, 8, 11, 12, 15]), much less attention has been paid on methodologies to evaluate the quality of different approaches. Specifically, we are aware of only one publication that proposes a benchmark of physical design tools: the Toronto Autonomic Benchmark, or TAB for short [9]. In this paper we describe our experiences with TAB when evaluating the quality of different alternatives, both in the context of a shipping product [3] and also on different experimental prototypes that we implemented over the last few years [5, 6]. Specifically, in Section 2 we review the TAB benchmark [9]. In Section 3 we report an experimental evaluation of TAB. Then, in Section 4 we analyze both the results of the experimental evaluation and also the benchmark itself. In doing so, we identify certain weaknesses in the design of TAB (specifically, on the benchmark metrics, the choice of baseline configurations, and some combination of database/workloads) and briefly comment on alternatives to mitigate their impact.

2 The *TAB* Benchmark

Reference [9] introduces a framework to evaluate the quality of automated physical design tuners, which we refer to as *TAB*. We next review the three components of the benchmark: the evaluation metrics, a baseline configuration to compare against recommendations, and the set of databases/workloads to tune.

Evaluation Metric Consider a workload W over a database D, and suppose that a tuner recommends configuration C for W. *TAB* evaluates the quality of C using $\mathcal{M}_{C,W}$, which returns, for an input time t, the number of queries in W that executed faster than t:

$$\mathcal{M}_{C,W}(t) = \frac{|\{q \in W : cost(q, C) \leq t\}|}{|W|}$$

where $cost(q, C)$ is the actual execution time of query q under configuration C. For pragmatic purposes, a timeout T_{max} is chosen and $cost(q, C)$ is capped by T_{max}. Therefore, it is always $\mathcal{M}_{C,W}(T_{max}) = 1$.

Baseline Configuration *TAB* identifies a special configuration, called $1C$, which consists of all single-column indexes over the database tables. Reference [9] justifies the choice of $1C$ by stating that *"... the consistently good performance of the single column configuration suggests a practical improvement of DBMS configuration recommenders..."*, *"...1C was also far better than the configurations recommended by both systems..."*, and *"...a very conservative overall workload assessment results in 1C producing almost 17 times better results than R!"*.

Databases and Workloads *TAB* uses two databases. The first one is a publicly available non-redundant reference protein database [13], or *NREF* for short, which

provides a collection of protein sequence data from several genome sequencing projects. The second one is the TPC-H benchmark used to evaluate the performance of database systems [1]. The workloads in [9] are chosen to *"...represent fragments of typical iceberg queries, that is, queries that compute aggregate functions over a set of attributes to find aggregate values satisfying certain conditions, grouped in different ways"*. A typical query for the reference protein database is shown below[1]:

```
SELECT T1.nref_id, COUNT(DISTINCT T2.nref_id)
FROM taxonomy T1, taxonomy T2, protein P
WHERE T1.taxon_id = T2.taxon_id AND
      T1.nref_id = P.nref_id AND
      P.p_name = 'Phosphotransferase'
GROUP BY T1.nref_id
```

For the TPC-H database, *TAB* does not use the QGen workload of [1], but rather one that mimics that of *NREF*.

3 Running *TAB*

We now report an experimental evaluation of TAB in our physical design tuning prototype based on [5]. Our objective with this experiment was two-fold. First, we wanted to analyze the performance of our prototype design tuner and compare its quality with the baseline configuration of [9]. Second, we wanted to understand the design decisions behind TAB, and question whether there was room for improvement in the benchmark definition itself. We used a Intel Xeon 3.2 GHz CPU with 2GB of RAM (we allocated 1GB of RAM to the DBMS for the experiments) and a 250GB, 7200rpm hard drive to store data. We used Microsoft SQL Server 2005 as the database engine. For each workload, we proceeded as follows. Following [9], we first created three copies of the original database, and deployed a different configuration on each instance. The first one, which we denote by adding a suffix *-P* to the database name, has only primary indexes. The second one, which we denote by adding a suffix *-P1C* to the database name, additionally contains all valid single-column indexes[2]. The third one, which we denote by adding a suffix *-R* to the database name, is obtained by running our physical design tuning tool for the input workload considering both clustered and non-clustered indexes with a storage bound equal to the size of the *-P1C* configuration. Table 1 shows statistics on the databases and workloads. (Note that we also evaluated the original TPC-H workload generated using the QGen utility.)

