Experimental Evaluation of Autonomic Indexing

Denilson Barbosa
University of Toronto
Toronto, Canada
dmb@cs.toronto.edu

Mariano P. Consens
University of Toronto
Toronto, Canada
consens@mie.utoronto.ca

Laurent Mignet*
IBM India Research Laboratory
New Delhi, India
lamignet@in.ibm.com

Technical Report CSRG-495, University of Toronto, July 2004

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 index configurations for a given workload are prominent examples of this promising trend. In this paper, we introduce a flexible characterization of the performance goals of an indexing recommender and develop an experimental evaluation approach to assess the effectiveness of these 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 indexing 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).

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 [5, 9]) are crucial first steps toward autonomic data management. Agrawal *et al.* [1] 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 [6]. The input to the process is typically a query workload and a budget (often in as a bound on the disk space that can be used for additional indexes). In this approach, a tool enumerates several "interesting configurations" that do not exceed the budget, and uses the query optimizer to find one with the lowest cost. The cost of a given configuration is obtained by feeding the query optimizer with the description of the configuration and the query workload. In order to describe a given configuration, the tool must estimate cardinalities and selectivities for the hypothetical indexes, and, since such indexes can be defined as queries, estimates provided by the query optimizer can also be used during this phase. Currently, most major DBMS vendors support tools that behave in this way [7, 15].

One potential limitation of the model described above is that they rely on estimated statistics for parameters such as size and cardinality of certain queries, and it is well known that the quality of such estimates degrades severely as more operations are performed [8].

The potential impact on the system's performance of an effective index configuration dwarfs any other system parameter that a database administrator could tune. For instance, more than two decades ago

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

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

Figure 1: Relational schema for the NREF database; primary keys are underlined.

Boral and DeWitt [2] concluded that parallelism is no substitute for effective and efficient indexing. Therefore, the practical value of index recommender tools could be very significant.

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 Nonredundant REFerence protein database (NREF, for short), published on the web by the Protein Information Resource [16]. NREF is provides a comprehensive collection of protein sequence data from several genome sequencing projects (PIR-PSD, SwissProt, 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 in Figure 1. The Protein relation (1.1 million rows) contains a unique identifier for each of the aminoacid 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 aminoacid 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.

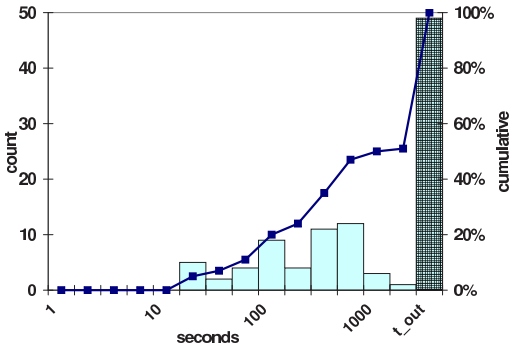

Figure 2: Query execution times on system A using a configuration with primary keys only.

