Goals and Benchmarks for Autonomic Configuration Recommenders

Mariano P. Consens
University of Toronto
Toronto, ON
Canada
consens@cs.toronto.edu

Denilson Barbosa*
University of Calgary
Calgary, AB
Canada
denilson@cpsc.ucalgary.ca

Adrian Teisanu
University of Toronto
Toronto, ON
Canada
ateisanu@cs.toronto.edu

Laurent Mignet†
IBM Research
New Delhi
India
lamignet@in.ibm.com

ABSTRACT

We are witnessing an explosive increase in the complexity of the information systems we rely upon. Autonomic systems address this challenge by continuously configuring and tuning themselves. Recently, a number of autonomic features have been incorporated into commercial RDBMS; tools for recommending database configurations (i.e., indexes, materialized views, partitions) for a given workload are prominent examples of this promising trend.

In this paper, we introduce a flexible characterization of the performance goals of configuration recommenders and develop an experimental evaluation approach to benchmark the effectiveness of these autonomic tools. We focus on exploratory queries and present extensive experimental results using both real and synthetic data that demonstrate the validity of the approach introduced. Our results identify a specific index configuration based on single-column indexes as a very useful baseline for comparisons in the exploratory setting. Furthermore, the experimental results demonstrate the unfulfilled potential for achieving improvements of several orders of magnitude.

1. INTRODUCTION

The area of autonomic computing has received considerable attention in the recent years, particularly in industry, and aims at providing systems that can adjust themselves to a changing environment. The vision of autonomic computing is to eliminate the need for human intervention in tuning the systems, and is motivated by: (1) increasing system usability (by allowing non-expert users to achieve acceptable performance); (2) decreasing operational costs (by reducing the demands on system administrators); (3) deploying systems in scenarios where human intervention is impossible or undesirable (e.g., in pervasive and embedded computing environments).

*Work done while the author was a PhD student at the University of Toronto.

†Work done while the author was a Postdoctoral Fellow at the University of Toronto.

Recently, a number of autonomic features have been incorporated into commercial RDBMSs as well. In particular, tools that recommend indexes for a given workload (such as [6, 14]) are crucial first steps toward autonomic data management. Agrawal *et al.* [2] discuss the recommendation of materialized views as well as indexes, possibly defined over the recommended views, given a query workload. Most of the work on autonomic databases has centered around using the DBMS's own query optimizer for comparing hypothetical scenarios [7]. The input to the process is typically a query workload and a budget (often in the form of a bound on the disk space that can be used for additional indexes). Currently, most major DBMS vendors support tools that behave in this way [1, 8, 10, 22, 25].

The potential performance impact of an effective index configuration can dwarf any other system parameter that a database administrator or an autonomic tool could possibly tune. This observation is not new: two decades ago Boral and DeWitt[3] concluded that parallelism is no substitute for effective and efficient indexing. The experimental results in this paper demonstrate performance improvements of several orders of magnitude due to indexing only. Therefore, the practical value of configuration recommender tools can be very significant.

In this paper, we introduce a flexible characterization of the performance goals of configuration recommenders and develop an experimental evaluation approach to benchmark the effectiveness of these autonomic tools. The proposed benchmark[1] provides a valuable assessment of the capabilities of the current crop of configuration recommenders. The benchmark also provides a foundation to evaluate future autonomic proposals. Finally, the experiments demonstrate the need for additional research in the area: there is the potential for achieving improvements of several orders of magnitude compared to current tools.

1.1 Exploratory Queries on NREF

To motivate this work, we present a realistic scenario for autonomic data management tools in the context of supporting exploratory queries on the Non-redundant REFerence protein database (NREF, for short), published on the web by the Protein Information Resource [23]. NREF provides a comprehensive collection of protein sequence data from several genome sequencing projects (PIR-PSD, SwissProt,

[1]Files available at `http://www.cs.toronto.edu/~consens/tab/`.

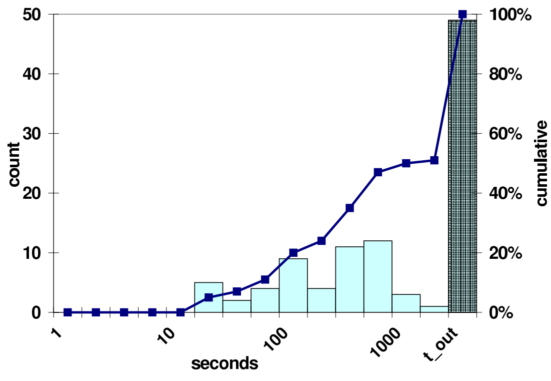

Figure 1: Query execution times on system A using a configuration with primary keys only.

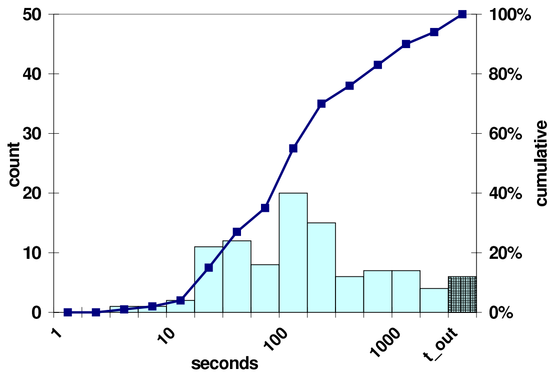

Figure 2: Query execution times on system A using a recommended configuration.

TrEMBL, RefSeq, GenPept, and PDB) and has identical sequences from the same source organism reported as a single NREF entry. The database is updated biweekly; release 1.34 contains 1,393,678 entries and its XML representation has 17GB. Once the XML data is converted to "raw" relational format (i.e., CSV text files) it occupies 6.5GB.

The relational schema for the NREF database is shown below (primary keys are underlined).

Protein(<u>nref_id</u>, p_name, last_updated, sequence, length)
Source(<u>nref_id</u>, <u>p_id</u>, <u>taxon_id</u>, <u>accession</u>, p_name, source)
Taxonomy(<u>nref_id</u>, <u>taxon_id</u>, <u>lineage</u>, species_name, common_name)
Organism(<u>nref_id</u>, <u>ordinal</u>, taxon_id, name)
Neighboring_seq(<u>nref_id_1</u>, ordinal, <u>nref_id_2</u>, taxon_id_2, length_2, score, overlap_length, start_1, start_2, end_1, end_2)
Identical_seq(<u>nref_id_1</u>, ordinal, <u>nref_id_2</u>, taxon_id)

The Protein relation (1.1 million rows) contains a unique identifier for each of the amino acid sequences in the database. The Source relation (3 million rows) contains the name of the database (e.g., SwissProt) where a given sequence is reported, and the corresponding access key for the protein on that database. All known taxonomic information about a given amino acid sequence is stored in the Taxonomy and Organism relations (15.1 and 1.2 million rows, respectively). Finally, the Neighboring_seq relation (78.7 million rows) associates pairs of closely related sequences within the same organism, while the Identical_seq relation (0.5 million rows) contains pairs of identical sequences that occur in different organisms. Neighbor NREF sequences are identified based on scores obtained by performing all-against-all FASTA searches.