To avoid external factors in skewing the results, we performed the following additional steps. First, we stopped all non-essential operating system services to avoid interference. Second, we defragmented both the disk where data resided and also the indexes inside the database. Third, we created the same set of statistics in all databases. Finally, we executed each query five times –with cold buffers– and kept the median execution time. We used a timeout T_{max} of 30 minutes as in [9], but no execution exceeded T_{max}.

Database	Size	# Indexes
NREF-P	8GB	6
NREF-P1C	34GB	35
NREF-R (tuned with NREF3J)	28GB	31
TPC-H-P	12GB	8
TPC-H-P1C	34GB	61
TPC-H-R (tuned with UnTH3J)	21GB	29
TPC-H-R (tuned with QGen[1])	34GB	15

Table 1: Databases used in the evaluation.

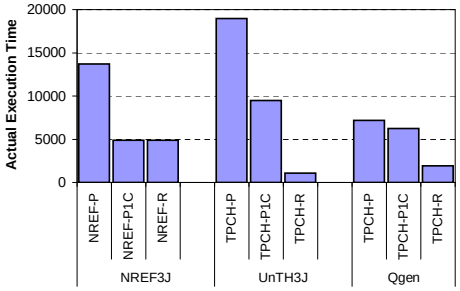


Figure 2: Overall Execution Times.

Figure 2 shows the overall execution times for all workloads and databases. Figure 3 shows $\mathcal{M}(t)$ for each database/workload combination. Finally, Figure 4 shows a variation of the $\mathcal{M}$ metric where we used the optimizer's estimated cost rather than the actual execution cost for the queries in the workload. We analyze these results next.

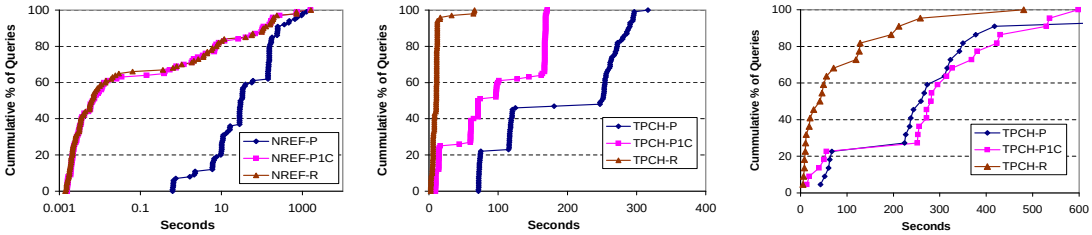
4 Analyzing *TAB*

We now analyze the results of the experimental evaluation of the previous section. In doing so, we also address some issues on TAB that we found during the evaluation and comment on some alternatives to diminish their impact.

Overall Comments. Figure 2 shows that the recommended configurations resulted in substantial improvement over the basic *-P* configurations. Specifically, the improvements were 64% for *NREF/NREF3J*, 94% for *TPC-H/UnTH3J*, and 73% for *TPC-H/QGen*. A noticeable difference with [9] is the performance of the *-P1C* configurations. While for *NREF/NREF3J* both *-P1C* and *-R* resulted in roughly the same performance, for *TPC-H/UnTH3J* the performance of *-P1C* lies almost exactly between that of *-P* and *-R*. Also, for *TPC-H/QGen* the performance of *-P1C*

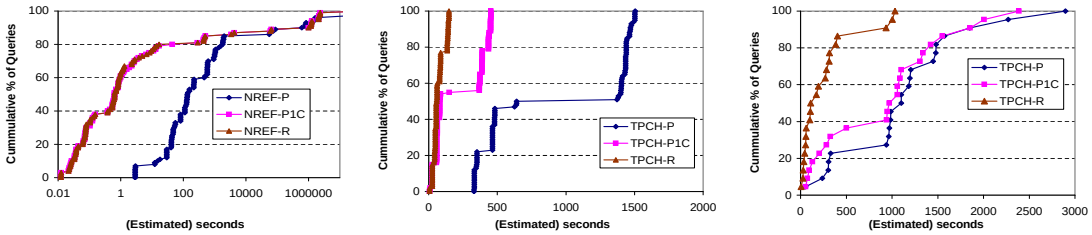
[1] All queries in the workload follow the same pattern: (i) self-join of a table T_1, (ii) join with a table T_2 that has a selective predicate, (iii) aggregates on the values of T_1 tables.