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
```

For concreteness, 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 2 shows the response times of a commercial system (which we call system A) on 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 logarithmic scale; also, we report all "timeout" queries on a single bin, labeled `t_out` in the figures; for this work, we define a timeout limit of 30 minutes for each query.) Contrast that histogram with the response times observed on the same system A, but using a configuration with several recommended indexes shown in Figure 3. Not only there

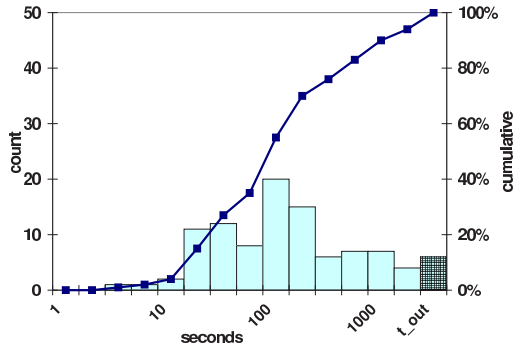

Figure 3: Query execution times on system A using a recommended configuration.

is a significantly smaller number of timeout queries in the recommender system, but also the proportion of queries that complete in about 5 minutes is much larger.

The lines in the figures are the cumulative histogram for the queries in each system, which, as we argue more precisely below, provide a concise way of comparing the behavior of different configurations on the same system. For instance, we read that 55% of all queries finish in at most 100 seconds (each) on the recommend configuration; a similar reading in Figure 2 shows that only 20% finish in 100 seconds or less.

1.2 Our Contributions.

The focus of literature describing index recommenders has been on reporting results characterizing how *efficiently* these tools arrive at useful recommendations; that is, how quickly they can produce a solution to the combinatorial problem of index selection. However, to the best of our knowledge, there has been no comprehensive assessment of the effectiveness of index recommenders, as the ones we discuss here.

In this paper, we describe an approach to evaluate the quality of the index configurations suggested by index recommenders. We present extensive experimental results characterizing the effectiveness of commercial RDBMS index 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 index recommenders. In particular, our framework supports describing very large workloads and also provides a flexible characterization of performance goals.
- We present results on state-of-the-art commercial RDBMS index 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 index recommenders: do not overlook the potential gains brought by single column indexes.

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 [11], but without considering the recommendation of indexes.

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 an index 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 Recommender Task

In broad terms, the basic task of an index recommender is to select a new configuration for the RDBMS system that improves the performance of 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 [1, 17]).

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 [3]. 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

Notation. 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 configuration C_i to C_j.

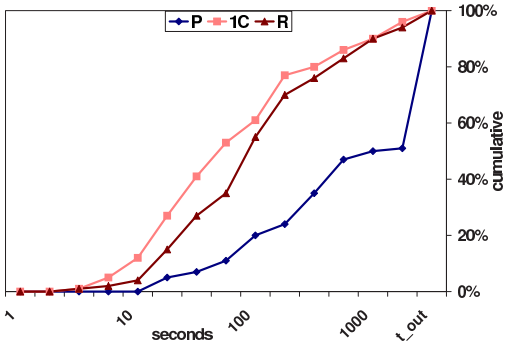

Figure 4: Behavior of system A on NREF2J.

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 2 and 3 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 objective and accurate way of comparing different configurations; for instance, Figure 4 compares three configurations, called P, 1C and R (which will be explained later) on system A. The figure shows that configuration 1C is superior to both R and P: 57% of the queries take complete in up to 1 minute (each) on configuration 1C, while for R and P, only 35% and 27% (resp.) finish in 1 minute or less.