Consider now that the biologist in our scenario is interested in executing hundreds of exploratory queries such as the one below:

EXAMPLE 1. *This SQL query finds the number of protein sequences (*`nref_id`*s) for each taxon associated with a virus that infects apes, and has been recently linked with cancer in humans (see* `http://www.cancer.gov/newscenter/sv40`*).*

```
SELECT t.lineage, count(distinct t2.nref_id)
FROM source s, taxonomy t, taxonomy t2
WHERE t.nref_id = s.nref_id
  AND t.lineage = t2.lineage
  AND s.p_name = 'Simian Virus 40'
GROUP BY t.lineage
```

Further assume that, during her exploration of NREF, the biologist has to execute 100 queries sampled from a much larger family of relevant queries, which we will call NREF2J in the sequel. Each of those queries executes in a certain amount of time. We visualize the response times experienced by the biologist while using a given configuration of NREF on a given DBMS by plotting the histogram of the query execution times.

For instance, Figure 1 shows the response times of the 100 queries executed on a commercial RDBMS (which we call System A). The RDBMS has a configuration of the NREF database where the only indexes present are those automatically created for the primary keys of each relation. Note that we define the bins using a logarithmic scale, resulting in execution times that are *three orders of magnitude slower* on the right of the graph than those on the left. Also, we report all "timeout" queries on a single bin (labeled `t_out` in the figure) using a reasonable timeout limit of 30 minutes for each individual query.

Contrast the histogram in Figure 1 with the one in Figure 2 showing response times observed on System A using a different configuration with several additional recommended indexes. Not only there is a significantly smaller number of timeout queries in the recommended configuration, but also the proportion of queries that complete in about 5 minutes is much larger.

The lines superimposed in both of the preceding figures are the cumulative frequencies of the query response times. Throughout this paper we argue that these curves provide a concise and valuable way of comparing the behavior of a workload on different configurations of the same system. For instance, we read in Figure 2 that 55% of the queries in the workload execute in less than 100 seconds on the recommended configuration. A similar reading in Figure 1 shows that only 20% finish in 100 seconds or less in the original configuration.

The bend on the cumulative frequency curve provides visual contrast between a satisfied (bend towards the top left corner) versus a frustrated (bend towards the bottom right corner) biologist-turned-database-user.

1.2 Our Contributions.

The literature on configuration recommenders provides very limited experimental evaluation of the *quality* of the recommendations produced by the tools. A few results are reported that characterize how *efficiently* these tools arrive at some recommendation, such as results showing how quickly they can produce a heuristic solution to the combinatorial problem of index selection. However, to the best of our knowledge, there has been no evaluation of the effectiveness of configuration recommenders.

In this paper, we describe an approach to evaluate the quality of the configurations suggested by configuration recommenders. We focus on the two aspects of the system configuration that can have the biggest impact: indexes and (possibly indexed) materialized views. We present extensive experimental results characterizing the effectiveness of commercial RDBMS configuration recommenders when presented with a workload consisting of exploratory queries. Our contributions are as follows:

- We provide a novel framework for assessing the effectiveness of configuration recommenders. In particular, our framework supports describing very large workloads and also provides a flexible characterization of performance goals using cumulative frequency curves.
- We present extensive experimental results on state-of-the-art commercial RDBMS configuration recommenders and show that there is substantial room for improvement.
- We identify a configuration that covers all single column indexes as a very useful baseline for comparing against recommendations. In fact, the consistently good performance of the single column configuration suggests a practical improvement for RDBMS configuration recommenders: avoid missing the potential gains brought by single column indexes.

The use of cumulative frequency curves to describe the response time behavior of database systems is common among practitioners and in the experimental systems literature. We do not claim originality in the measure itself; on the contrary, we endorse it because of its widespread acceptance. However, we are not aware of the use of cumulative frequency curves either as a goal for a recommender or as a measure of its effectiveness, despite the natural applicability of the concept. In fact, existing recommenders use a single numeric measure to describe workload behavior (something that we consider insufficiently rich to capture enough relevant aspects of the workload behavior).

The single column indexing approach that we discuss here can be viewed as an extreme case of schema design by vertical partitioning. We note that there has been work on autonomic schema design tools that use vertical partitioning [16], but without considering the recommendation of indexes.

The outline of the paper is as follows. We introduce a framework for evaluating autonomic indexing tools in Section 2, followed by a description of the challenges and the approach we choose in designing a benchmark in Section 3. We present our initial experimental results in Section 4, followed by a more detailed analysis of the recommenders performance in Section 5. Finally, we conclude in Section 6.

2. EVALUATION FRAMEWORK

In this section we describe the framework used to evaluate the performance of a configuration recommender. We start by describing the task that a recommender performs as well as the factors in the RDBMS environment that affect the recommendations. We then present some basic definitions and notation for characterizing costs and performance goals.

2.1 The Recommendation Task

In broad terms, the basic task of a configuration recommender is to select a new configuration for the RDBMS system that improves the performance that the system exhibits when executing a workload. Alternatively, the recommender can be given a performance target and it should find a configuration where the target is reached. The selected recommendation can be applied by the recommender itself, or the user may be given the option to accept or reject the recommended change in configuration. To produce a recommendation, the recommender has to: (1) assess the cost of executing a workload in a given configuration; (2) assess the cost of changing the system configuration; (3) search among possible system configurations to find a better performing configuration, given some constraints (such as a budget for changing configurations).

The most relevant aspect of the system configuration for an index recommender is, not surprisingly, the set of indexes available. However, a number of additional aspects can be considered part of the configurations being recommended such as data placement or the selection of materialized views [2, 24]).

There needs to be some definition of the performance goal that a recommender is trying to reach or improve upon. This goal can be a simple number (the execution time of a workload) or a more comprehensive (perhaps multidimensional) measure of overall system performance. The workload can also be defined in a variety of ways. The recommender may assume a known workload: the queries (and updates) together with their frequencies are given in advance. There may be a component in charge of automatically providing such a workload to the recommender based on observing the RDBMS operation [4]. Alternatively, there may be a describable set of potential queries that are candidates for workloads.

The RDBMS environment that influences the recommender task includes the instance of the data (or a summary description of the database instance, such as selected statistics), the parameters selected for the RDBMS engine as well as the engine itself (the supported query plans and the operators implemented), and all aspects of the physical data storage (including not just indexes, but also the layout of the data in the storage medium, replicas, materialized views, and so on).

2.2 Costs and Performance Goals

Let us denote by C_i the configuration (i.e., set of indexes, materialized views, etc.) of a system in state i. There is a set of possible configurations $C_{j_1}, C_{j_2}, \ldots, C_{j_n}$ that the recommender can possibly select for the next system state j, and one actual selected recommendation $C_j = C_{j_m}$ for some m.

Consider $q_k \in \mathcal{F}$, where $\mathcal{F}$ is the family of queries (or updates) that the system may execute. We denote by $A(q_k, C_i)$ the actual cost (a measure) of the system executing query q_k in configuration C_i. Similarly $E(q_k, C_i)$ denotes the *estimated* cost of executing query q_k in configuration C_i. Finally, $AT(C_i, C_j)$ is the cost of changing the system from configuration C_i to configuration C_j, while $ET(C_i, C_j)$ is the corresponding estimated transition cost from configura-

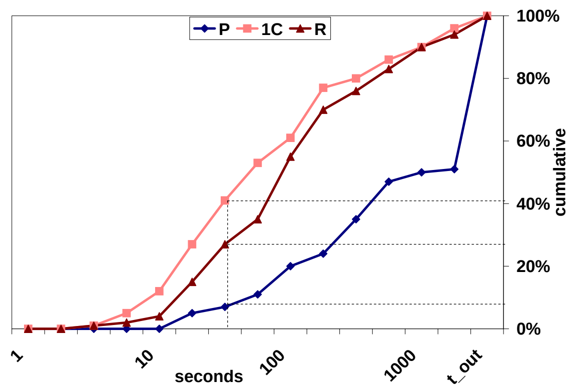

Figure 3: Behavior of system A on NREF2J.

tion C_i to C_j.