[2] Restrictions in the DBMS prevent us from creating certain indexes, such as indexes with keys larger than 900 bytes.



(a) NREF/NREF3J. (b) TPC-H/UnTH3J. (c) TPC-H/QGen.

Figure 3: Actual execution times for varying databases and workloads.



(a) NREF/NREF3J. (b) TPC-H/UnTH3J. (c) TPC-H/QGen.

Figure 4: Optimizer estimated execution times for varying databases and workloads.

is only slightly better than that of *-P* (a 13% improvement compared with 73% of *-R*). Figures 3 and 4 give additional information about the relative performance of different configurations. Almost 80% of the queries in *NREF-3J* finished in less than 10 seconds under either *-P1C* or *-R*, but only 10% of the queries under *-P* finished in that amount of time. For *TPC-H/UnTH3J*, all 100% of the queries finished in 75 seconds or less under *R*, where the percentages were 50% for *-P1C* and only 5% for *-P*. Finally, for *TPC-H/QGen*, 90% of the queries ran in less than 220 seconds under *-R*, where only 22% of the queries did the same under either *-P* or *-P1C*. Interestingly, the $\mathcal{M}$ curves for *-P* and *-P1C* cross each other for *TPC-H/QGen* in Figure 3(c), and therefore it is not clear how to interpret their relative performance beyond our original claim that the configurations were comparable. We examine and comment on the design of the TAB benchmark itself next.

4.1 Evaluation Metrics

The metric used to compare tuners is a crucial component of a benchmark. Usually, the existing literature uses a single number to measure the quality of recommendations, called *percentage improvement*, and defined as $1 - \textit{actual cost}/\textit{recommended cost}$. TAB recognizes that a single number might not provide enough detail to thoroughly evaluate and compare physical design tuners, and proposes the $\mathcal{M}$ metric to address this limitation. While we agree with the deficiency pointed out in [9] regarding single-value metrics, we identify some problems in $\mathcal{M}$.

4.1.1 Actual vs. Estimated Cost

The $\mathcal{M}$ metric is based on the actual time it takes to execute queries in the workload. We believe that in the context of evaluating a full system (i.e., not only the tuning tool, but also the query optimizer, query processor, and even the underlying operating system) this is clearly the best, most unbiased choice. However, if the purpose is an isolated evaluation of physical design tools, we claim that execution costs are, although important, less relevant. The reason is that using execution costs potentially introduces additional variables that are outside the scope of the evaluated tool. We next clarify this claim with real examples.

The Role of the Optimizer. It is important to note the we are bound to execute what the optimizer decides it is the best plan for a given query[3]. Consider the following example, simplified from a real query in *NREF/NREF3J*:

```
SELECT R.* FROM R, S
WHERE predicate(R) AND R.x=S.y
```

and suppose that the optimizer estimates that only a handful of tuples from R satisfy `predicate(R)`. If an index on $S.y$ is available, the optimizer would find that a nested-index-loop alternative that first gets all valid tuples from R and then fetches the matches from S might be a better alternative than, say, a hash join. Now suppose that the estimate is not right due to limitations in the optimizer's cost model, and in reality almost all tuples in R satisfy `predicate(R)`. In this case, the index-nested-loop plan, although it is costed the lowest by the optimizer and therefore chosen if possible, would execute much slower than the sub-optimal (to the eyes of the optimizer) hash-join alternative. Now the problem is clear. Consider the query above under the *-P* and *-P1C* configurations. The optimizer would pick the hash-join based alternative under *-P* (because there is no index on $S.y$ in *-P*) and the

[3]Hints can be used to override optimizer's decisions, but should be used with caution and as a last resource

index-nested-loop alternative under -*P1C* (because the index is present). The net effect is that the execution cost under -*P1C* would be significantly larger than that under -*P*, and we would tend to rank the tuner that produced -*P* higher than the one that produced -*P1C*. However, note that under -*P1C* the optimizer *considered* the hash-join alternative but discarded it in favor of the index-nested-loop plan! In fact, within the optimizer's cost model, the index-nested-loop alternative is better than the hash-based alternative in both -*P1C* and -*P* (although the former plan is not implementable under -*P*).

When purely evaluating the *quality of a physical design tuner*, we should be careful to freeze any external variables. It is therefore reasonable to assume that the optimizer is correct and the physical design tool exploits accurate information. Using the optimizer's expected cost rather that the actual execution cost of queries has precisely that effect, provided that the optimizer is operating under the same statistical model for all configurations (which we can achieve by materializing the same set of statistics, including those that are associated with indexes, in each database instance).