A Model for Current Index Recommenders. We can describe the behavior of the index recommenders incorporated in commercial RDBMS [7, 15]

using the framework discussed above. The RDBMS index recommender takes as input a given workload $\mathcal{W}$, including the relative frequencies of the queries in the workload. The recommenders goal is to select a configuration C_j that improves the (frequency-weighted) total estimated cost of queries $E(\mathcal{W}, C_j)$, subject to an estimated storage budget (hence $ET(C_i, C_j)$ uses storage as the measure). The index 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 index recommender relies on a heuristic search to compute estimates for a subset of the configurations.

Performance Goals. A performance goal for the index 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" [13].

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 2 minutes, and 95% to complete before a 30 minute timeout. This goal can be described by a step function:*

$$\begin{aligned} G(x) &= 0, && x < 10 \\ G(x) &= 0.1, && 10 \leq x < 120 \\ G(x) &= 0.5, && 120 \leq x < 1800 \\ G(x) &= 0.95, && x \geq 1800 \end{aligned}$$

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 4, 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

There are a number of challenges in designing experiments that can adequately evaluate the performance of recommenders. We have to select a suitable database and an instance that is part of the initial configuration (or, more generally, a method for generating instances). Then, we need to provide a number of workloads $\mathcal{W}_1, \ldots, \mathcal{W}_k$ and goals and obtain a corresponding number of recommended configurations $C_{j_1}, C_{j_2}, \ldots, C_{j_k}$. To evaluate the *quality* of the configurations selected by the recommender we can use *reference configurations* $C_{h_1}, C_{h_2}, \ldots, C_{h_k}$ as reference points to compare $A(\mathcal{W}_l, C_{j_l})$ to $A(\mathcal{W}_l, C_{h_l})$.

Valentin *et. al* [15] provide an example of a qualitative evaluation of an index recommender based on the TPC-D version benchmark. The database and initial configuration are as defined in the 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 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, which is a very encouraging result: the comparison configuration used is expected to perform extremely well and hence matching its performance is quite an accomplishment. While this evaluation is useful, it is important to realize that the TPCD workload has only a small number of queries (that in addition, are well-known to the RDBMS implementations).

Our approach to evaluate index recommenders addresses the challenges described above using two

mechanisms. First, we model query workloads as *query families*, which are sets of queries that contain a large number of related yet suitably diverse queries. Second, we identify a *single* reference configuration that is used as the reference point in all the workloads. Our reference configuration has a single column index for each possible (indexable) column in the schema (and we refer to it as 1C, for 1-column index).

Design of the Query Families. The following criteria were used for designing the families we use in this work. First, the queries should have a meaningful interpretation. One way to achieve this is 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 Figure 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. To facilitate this, we use only select-project-join SQL queries defining simple aggregate functions and with at most one level of nesting. Third, queries that require the materialization of large intermediate results should be avoided, as they could make irrelevant the presence of indexes in the database. To achieve this, we use additional selective predicates for each query. Finally, the queries in the family should cover a reasonable spectrum of query execution times, from fast (e.g., subsecond respose) to slow (e.g., a timeout after a reasonable long execution time).