A workload is defined as a subset $\mathcal{W} \subseteq \mathcal{F}$ of the potential family of queries that the system may execute. Alternatively, it can also be defined as a bag, in which case the repetitions can model queries with a higher frequency or weight.

Given a workload $\mathcal{W}$, we can measure the *actual performance* of the system on a configuration C_i by a single quantity $A(\mathcal{W}, C_i) = \sum_{q_k \in \mathcal{W}} A(q_k, C_i)$ (total cost). Similarly, the *estimated performance* is defined as $E(\mathcal{W}, C_i) = \sum_{q_k \in \mathcal{W}} E(q_k, C_i)$ (total estimated cost).

Finally, we denote by CF_{C_j} the cumulative (relative) frequency of the elapsed times $A(q_k, C_j)$ for $q_k \in \mathcal{W}$ on configuration C_j, defined as:

$$CF_{C_j}(x) = \text{count}(\{q_k : A(q_k, C_j) < x\})/\text{size}(\mathcal{W})$$

Figures 1 and 2 show the cumulative frequency of the elapsed times of a 100-query workload for two database configurations, as discussed in Section 1.1. Contrasting cumulative frequencies of elapsed times on a given workload is an informative approach to compare different configurations; for instance, Figure 3 compares three configurations, called P, 1C and R (which will be explained later) on system A. The cumulative frequency polygons convey that configuration 1C is superior to both R and P. In particular, the dotted lines in the figure show that 41% of the queries in the workload executed on configuration 1C complete in less than 31.6 seconds (the value $10^{1.5}$ in the x axis), instead of 27% on the R configuration and only 7% on the P configuration.

While the use of cumulative frequency polygons has limitations[2] they have advantages over histograms (requires no binning and quantiles can be read directly) and they are widely employed in decision making requiring comparing distributions (our use corresponds to deciding first order stochastic dominance).

A Model for Configuration Recommenders. We can describe the behavior of the configuration recommenders incorporated in commercial RDBMS [8, 22] using the framework discussed above. The RDBMS configuration recommender takes as input a given workload $\mathcal{W}$, including the relative frequencies of the queries in the workload. The recommender's goal is to select a configuration C_j that improves the total estimated cost of queries $E(\mathcal{W}, C_j)$, where the total cost may use different weights for the queries in the workload. This optimization goal is subject to an estimated storage budget (hence $ET(C_i, C_j)$ uses storage as the measure). The configuration recommender uses the RDBMS optimizer's estimation capabilities to asses $E(\mathcal{W}, C_j)$. The optimizer has to hypothesize statistics for C_j from the statistics in the current configuration C_i. Since there is a combinatorial space of possible index configurations, the RDBMS configuration recommender relies on a heuristic search to compute estimates for a subset of the configurations.

[2]Humans are poor at judging the distance between curves, our visual processing assesses the closest differences between curves rather than the correct vertical distances [9].

Performance Goals. A performance goal for the configuration recommender can be stated as a simple target measure for the sum of the individual query execution measures over the queries in the workload. More specifically, if the measure is elapsed time, then we would have total elapsed time of executing the workload as the performance goal (for example, *complete the workload in less than two hours*). Finer-grained goals than total execution cost are usually more informative: "naive folks will use the average response time; more sophisticated specifiers will opt for the 90th or 95th percentile" [20].

A performance goal can also be stated as an improvement ratio $IR = A(\mathcal{W}, C_i)/A(\mathcal{W}, C_j)$ where C_i, C_j are the existing and selected configurations, respectively. Continuing with the elapsed time example, a goal could be to obtain a 10 times improvement (by decreasing elapsed times in the recommended configuration by an order of magnitude).

We note that performance goals can be a more elaborate than a single quantity. In fact, a performance goal can be viewed as a quality of service requirement that specifies minimum levels of performance that must be met by the system. Again, making use of elapsed time as an example measure, consider the performance goal below.

EXAMPLE 2. *A performance goal for the execution of a set of queries can be to expect 10% of the queries to complete in less than 10 seconds, 50% to complete in less than one minute, and 90% to complete before a 30 minute timeout. This goal can be described by a step function:*

$$\begin{aligned} G(x) &= 0, & x &< 10 \\ G(x) &= 0.1, & 10 \le x &< 60 \\ G(x) &= 0.5, & 60 \le x &< 1800 \\ G(x) &= 0.90, & x &\ge 1800 \end{aligned}$$

where seconds are used as units and we use values in the $(0, 1)$ *interval instead of percentages.*

A performance goal such as G above can be viewed as a constraint in the shape of the cumulative (relative) frequency (CF_{C_j}) of the elapsed times on configuration C_j. A configuration C_j satisfies the performance goal if $CF_{C_j} > G$. For instance, in Figure 3, configuration 1C satisfies the goal G above, while the other two do not. Note that any monotonic function G can be used as a performance goal in this setting.

3. BENCHMARK DESIGN

We start this section by discussing the challenges of evaluating the performance of configuration recommenders, and describe how we address those challenges in the benchmarks that we propose.

3.1 Design Criteria

Generic database benchmarks aim at providing an objective means of *comparing* competing systems executing the same task [12]. Usually, they define a database and a workload, a set of rules specifying the environment under which all tests must be run (e.g., which kinds of indexes are allowed), and rules for performing the measurements and reporting the results. The results of the tests are typically summarized into single quantities such as the system throughput or its price/performance ratio. The benchmark is then run on several systems, and the results are compared, determining the system with best performance or cost/performance trade-off, etc.

One of the fundamental challenges in designing a database benchmark that is accepted by the community is defining a workload that is representative of specific scenarios in real applications. Another important goal in benchmark design is to succeed in evaluating the specific aspects of the database system that are of interest in (relative) isolation. Later in the paper we spend considerable attention designing workloads. We are also careful to isolate the configuration recommenders behavior by reducing the number of factors that have an effect on the experimental evaluation and also by further limiting ourselves to just two aspects of the system configuration: indexes and (potentially indexed) materialized views.

The goal of our work is *not to compare* configuration recommenders *across competing database systems*. Instead, we focus on *comparing the quality* of the recommendations produced by one or more configuration recommenders *running on the same database system*. System configurations and the results achieved by a recommender suggesting one configuration over another are inherently system-specific. This makes a recommendation not directly comparable to another recommendation in a different system. The valid question of comparing a database system equipped with a configuration recommender to a competing database system should be addressed separately (and it is not an objective of this work).

We focus now on criteria tailored to a benchmark for configuration recommenders. There are a number of challenges in designing experiments that can adequately evaluate the performance of these recommenders. First, we have to select a suitable database and define an initial configuration (or, more generally, a method for generating such instances).

Second, we need to provide a number of sufficiently large and varied workloads $\mathcal{W}_1, \ldots, \mathcal{W}_k$ that are representative of an application domain. Both the heterogeneity of the queries in the workload and the number of different queries included has an impact on the level of difficulty of the recommender task. It is not unusual for large database systems to routinely deal with workloads of thousands or even tens of thousands of different queries. As we will present later on, workloads with one hundred queries are challenging enough for the current state of the art.

Third, we must define the input parameters for obtaining the recommended configurations $C_{j_1}, C_{j_2}, \ldots, C_{j_k}$ (such as the resources available). Finally, we must provide a way of evaluating the *quality* of the recommendations. In this work, we propose comparing the cumulative frequency of the elapsed times for each workload evaluated in the recommended configuration against the elapsed times on one or more *reference configurations* $C_{h_1}, C_{h_2}, \ldots, C_{h_l}$.