Runtime conditions. Another problem when using actual execution times is the unwanted presence of external factors that can compromise the accuracy of the measurements. In one of our earlier experiments, we noticed that the execution cost of a plan under -*P* was twice as fast as the corresponding plan under -*R* (which was odd since -*R* contained a strict superset of the indexes in -*P* and the query did not do any updates). Even more puzzling, a closer inspection of both plans revealed that they were indeed identical. After a long debugging session, we realized that the root cause of the problem was index fragmentation. In fact, the query required a sequential scan over an index. Since the index under -*P* was not fragmented, the execution engine could go through the index using sequential I/O, which is fast. In contrast, under -*R* the execution engine had to do one random I/O every 5 disk blocks on average due to fragmentation in the index, which resulted in a larger execution time overall.

It seems unfair to punish a tuner tool due to external factors that are not under its control. Although we minimized this effect by defragmenting the indexes and underlying disk in our experiments, there is always a chance that external factors play a role in biasing the results.

4.1.2 Timeouts in the $\mathcal{M}$ Metric

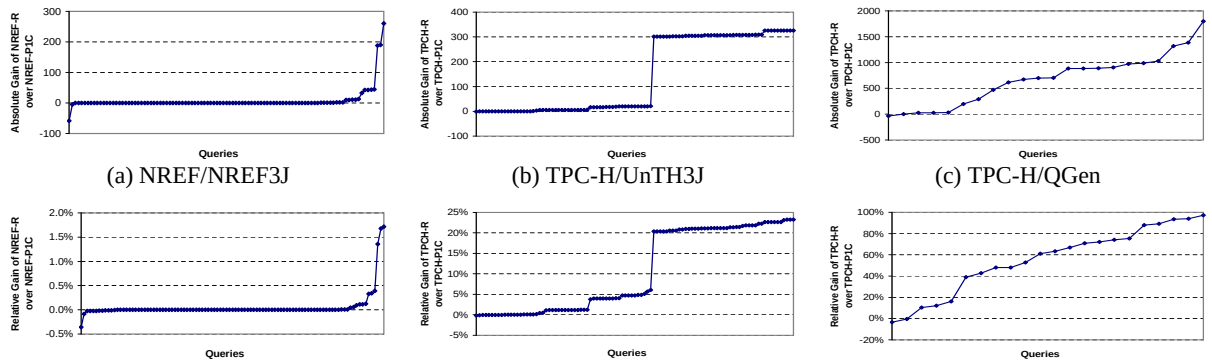
Reference [9] introduces a timeout value T_{max} that caps the maximum execution time of a query, set as 30 minutes. Although this is a practical issue to avoid very long running queries, it introduces some problems in the benchmark methodology. Specifically, it changes *a-posteriori* the optimization function that has been agreed upon and leveraged in tuning tools. Consider the following extreme scenario, with a 2-query workload that contains a light query q_1, which executes in 5 seconds under -*P* and a heavy query q_2 that executes in 3,600 seconds under -*P*. Consider a tuner T_1 that optimizes q_2 as much as possible at the expense of not fully optimizing q_1, and suppose that the resulting times are $(q_1{=}4, q_2{=}1900)$, with an overall execution time of 1,905 seconds, or a 47% improvement. A second tuning tool T_2, knowing *in advance* the 1,800-second timeout value, might optimize q_1 without considering q_2 obtaining the following times $(q_1{=}1, q_2{=}3600)$, with an overall execution time of 3,601 seconds, or just 0.1% improvement. Considering timeouts, the results are $(q_1{=}4, q_2{=}T_{max})$ for T_1 vs. $(q_1{=}1, q_2{=}T_{max})$ for T_2, harshly underestimating T_1's quality.

We believe that timeouts open the door for the possibility of "cheating" the benchmark by tools that exploit the subtle issues described above, and therefore recommend against using timeouts when evaluating configurations. (Strictly speaking, $\mathcal{M}$ itself uses a different optimization criterium to what has been adopted in tuning tools, but its limitations are less severe than those derived from timeout values.)