In our experimental evaluation we use three databases that are scaled appropriately to the computing resources available in the desktop computing environment we utilize. The databases selected are the NREF database discussed in Section 1.1, TPC-H [14], and a skewed version of the TPC-H [4], generated with a Zipfian factor of 1. Due to the exploratory nature of our motivating scenario, we focus on queries that 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. We note that although we have experimented with several families with a wide range of characteristics, we summarize our results using two families for the NREF database, and three families for the TPC-H databases.

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 a 10GB TPC-H database generated with skewed data (using a Zipfian factor of 1). For each query, 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 $R \bowtie S$. Up to three $\theta(S.c_3)$ are chosen such that the final query results are not empty; also, the three constraints selected cause the intermediate result sizes for $R \bowtie S$ to take values k_1, k_2 and k_3, where 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 θ(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 restricted to be chosen from 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 10GB 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).

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. We choose to report here best results achieved by each system, regardless of the machine used.

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 in isolation, and the machine is fully dedicated to running the experiments. For obvious practical reasons, a timeout limit of 30min 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.

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.

Despite the reduction in size, running just both NREF families on all configurations and systems remains a daunting task: it may require $(485 + 373) \times 3$ runs $\times$ 7 systems/configurations $\times$ 30min $= 9,009$ 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.

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.

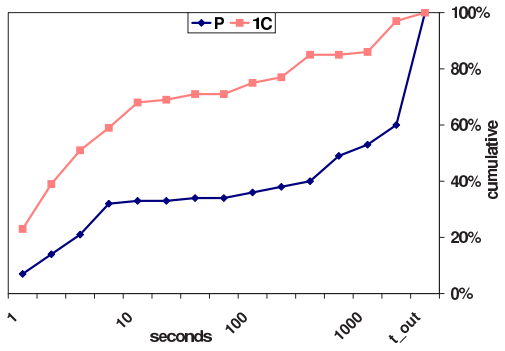

Figure 5: Behavior of system A on NREF3J.

4.1.2 Configurations Tested.

In each experiment, the initial configuration contains only those indexes automatically created for the primary keys of each table; we call this configuration **P** (for primary-key indexes only). The *comparison* configuration is defined by adding to the P configuration individual indexes on each indexable column in the schema; we call this configuration **1C** (for 1-column indexes). One *recommended* configuration is used for each query family in each experiment, produced by the index recommender using: the P configuration as starting point, a single query family as workload, the difference in size of 1C and P as budget, and no limit on the time the recommender is allowed to run.

As a 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.

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 scenarios, which is generous to the recommenders; in fact, as Table 1 shows, no recommended configuration uses as much space as 1C. (We also obtained recommendations with an unlimited storage budget and in most cases observed no significant difference compared to those constrained to a budget.)