One of the few recommender evaluations in the literature is described by Valentin *et. al* [22]. The database and initial configuration are those defined in the TPC-D benchmark; the 17 queries in the benchmark are used as the single workload $\mathcal{W}$, and an expert-tuned configuration C_h is used as reference configuration. The results presented in that work show that the recommender suggested a configuration C_j that performed as well as C_h in 14 out of the 17 queries. This is a very encouraging result: the comparison configuration used is expected to perform extremely well and hence matching its performance is quite an accomplishment. However, the level of difficulty of the task is limited by the small number of queries in the workload.

3.2 Our Approach

We address the criteria described above as follows. First, we use both real and synthetic databases, which are adequately scaled to the computing resources available in the desktop computing environment we utilize. The initial configurations are instances in which all primary key and foreign key constraints in the relational schema are defined, and where only primary key indexes are created; we refer to such initial configurations as P configurations. Second, we define the workloads as *query families*, which are sets of queries that contain a large number of structurally related yet suitably diverse queries. While we focus on retrieval queries and do not consider updates in our workloads, we present experimental results quantifying the effect of inserts on the recommended and the reference configurations. Third, the only restriction we impose on the recommended configurations is that they do not use more space for indexes than the reference configurations. Finally, we identify a *single* reference configuration that has a single column index for each possible (indexable) column in the schema (and we refer to it as 1C, for 1-column index).

3.2.1 The Databases

Three databases are used in our experiments: the NREF database discussed in Section 1.1, the TPC-H [21] database, and also a skewed version of the TPC-H [5], generated with a Zipfian factor of 1. The sizes of these databases in raw format are 6.5GB for NREF, and 10GB for each TPC-H database. While all the sizes are relatively modest they cannot be dismissed as ridiculously small. Furthermore, the raw data size is an order of magnitude larger than the main memory of the computers utilized.

As mentioned above, the initial configurations are instances in which all primary and foreign key constraints are defined. For the NREF database, such constraints are as defined in the schema in Section 1.1, while for the TPC-H databases we follow the schemas specified in that benchmark [21].

3.2.2 The Query Families

Due to the exploratory nature of our motivating scenario, we focus on queries that represent fragments of typical "iceberg" queries [11]; that is, queries that compute aggregate functions over a set of attributes to find aggregate values satisfying certain conditions, grouped in different ways.

We follow an approach similar to the one described in [18] (used for generating part of the query workload in the TPC-

DS benchmark [17]), in which query workloads are given as large *families* of queries described by templates.

In each template, variables are used instead of relation names, column names, and constants; the actual queries are obtained by binding such variables to relations and columns in the schema, as well as constants selected from the database.

The following criteria were used for designing the families we use in this work. First, the queries should have a meaningful interpretation. This is achieved by grouping columns in the schema by domains, and allowing joins on attributes in the same domain only. For example, referring to the NREF schema in Section 1.1, all attributes used for the scientific or common names of proteins, species and organisms are in the same broad domain and could be joined meaningfully. Second, the queries should be simple enough for query optimizers to have a good chance of handling them well. Thus, we use only simple select-project-join SQL queries defining simple aggregate functions and with at most one level of nesting, and defining only equality predicates. Third, the queries should not require the materialization of large intermediate results, as this could make irrelevant the presence of indexes in the database. To achieve this, we use additional selective predicates for each query. Fourth, the queries in the family should cover a reasonable spectrum of query execution times, from fast (e.g., sub-second response) to slow (e.g., a timeout after a reasonable long execution time). Finally, and perhaps most importantly, the queries should be amenable to improvement by the addition of both simple and complex indexes, defined either over base tables or materialized views. We achieve this goal in our work by defining join predicates only on indexable columns, as well as "wide" group by clauses in our queries.

Next, we give a brief description of each of the query families used in our experiments, followed by its SQL *template* and an explanation of how we obtain the constants used for generating the actual queries. Although we have experimented with several other families with a wide range of characteristics, the results presented for two families on the NREF database and three families on two different TPC-H database provide a useful assessment of the performance of current recommenders.

Family NREF3J. The first family, on the NREF database, is a generalization of the self-join pattern in the query described in Example 1. We pick a table R, and a column c_1 to define a self-join on R; then pick a another table S, and column c_2 (in the same domain as $c1$) and join $R.c_2$ with $S.c_3$. Next, we choose up to three other columns $c_{i_1}, \ldots, c_{i_3}$ in R and define a group by that includes c_1 as well. Finally, we add a selection condition of the form $S.c_4 = k$, where k is a constant selected as follows. For each column in each table, we pick three values k_1, k_2 and k_3 that can be used as the constant k such that k_1 has the highest selectivity for the column and the frequencies of k_2 and k_3 are one and two orders of magnitude (resp.) greater than the frequency of k_1. This is a template for the family:

```
SELECT r1.$c_{i_1}$,...,r1.$c_{i_3}$,r1.$c_1$,COUNT(DISTINCT r2.$c_2$)
FROM $R$ r1, $R$ r2, $S$ s
WHERE r1.$c_1$ = r2.$c_1$
AND r1.$c_2$ = s.$c_3$
AND s.$c_4$ = $k$
GROUP BY r1.$c_{i_1}$,...,r1.$c_{i_3}$,r1.$c_1$
```

Family NREF2J. Queries in the second NREF family count co-occurrences of values (from the same domain) in different tables. We pick tables R, S and a column from each table (c_1 and c_2) such that these columns are in the same domain; we then count the number of co-occurrences of values by joining $R.c_1$ and $S.c_2$. Next, we pick up to three other columns $c_{i_1}, \ldots, c_{i_3}$ in R to define a group by clause. Finally, we further restrict the values of both $R.c_1$ and $S.c_2$ to be relatively infrequent (i.e., occur less than 4 times) in order to limit the size of the intermediate join $R \bowtie S$. The template for this family is as follows:

```
SELECT r.$c_{i_1}$,...,r.$c_{i_3}$,r.$c_1$, COUNT(*)
FROM $R$ r, $S$ s
WHERE r.$c_1$ = s.$c_2$
AND r.$c_1$ in
  (SELECT $c1$ FROM $R$ GROUP BY $c1$
   HAVING COUNT(*) < 4)
AND s.$c_2$ in
  (SELECT $c_2$ FROM $S$ GROUP BY $c_2$
   HAVING COUNT(*) < 4)
GROUP BY r.$c_{i_1}$,...,r.$c_{i_3}$,r.$c_1$
```

Family SkTH3J. Queries in this family define three-way joins on the skewed TPC-H database (using a Zipfian factor of 1). Each query is obtained as follows. We pick tables R, S and T; define a join $R \bowtie S$ via primary key and foreign key correspondences; define a join $S \bowtie T$ via a pair of non-key columns $S.c_1, T.c_2$ from the same domain; and define a selection condition $\theta(S.c_3)$ on a column c_3 of S to limit the number of tuples in $R \bowtie S$.

In this family, $\theta(S.c_3)$ is one of $S.c = p$ or $S.c$ `IN (SELECT` c `FROM` S `GROUP BY` c `HAVING COUNT(*)=`p`)`, and the parameter p is used to control the sizes of the intermediate result $R \bowtie S$. Three $\theta(S.c_3)$ are used for each assignment of R, S and T, each defining a constant for the variable p. The criteria for selecting these three constants are that the resulting query is not empty and that the sizes of the intermediate result $R \bowtie S$ are in different orders of magnitude. That is, if k_1, k_2 and k_3 are the sizes of $R \bowtie S$ for each of the constants chosen, we have that k_2 and k_3 are one and two orders of magnitude (resp.) greater than k_1. Finally, each query returns a `COUNT(*)` where the group is defined by choosing up to 4 columns from relation T.