4.1.3 Aggregating individual results

Once we obtain execution times for each query in the workload, we need to display this information in a meaningful manner. TAB therefore introduced the $\mathcal{M}$ metric to show detailed information about performance of physical tuners. This metric is interesting in the sense that (i) allows to compare multiple tuners simultaneously, and (ii) allows for certain goal-oriented evaluation (such as 30% of the queries should execute in sub-second time [9]). One drawback of the $\mathcal{M}$ metric is that it does not report per-query comparisons because the individual queries are sorted in different orders. It is not possible, just by looking at $\mathcal{M}$ to draw conclusions about the performance of specific queries. For instance, although some queries were better under -*P* than under -*P1C* for *NREF*, Figure 3(a) is not enough to show this fact.

We next propose a complementary metric, which we call $\mathcal{I}$, that focuses on query-by-query performance. Consider configurations C_1 and C_2 coming from two tuning tools. We then compute, for each query q_i in the workload, the value $v_i{=}cost(q_i, C_1) - cost(q_i, C_2)$. Clearly, positive v_i values correspond to queries that were better under C_1 than under C_2, and negative v_i values correspond to the opposite situation. We then sort v_i values and plot the results. Figures 5(a-c) show our proposed metric for the databases/workloads in our evaluation. Analogously, Figures 5(d-f) shows a variation of the $\mathcal{I}$ metric that normalizes each v_i value by $cost(q_i, \text{-}P)$ (i.e., the cost of the query under the configuration that only has primary indexes). We can quickly see, for instance, that for



(a) NREF/NREF3J (b) TPC-H/UnTH3J (c) TPC-H/QGen

(d) NREF/NREF3J (e) TPC-H/UnTH3J (f) TPC-H/QGen

Figure 5: Proposed $\mathcal{I}$ metric to compare physical design tuners.

NREF/NREF3J both *-P1C* and *-R* result in almost no difference in performance, but there are still some queries (which are easily identified in the figure) for which *-R* resulted in better performance. Also, for *TPC-H/UnTH3J* we can see that there are two clusters of queries: one that results in almost no variation between *-P1C* and *-R*, and another for which the variation is significant in *-R*'s favor. Finally, *TPC-H/QGen* goes from no variation to almost 100% relative change in performance.

Although the $\mathcal{I}$ metric gives additional information on a per-query basis, it cannot be used to compare more than two configurations. We believe that $\mathcal{M}$ and $\mathcal{I}$ are complementary metrics that provide different types of insights when comparing physical design tuners.

4.2 Baseline Configuration

Before beginning our experiments we were surprised by the consistently good performance of *-P1C* claimed in [9]. Our experiments led to two key observations. First, current tuning tools result in configurations that range from comparable to *-P1C* to significantly better than *-P1C*. Second, there is a very large variance of performance of *-P1C* configurations, ranging from close to the best known solutions to close to the trivial configurations. In light of these observations, and based on Figures 3 and 4, we argue against using *-P1C* as a baseline configuration to compare against recommendations.

At some level, it is intuitive that *-P1C* would not be particularly helpful in general, and specifically for decision support workloads that require aggregating or filtering multiple columns. However, *-P1C* is essentially indistinguishable from the best recommended configuration for the *NREF/NREF3J* instance, which features queries with joins and aggregation. We next explain the main reasons behind this rather unexpected result.

Implied Index Columns. Secondary indexes in a DBMS store at the leaf nodes enough information to locate tuples in the primary index. To avoid storing record-ids, which are volatile in the presence of updates, modern systems use the columns in the primary index as this identifier[4]. This implies that, for all practical purposes, single-column indexes implicitly behave as multi-column indexes. We cannot seek these implied columns, but execution plans can rely on them as if they were explicitly declared. Now consider the *NREF* database. Not only the tables in *NREF* are narrow (the median number of columns is only five), but also the primary indexes are wide. As an example, consider table `source`, which is composed of six columns, four of which are part of the primary index. In this case, every single-column index on `source` essentially contains 4 or 5 out of the 6 columns of the table! In fact, since just a minority of the table columns is not present in the index leaf nodes, single-column indexes in *-P1C* actually behave like "covering-indexes" for *NREF*.