We note that we were not able to obtain recommendations for family NREF3J using system A. We tried with a few other samples of 100 queries from family NREF3J, as well as 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.

Summary of Recommendations. 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.

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 4 in Section 2.2 describes the behavior of system A for family NREF2J. Besides the substantial improvements in both A_NREF_1C and A_NREF2J_R relative to A_NREF_P, the figure shows that the recommended configuration behaves much closer to A_NREF_1C for queries that require more than 177 seconds to complete. Figure 5 shows the result of System A on family NREF3J. The graph

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 ⋈ 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.

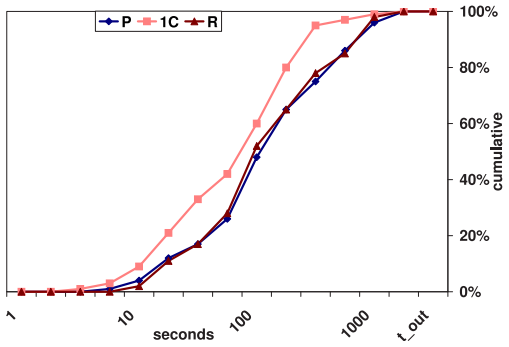

Figure 6: Behavior of system B on NREF2J.

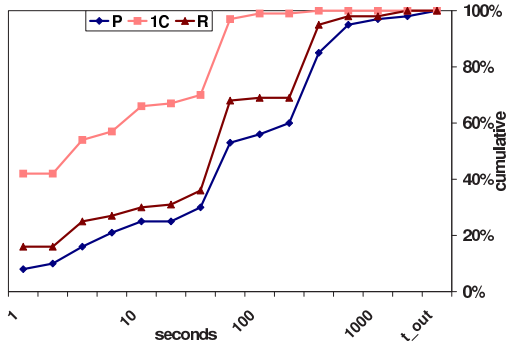

Figure 7: Behavior of system B on NREF3J.

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! As discussed above, despite the vast benefit provided by indexes, system A was unable to produce a recommendation for this family.

Figures 6 and 7 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 (and sometimes far) superior to both P and the recommended configurations we found. Moreover, a careful look at Figures 5 and 7 shows wide gaps between the P and 1C configurations, indicating large potential performance improvements by the use of indexes. Therefore, we arrive at the surprising observation

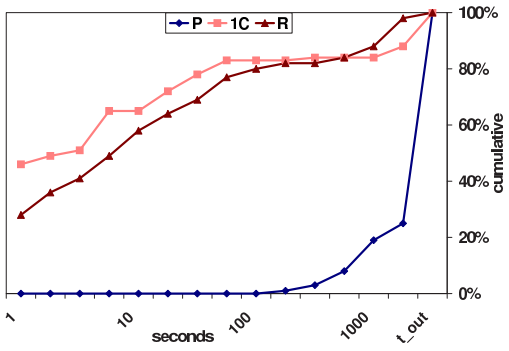

Figure 8: Behavior of System C on SkTH3Js.

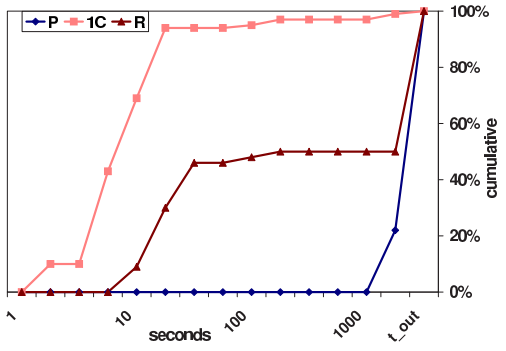

Figure 9: Behavior of System C on SkTH3J.

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

We note that the behavior observed for the NREF benchmark is consistent with the behavior observed on the synthetic datasets (Figures 8, 9 and 10). Notably, the 1C configuration is very competitive with, and most of the times far superior to the recommended configurations. This happens, despite the fact that some of the recommendations use even non-trivial materialized views defined on joins of base tables, which makes the relative performance of 1C truly remarkable.

The only recommendation in all our experiments to outperform 1C (even though only for a small fraction of the workload) was obtained on family SkTH3Js (see Figure 8). A comparison of Figures 8 and 9 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 index recommender on the input workload.

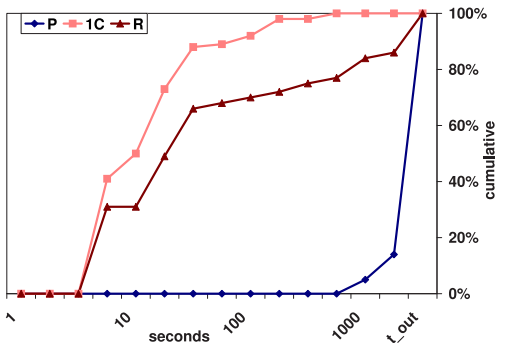

Figure 10: Behavior of System C on UnTH3J.

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 9 and 10. Clearly, the recommender did perform much 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 respectively (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!

5 Recommender Evaluation

Recall the discussion in Section 2.2. Current RDBMS index recommenders take three inputs: an existing system configuration C_i, a query workload $\mathcal{W}$, and a space budget B that limits the 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 index recommender are actually built during the recommendation phase. The estimated performance of a given *hypothetical* configuration C_x considered by the index recommender is obtained by feeding the RDBMS *query optimizer* with parameters that described C_x (e.g., statistical information about indexes in C_x). The parameters describing C_x are also *estimated* using the query optimizer [5, 9].