This is a template for the family:

```
SELECT t.$c_{i_1}$,...,t.$c_{i_4}$,COUNT(*)
FROM $R$ r, $S$ s, $T$ t
WHERE r.$c_{p_1}$ = s.$c_{f_1}$ AND ... AND r.$c_{p_j}$ = s.$c_{f_j}$
AND s.$c_1$ = t.$c_2$
AND $\theta$(s.$c_3$)
GROUP BY t.$c_{i_1}$,...,t.$c_{i_4}$
```

Family SkTH3Js. This family, also defined for the TPC-H database generated with skewed data, is a simpler version of family SkTH3J in which R,S and T are always one of Lineitem, Orders and Partsupp. An additional simplification is that all the $\theta(s.c)$ constraints are of the form $S.c = p$, where the constants are chosen as before.

Family UnTH3J. The last family uses the standard version of a TPC-H database (where all values are sampled with uniform distributions) and its queries are the same as those in the family SkTH3J above (except that different selection constants are used).

3.2.3 *Reference and Recommended Configurations*

We conclude the description of the benchmarks with a discussion of the relevant parameters used for obtaining the reference and the recommended configurations from the various systems. As mentioned earlier, the P configuration we use as initial configuration contains only those indexes automatically created for the primary keys of each table. The 1C reference configuration is created by adding to P all possible single column indexes (i.e., one index for each indexable column in the schema).

We obtain one recommended configuration for each query family in the benchmark. We direct the systems to collect statistics before obtaining the recommendations and before running the queries. All the recommended configurations are obtained using the P configuration as the starting point, the difference in size between 1C and P as the space budget, and no limit on the time the recommender is allowed to run.

The decision to define the space budget as above is motivated by the desire to make 1C as comparable as possible to the recommended configuration (the space used by 1C is the same space available for the recommendation). The space usage of 1C would be considered a high budget in many application scenarios and provides a generous amount of additional storage for the recommenders to work with. However, in our results (see Table 1, described in the next section), no recommended configuration in our experiments used as much space as 1C. We also obtained recommendations with an unlimited storage budget. The resulting unlimited space recommendations increased storage usage (in all cases by a reasonable amount) and did exhibit better performance than the space constrained recommendations in some (but not in all) cases.

We observe that the query families used in our tests resulted in fairly complex recommendations, involving single and multiple column indexes, as well as materialized views over joins of base tables (Tables 2 and 3, described in the next section).

4. EXPERIMENTAL RESULTS

In this section we describe the setup used for the experiments and we present our results.

4.1 Experimental Setup

We used two commercial RDBMSs running on four Pentium 4 desktop PCs ranging from a 2GHz machine with 752 MB of RAM running Windows 2000 Server; to a 2.6GHz machine with 1GB of RAM running Windows XP.

Two sets of experiments were run. The first experiment was run on the NREF benchmark and was aimed at understanding the behavior of the systems tested (which we call Systems A and B for this experiment) on a realistic scenario, using real data. The second experiment was run on both TPC-H benchmarks and was aimed at verifying our observations on a standard benchmark database, and to observe the impact of uniform versus skewed data on the behavior of the index recommenders. We selected one of the two systems for the second experiment, which we will refer to as System C.

The results we discuss next are based on actual executions of the query families in each benchmark. In all cases, the queries are run sequentially, and the machine is fully dedicated to running the experiments. For obvious practical reasons, a timeout limit of 30 minutes is set for running each query; queries that do not finish in that amount of time are reported as "timeout". We perform two additional runs on the queries that do not timeout on a first run, and report the average time of the three measures (the small variances observed do not justify removing outliers).

Benchmark	System	Size (GB)	Time (min)
NREF	A_NREF_P	13.5	322
	A_NREF2J_R	18.0	335
	A_NREF_1C	35.7	1171
	B_NREF_P	11.1	2161
	B_NREF2J_R	14.6	117
	B_NREF3J_R	15.1	281
	B_NREF_1C	17.1	1795
SkTH	C_SkTH_P	21.4	959
	C_SkTH3J_R	22.7	153
	C_SkTH3Js_R	21.8	576
	C_SkTH_1C	38.5	2860
UnTH	C_UnTH_P	21.4	923
	C_UnTH3J_R	23.2	901
	C_UnTH_1C	38.5	2197

Table 1: Sizes and build times of all configurations used in the experiments.

4.1.1 *Query Workloads*

The families presented in Section 3 contain large numbers of queries. For instance, NREF2J has 110,970 queries while NREF3J has 6,336 queries. We adopt a number of practical restrictions to further reduce the space of possible queries to consider. For instance, only subsets of each relational schemas are used in the queries: all non-indexable columns were ignored and we did not use more than 4 columns per table. Another restriction was to consider fewer selection criteria (thus, fewer queries) on the larger tables on each database; similarly, we used fewer columns in group by clauses on these tables. With all the additional restrictions, the resulting sizes of families NREF2J and NREF3J are 485 and 373 queries, respectively.

Despite the reduction in size, running just both NREF families on all configurations and systems remains a daunting task: a quick math shows it may require $(485 + 373) \times 3$ runs $\times\, 7$ configurations $\times\, 30$min $= 9,009$ hours or 375 days of machine use! The final reduction was motivated by the desire to work with the same (round) number of queries for all families: we sampled 100 queries from each family, in a way that the distribution of elapsed times of the larger family was preserved. While the query families for the TPC-H based benchmarks are substantially smaller (as fewer meaningful joins can be defined in that schema), we also work with samples of 100 queries for those families.

4.1.2 *Configurations Tested*

As a naming convention, the system name is used as a prefix for identifying configurations, and a "XXX_R" suffix, where XXX is the name of a query family, is used for identifying the recommended configurations; for instance, A_NREF_P refers to the P configuration on system A for the NREF database, while B_NREF2J_R is the configuration recommended by system B for query family NREF2J. Table 1 shows the building times and storage required for all configurations used in our tests, obtained as discussed in Section 3.2.3.

We note that we were not able to obtain recommenda-

Table	A_NREF2J_R				B_NREF2J_R				B_NREF3J_R			
	1c	2c	3c	4c	1c	2c	3c	4c	1c	2c	3c	4c
Identical_seq		1			2	1			1	6		
Neighboring_seq	3	1			1	1			1			
Organism										5		
Protein		1			1				1	1		
Source	1				1			1		1		1
Taxonomy	2			1	1		1			1	1	
Totals	6	3	0	1	6	2	1	1	3	14	1	1

Table 2: Number of 1,2,3, and 4-column indexes in each recommended configuration for the NREF benchmark. No index with more than 4 columns was recommended.

Table	C_SkTH3Js_R				C_SkTH3J_R				C_UnTH3J_R			
	1c	2c	3c	4c	1c	2c	3c	4c	1c	2c	3c	4c
Lineitem	2	2							1			
Orders	1				1	1			1	2		
Partsupp	2				1	2						
Supplier					1							
2 Views on Lineitem	N/A	N/A	N/A	N/A			1	1	N/A	N/A	N/A	N/A
9 Views on Lineitem $\bowtie$ Partsupp	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2	3	4	3
Totals	5	2	0	0	3	3	1	1	4	5	4	3

Table 3: Number of 1,2,3, and 4-column indexes in each recommended configuration for the TPC-H benchmarks. No index with more than 4 columns was recommended. Also, no indexes on Customer or Part were recommended.

tions for family NREF3J using system A (that is, the recommender did not output any recommended configuration at all). We tried with a few other samples of 100 queries from family NREF3J, as well as with smaller workloads consisting of 25, 12, 6, and 3 queries. While we verified that we could obtain recommendations for several of the smaller workloads, it did not make sense to pick any such configuration to represent the missing recommendation for the 100 query workload. Therefore, we do not report any recommendation for family NREF3J using system A's recommender.