Workload. Even for the "quasi"-covering-indexes in *-P1C* there are very simple examples that result in bad execution plans. Consider the following query in *NREF*:

```
SELECT taxon_id_2
FROM neighboring_seq
WHERE nref_id_2 < 'NF00000300'
```

where the predicate filters all but 7531 rows. The recommended configuration for this query has a covering index on `(nref_id_2, taxon_id)`, so it can seek the relevant tuples and return the results optimally with an expected time of 0.51 units and an actual execution time of 0.078 seconds. Note that the primary index for table `neighboring_seq` does not contain column `taxon_id`. Therefore, *-P1C* cannot use the index on `nref_id_2` to locate the valid tuples and then fetch the remaining columns because the cost would be too high. Instead, the best plan for *-P1C* is to scan the index on `taxon_id`, which implicitly contains column `nref_id_2` and filter on the fly the resulting tuples. The expected cost of this strategy is 3.22 units (632 times slower than *-R*), and the actual execution

[4]If the primary index is not unique, a special "uniquifier" column is implicitly added.

time is 67.6 seconds (8667 times slower than *-R*). Additionally, for workloads with many updates, the performance of *-P1C* would be heavily deteriorated due to the overhead of updating the relevant indexes. Clearly, *-P1C* can result in very bad execution plans for the simplest of queries. A closer analysis of *NREF3J* shows, however, that for virtually all queries such situations fortunately do not happen, and thus *-P1C* performs extremely well in this scenario.

4.3 Database/Workloads

Once the metrics have been defined, the most important component of a benchmark is the actual databases and workloads over which it would be run. The TAB benchmark goes in the right direction by proposing both real (*NREF*) and synthetic (*TPC-H*) databases and workloads. However, it is also an example of how careful we need to be when designing benchmarks: by only considering *NREF/NREF3J* and *TPC-H/UnTH3J*, reference [9] arrives at the questionable conclusion that *-P1C* is a very competitive configuration. Another subtle problem with the *NREF* workload is that there is over six orders of magnitude difference between the slowest and fastest queries. Having very long queries in the workload is that these "rogue" queries might bias the result, specially in conjunction with timeout values in the $\mathcal{M}$ metric.

We believe that database/workload generation for the purposes of physical design benchmarks is an open area of research. In the meantime, we believe that useful benchmarks should contain databases/workloads taken from at least the following three "buckets":

- Micro-benchmarks that evaluate the different capabilities of the underlying DBMS and for which optimal configurations can be manually derived.
- Synthetic, complex workloads that exercise the full capabilities of the underlying query processor and cannot be manually analyzed.
- Real databases and workloads to address subtle scenarios that might have been overlooked in the previous two buckets.

5 Conclusions

In this paper we reported an experimental evaluation of the TAB benchmark for automated physical design tuners. We described TAB and its design choices and analyzed the quality of recommendations of our prototypes for the databases and workloads specified in TAB. In doing so, we identified certain weaknesses in the design of TAB and proposed alternatives to mitigate their impact. While TAB is an important first step in the area of physical tuning tool benchmarking, we believe that more work is needed. In particular, one of the biggest challenges in the area is to obtain a principled way to generate databases and workloads that are comprehensive enough to compare competing tools that might be based on very different principles. We note that both [9] and this work assume that the underlying database system does not change across alternative physical design tuners. If this assumption does not hold, it is not even clear how the different tuners could/should be compared (actual execution times might be an ultimate metric, but they evaluate the whole system rather than just the tuning tool). We believe that this is a rather deep problem that might have profound implications in future research on physical design tuning.

References

[1] TPC Benchmark H. Available at `http://www.tpc.org`.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of VLDB*, 2000.

[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of VLDB*, 2004.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of SIGMOD*, 2004.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of SIGMOD*, 2005.

[6] N. Bruno and S. Chaudhuri. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of VLDB*, 2006.

[7] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. In *IEEE Trans. Knowl. Data Eng. 16(11)*, 2004.

[8] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of VLDB*, 1997.

[9] M. Consens, D. Barbosa, A. Teisanu, and L. Mignet. Goals and benchmarks for autonomic configuration recommenders. In *Proceedings of SIGMOD*, 2005.

[10] B. Dageville et al. Automatic SQL Tuning in Oracle 10g. In *Proceedings of VLDB*, 2004.

[11] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *Workshop on Self-Managing Database Systems*, 2007.

[12] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of ICDE*, 2000.

[13] C. Wu et al. The protein information resource: an integrated public resource of functional annotation of proteins. In *Nucleic Acids Research*, 2002.

[14] D. Zilio et al. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of VLDB*, 2004.

[15] D. Zilio et al. Recommending materialized views and indexes with IBM DB2 design advisor. In *International Conference on Autonomic Computing*, 2004.