Tables 2 and 3 show the number of indexes in each recommended configuration for the NREF and TPC-H experiments, respectively. Note that the recommendations for SkTH3J and UnTH3J contain indexes on both base tables and materialized views. For SkTH3J, 2 recommended indexes were defined on materialized views of Lineitem, while for UnTH3J, 12 of the 16 indexes recommended were defined on 9 materialized views over the join of Lineitem and Partsupp.

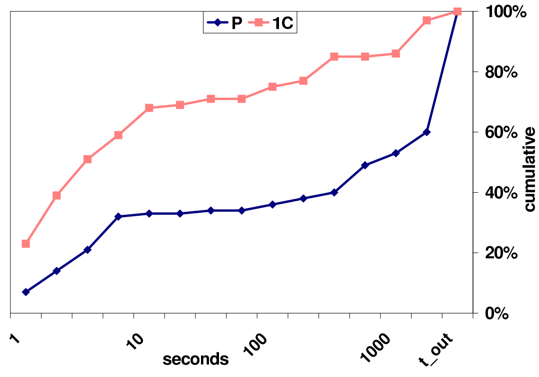

Figure 4: Behavior of system A on NREF3J.

4.2 Results on the NREF Benchmark

Recall the discussion in Section 2.2 about cumulative frequency distributions, and how to use them for comparing different configurations on the same system and workload.

Figure 3 in Section 2.2 describes the behavior of system A for family NREF2J. The curves readily show substantial improvements in the query executions times for the workload in both the A_NREF_1C and A_NREF2J_R configurations (which we will refer to as 1C and R within the context of the family and the system) relative to A_NREF_P configuration (similarly, P in context). The graphs also shows that the reference configuration 1C behaves better than the recommended configuration R, although the performance gap is very small for queries that require more than 1000 seconds to complete.

Figure 4 shows the result of System A on family NREF3J. The graph shows a more pronounced difference in performance for the P and 1C configurations than before: on 1C, 59% of the queries finish in less than 6 seconds (each), while on P, 60% of the queries may take as long as 1780 seconds to complete. Another way of looking at these numbers is: it takes 98 seconds to complete 60% of the queries on 1C, while it takes 4 hours and 45 minutes to complete 60% of the queries on P: an improvement of 174 times! There are only two curves in Figure 4 since, as discussed in the preceding subsection, the recommender in System A was unable to produce a recommended configuration R for this family. This is particularly surprising giving the vast benefit provided by the 1C configuration using only single column indexes.

Figures 5 and 6 show the behavior of system B on families NREF2J and NREF3J, respectively. As one can see, the performance of the recommended configuration for query family NREF2J is almost indistinguishable from that of the P configuration. In family NREF3J, the recommended configuration performs relatively better, but the gap it exhibits to the 1C configuration is still significant.

In summary, the 1C configuration was always far superior

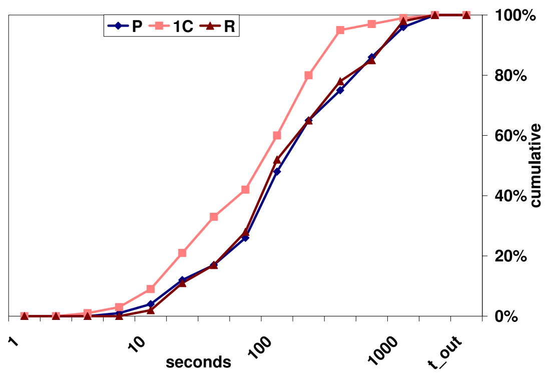

Figure 5: Behavior of system B on NREF2J.

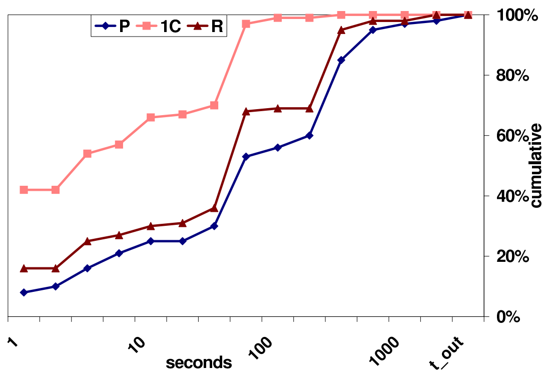

Figure 6: Behavior of system B on NREF3J.

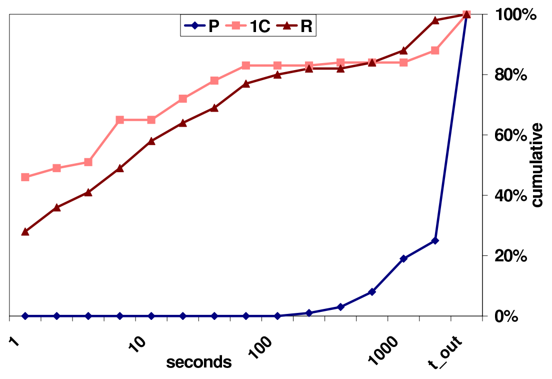

Figure 7: Behavior of System C on SkTH3Js.

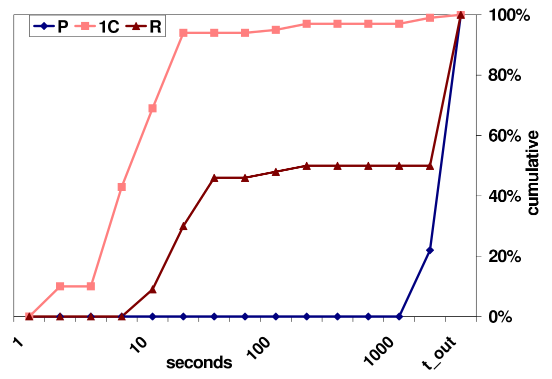

Figure 8: Behavior of System C on SkTH3J.

to the initial configuration P. A careful look at Figures 4 and 6 shows wide gaps between the P and 1C configurations, indicating large potential performance improvements by the use of indexes. In many cases, 1C was also far better than the configurations recommended by both systems. Therefore, we arrive at the surprising observation that both recommenders may fail to improve on the P configuration even when the potential for improvement is considerable.

4.3 Results on the TPC-H Benchmarks

The behavior observed on both synthetic datasets (shown in Figures 7, 8 and 9) is consistent with the behavior of the NREF benchmarks described in the preceding subsection. As mentioned earlier, we limit the results presented to one of the DBMSs that we call System C.

Notably, the 1C configuration continues to be very competitive with the R configurations (again, on occasions, 1C is far superior to R). This happens despite some of the recommendations using materialized views defined on joins of base tables, making the relatively better performance of 1C even more remarkable.

The only recommendation R in all our experiments to outperform 1C even on a small portion of the workload was obtained on family SkTH3Js (see Figure 7). The R configuration succeeds in speeding up the most expensive queries compared to 1C. Since the goal used by System C's recommender is total cost (instead of the quality of service curve represented in the graph), it is not surprising that the recommender favors improving long-running queries (the ones that dominate total cost).

A comparison of Figures 7 and 8 shows a sharp contrast between the behavior of System C for the simpler and the "generalized" 3-way join families in the TPC-H benchmark. This emphasizes the dependence of the configuration recommender on the input workload (for even relatively small variations in the structure of the workload).

Another interesting observation can be made by comparing the behavior of the recommender on skewed versus uniform data. Contrast the recommendations for SkTH3J and UnTH3J (Table 3) and the relative performance of these configurations in Figures 8 and 9. Clearly, the recommender did perform better for the uniformly distributed data. Nevertheless, the 1C configuration still proved the best overall.

Finally, given that the recommender considers the overall workload performance, and not the distribution of the individual query execution times, it is informative to present overall numbers for one workload. Consider the results of running SkTH3J on the configuration P. We observe that the total execution time for the queries that do not timeout is 34461 seconds, while there are 78 queries that timeout (taking at least 1800 seconds each). While we do not know how long timeout queries could take, we can use the timeout value to obtain a lower bound for the execution of workload SkTH3J on P of 174,861 seconds. A similar calculation gives lower bounds for the execution of workload SkTH3J on the configurations 1C and R of 5445 and 91019 seconds, respectively. Keep in mind, though, that 1C has only one timeout query while P and R have 78 and 50 timeout queries respec-

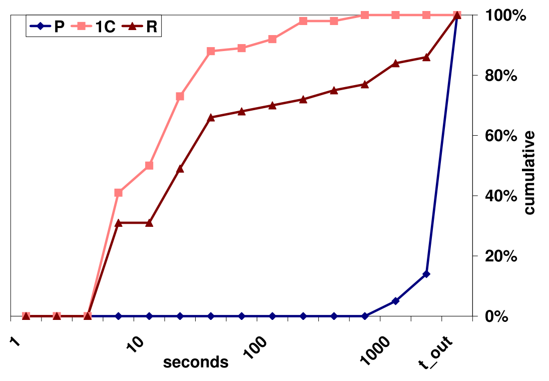

Figure 9: Behavior of System C on UnTH3J.

tively (hence the lower bound is much tighter on 1C than on R and P). Thus, a very conservative overall workload assessment results in 1C producing almost 17 times better results than R!

4.4 The Impact of Insertions

The workloads considered in this paper do not include updates. Designing update workloads (in particular, containing complex updates) is a valuable extension to the current benchmark. However, it is important to observe that using 1C as a reference configuration remains valid in the presence of a reasonable number of insertions.

To illustrate this, we run an experiment inserting tuples into the Neighboring_seq relation, both the widest and the largest relation in the NREF database, with almost 80 million rows. The insertions take roughly linear time in the number of tuples inserted in all configurations. As expected, it takes longer to insert tuples in the configuration 1C than in the recommended configuration. Insertions in the initial P configuration are faster than in the recommended configuration. Considering the workload NREF2J, we can determine the number of tuples that have to be inserted to make the elapsed time of executing the insertions plus the workload on 1C the same as the time of executing the insertions plus the workload on R. This is the break point at which the slower insertions, faster queries in 1C overtake the slower queries, faster inserts in NREF2J_R: in both systems A and B this number is close to 400,000 tuples. Keep in mind that this calculation is based on enforcing timeouts for query executions, so this is just a lower bound for execution times. Furthermore, the threshold of 400,000 tuple insertions is based on a single execution of each one of the 100 queries in the workload. If the queries are executed 20 times each, then the insertion threshold corresponds to increasing the database size by almost 10%.

5. RECOMMENDER LIMITATIONS

This section presents additional empirical evidence that highlights an important limitation in the existing crop of commercial recommenders.

5.1 Reliance on Estimation

As discussed in Section 2.2, current RDBMS configuration recommenders look at statistics from an existing system configuration C_i and are given query workload $\mathcal{W}$ together with a space budget B that limits the target size of the recommended configuration.

Recommendations are produced by performing a heuristic search on the space of possible configurations $C_{j_1}, \ldots, C_{j_n}$ that satisfy the budget B, while minimizing the *estimated* cost of execution of (possibly a sample of) the workload $\mathcal{W}$. That is, none of the configurations considered by the configuration recommender are actually built during the recommendation phase. The estimated performance of a given *hypothetical* configuration C_{j_k} considered by the configuration recommender is obtained by feeding the RDBMS *query optimizer* with parameters that describe C_{j_k} (e.g., statistical information about indexes in C_{j_k}). The parameters describing C_{j_k} are also *estimated* by the query optimizer [6, 14].

The use of the RDBMS query optimizer by the recommender tool has several practical advantages. Most notably, it avoids potential mismatches in estimated query costs for a given configuration, as the same estimator (i.e., the query optimizer) is used for both recommending configurations and executing queries. In practice, this eliminates the risk of having indexes that are recommended but never used in the execution plans selected by the optimizer. However, the recommender tool ends up relying extensively on the RDBMS query optimizer producing accurate estimates for the costs of executing queries (it is well known that the quality of such estimates degrades severely as query complexity increases [13]). The recommender is also vulnerable to estimation errors in the optimizer when assessing the parameters of the hypothetical configurations considered.

We note that there has been work on improving the accuracy of estimates based on observing actual query execution costs. The learning optimizer LEO described in [15] is an example of a technique that has been shown to continuously improve cardinality estimates on a commercial optimizer. However, this approach has limitations when applied to a recommender: since LEO depends on observed executions in the *current* configuration, the technique may not improve estimator accuracy for *hypothetical* configurations (as there is no way to observe these executions without changing the current system configuration).

Recall that Figure 6 shows the actual execution times of the three configurations P, R and 1C for family NREF3J on System B. Figure 10 shows similar cumulative frequency curves for family NREF3J on System B but looking at the values of the optimizer estimates (represented here by arbitrary units and not as seconds). Five different curves are presented in the figure. The curves EP, ER and E1C are the estimates for the queries provided by the optimizer when the system is in the configuration P, R and 1C respectively. The optimizer correctly estimates that the behavior of R improves over P and that 1C improves even further. However, the magnitude of the improvements is quite conservative compared with the improvements seen in the actual executions (again, refer to Figure 6).

The recommender does not have access to the ER and E1C curves while the system is in the configuration P. Instead it can obtain hypothetical estimates HR and H1C for the configurations R and 1C (respectively) while on the current configuration P. The curves HR and H1C also appear in Figure 10. Note that while the curve for HR almost coincides with the curve for ER, the curve for H1C is much more conservative about the advantages of 1C than E1C. While

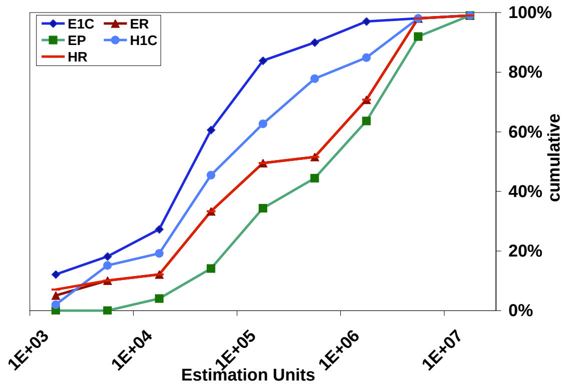

Figure 10: Cumulative curves for the estimates for family NREF3J on System B.

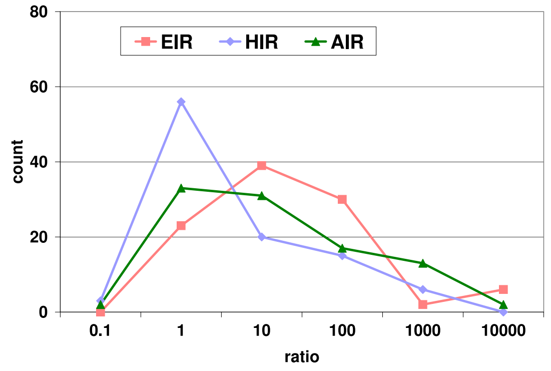

Figure 11: Histograms for the improvement ratios for family NREF3J on System B.

in this example the relative goodness of HR and H1C are correct, the data shows that the accuracy of the hypothetical estimations is not always closely related to the estimates taken at the target configuration.

5.2 Measuring Improvements

We define the *estimated improvement ratio* of query q_k when estimated on configuration C_i compared to configuration C_j as the ratio $EIR(q_k) = E(q_k, C_i)/E(q_k, C_j)$; similarly, the *actual improvement ratio* of changing between configurations C_i and C_j is $AIR(q_k) = A(q_k, C_i)/A(q_k, C_j)$. For simplicity, actual improvements involving timeout queries are not considered.

We also define $H(q_k, C_h, C_a)$, the *hypothetical* cost of executing query q_k in the *hypothetical configuration* C_h, which is an estimate obtained while the system is in the *actual configuration* C_a as discussed in the previous section. Since in our experiments we are interested in evaluating hypothetical configurations in P, we define the *hypothetical improvement ratio* of query q_k when hypothetically estimated on P for configuration C_i compared to the hypothetically estimation on P for configuration C_j as the ratio

$$HIR(q_k) = H(q_k, C_i, P)/H(q_k, C_j, P).$$

We are interested in the improvement ratios that compare the recommended configuration R to the reference configuration 1C. Larger than 1 ratios indicate that R will do worse, while smaller ratios indicate how much faster R is than 1C for a given query.

Figure 11 shows the histograms (instead of cumulative polygons) of the three improvement ratios defined above comparing R to 1C for the queries in family NREF3J on System B. Looking at the actual executions (the AIR histogram curve), we see that 31 queries are 10 times faster in 1C than in R and 17 queries are 100 times faster in 1C than in R, while 33 queries show no improvement at all (ratio 1). When we look at the hypothetical estimates obtained in the configuration P (the HIR curve) we see that 20 queries are 10 times faster in 1C than in R and 15 queries are 100 times faster in 1C than in R, while 56 queries show no improvement at all. The HIR curve states that the R and 1C configurations would (hypothetically) perform much closer than they actually do. In contrast, looking at the estimates obtained in the configurations 1C and R (the EIR curve), we see improvement ratios that much more clearly favour 1C over R (in fact, even more so than the actual ratios AIR).

The previous example further illustrates the point made in the preceding section regarding the discrepancies between estimates and hypothetical estimates. The configurations are evaluated by the recommender on the basis of hypothetical estimates in the current configuration, since it has no access to estimates in the target configurations (although these estimates could be much more accurate than the hypothetical ones).

6. CONCLUSION

This paper introduces a benchmarking framework for assessing the quality of autonomic configuration recommenders. We propose a flexible notion of workload performance, we employ large and diverse query workloads described by families of similar queries, and we identify comprehensive single column indexing as a very useful configuration for baseline comparisons. Our use of curves depicting the cumulative frequencies of query execution times to characterize workload performance bring forward the advantages of designing recommenders that can accept *quality of service goals* specified by constraints on these curves.

Using the proposed framework, we describe three benchmarks using real and synthetic data and several classes of families which are used to provide the first assessment in the literature of current commercial configuration recommendation tools. We believe that the extensive experimental results we report have substantial value as they not only confirm the practical applicability of the approach proposed, but also demonstrate that improvements of several orders of magnitude can still be achieved.

We can regard the experimental data collected from our experiments as the missing observation step in the *observe, predict and react loop* applied to autonomic indexing. Current recommenders *predict* based on estimates and hypothetical configurations and *react* by recommending a new configuration but there is no attempt to *observe* the actual cost of query execution.

We note that the proposed benchmark methodology can be extended to support the evaluation of non-relational systems, such as recently proposed XML-based recommender tools [19].

Conducting extensive experimental evaluations are a first step towards assessing and improving the effectiveness of existing relational recommenders. More significantly, meaningful benchmarks form the basis for the scientific evaluation of future research in autonomic recommenders.

Acknowledgments. We thank Yannis Ioannidis, Sam Lightstone, Arun Marathe, Vivek Narasayya, Danny Zilio, and the referees for their feedback. This work was supported in part by grants from the Natural Science and Engineering Research Council of Canada. D. Barbosa was supported in part by an IBM PhD. Fellowship. A. Teisanu was supported in part by an IBM CAS Fellowship.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *Proceedings of 30th International Conference on Very Large Data Bases*, pages 1110–1121, Toronto, Canada, 2004.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.

[3] H. Boral and D. J. DeWitt. Database machines: An idea whose time has passed? a critique of the future of database machines. In *Proceedings of the International Workshop on Database Machines*, 1983. Reprinted in Parallel Architectures for Database Systems. IEEE Computer Society Press, 1989.

[4] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *SIGMOD*. ACM Press, 2002.

[5] S. Chaudhuri and V. R. Narasayya. TPC-D Data Generation with Skew. Available via anonymous ftp from `ftp.research.microsoft.com/users/viveknar/tpcdskew`.

[6] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*. Morgan Kaufmann Publishers Inc., 1997.

[7] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*, 1998.

[8] S. Chaudhuri and V. R. Narasayya. Microsoft Index Tuning Wizard for SQL Server 7.0. In *SIGMOD*, 1998.

[9] W. Cleveland. *The Elements of Graphing Data*. Hobart Press, 1994.

[10] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of 30th International Conference on Very Large Data Bases*, pages 1098–1109, Toronto, Canada, 2004.

[11] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing Iceberg Queries Efficiently. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pages 299–310, New York City, New York, USA, August 24-27 1998.

[12] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[13] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[14] S. S. Lightstone, G. Lohman, and D. Zilio. Toward Autonomic Computing with DB2 Universal Database. *ACM SIGMOD Record*, 31(3), 2002.

[15] V. Markl, G. M. Lohman, and V. Raman. LEO: An Autonomic Query Optimizer for DB2. *IBM Systems Journal*, 42(1), 2003.

[16] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 2004.

[17] M. Poess, B. Smith, L. Kollar, and P. Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 582–587. ACM Press, 2002.

[18] M. Poess and J. M. Stephens. Generating Thousand Benchmark Queries in Seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 1045–1053, Toronto, Canada, August 31 – September 3 2004.

[19] K. Runapongsa, J. M. Patel, R. Bordawekar, and S. Padmanabhan. XIST: An XML Index Selection Tool. In *XSym*, 2004.

[20] T. Sawyer. Doing your own benchmark. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[21] Transaction Processing Performance Council. *TPC Benchmark H - Decision Support*, 1999. Revision 1.3.0.

[22] G. Valentin, M. Zuliani, D. C. Zilio, and A. S. Guy Lohman. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *ICDE*, 2000.

[23] C. H. Wu, H. Huang, L. Arminski, J. Castro-Alvear, Y. Chen, Z.-Z. Hu, R. S. Ledley, K. C. Lewis, H.-W. Mewes, B. C. Orcutt, B. E. Suzek, A. Tsugita, C. R. Vinayaka, L.-S. L. Yeh, J. Zhang, , and W. C. Barker. The protein information resource: an integrated public resource of functional annotation of proteins. *Nucleic Acids Research*, 30, 2002.

[24] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor . In *Proceedings of the International Conference on Autonomic Computing*, 2004.

[25] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of 30th International Conference on Very Large Data Bases*, pages 1087–1097, Toronto, Canada, 2004.