Given that query optimizers are also heuristic tools, the model described above 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 the index recommender and the RDBMS. In practice, this eliminates the risk of having indexes that are recommended but never used. However, the model relies extensively on the RDBMS query optimizer producing accurate estimates for costs of executing queries and also for the various parameters describing the hypothetical configurations enumerated by the index recommender. This dependency is a potential limitation of the model as it is well known that the quality of such estimates degrades severely as more operations are performed [8].

We note that there has been work on improving the accuracy of estimates based on observing actual execution costs; LEO, the learning optimizer described in [10], is an example of a technique that has been shown to improve estimates on a commercial optimizer. However, this approach has its own limitations, as LEO depends on *observed executions* in the current system configuration, thus, it is not clear how it could be used for improving the estimates for hypothetical configurations (as there are no executions to observe on them).

Another potential limitation of the model is that it is *founded* on the assumption that estimated costs (seen by the index recommender) of hypothetical configurations are *identical* to estimated costs on those configurations *if they were implemented*. That is, the index recommender has an accurate picture of how each hypothetical configuration would behave if implemented. Based on this assumption, we use the estimated query execution costs given by the RDBMS to compare the expected (as seen by the index recommender) and the actual improvements achieved by the various configurations we used.

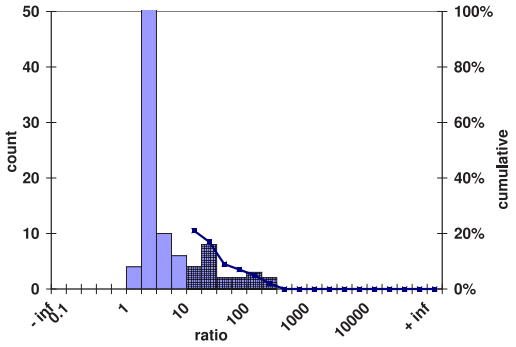

Figure 11: Estimated improvement achieved by changing from B_NREF_P to B_NREF3J_R.

5.1 Estimated and Actual Improvements

We define the *estimated improvement* of query q_k when moving from configuration C_i to configuration C_j as the ratio $EI(q_k) = E(q_k, C_i)/E(q_k, C_j)$; similarly, the *actual improvement* of changing between configurations C_i and C_j is $AI(q_k) = A(q_k, C_i)/A(q_k, C_j)$. (Actual improvements for timeout queries are defined as follows; if q_k does timeout in both C_i and C_j, then $AI(q_k) = 1$; if q_k does timeout on C_i only, we define $AI(q_k)$ to be "$+\infty$"; finally, if q_k does timeout on C_j only, then $AI(q_k)$ is "$-\infty$".)

Recall Figure 7 (page 9) showing the behavior of the three configurations for family NREF3J on System B. Figure 11 shows a distribution of the queries in that family by their estimated improvements when moving from P to the configuration recommended by System B. In other words, by the assumption discussed above, the figure shows the best improvements that the index recommender believed would be obtained by any configuration within the given budget. Figure 12, on the other hand, shows the actual improvement achieved by changing the same configurations. Note that we report one query with an "infinite" actual improvement.

We will focus our discussion on improvements (or losses) of more than one order of magnitude. Hence,

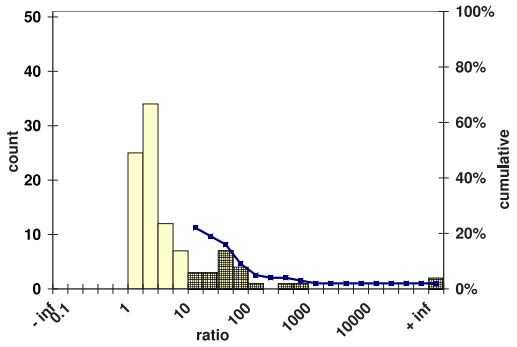

Figure 12: Actual improvement achieved by changing from B_NREF_P to B_NREF3J_R.

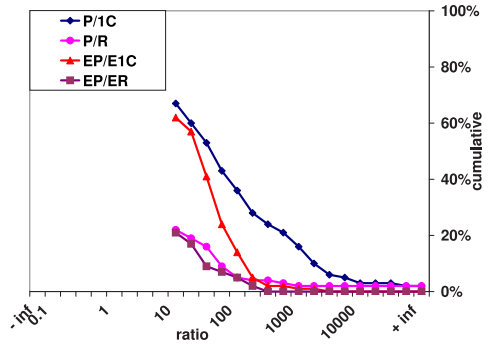

Figure 13: Improvements on family NREF3J for System B.

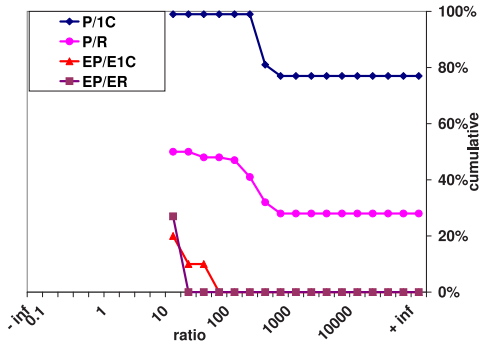

Figure 14: Improvements on family SkTH3J for System C.

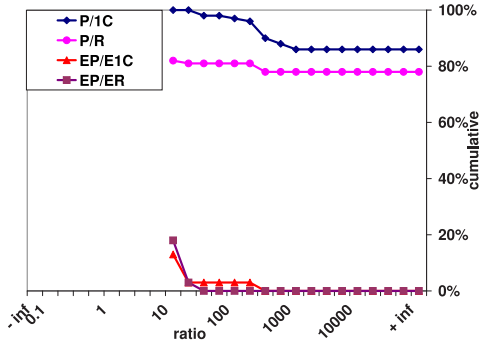

Figure 15: Improvements on family UnTH3J for System C.

we ignore the queries whose improvement lie in the interval (0.1,10) and we highlight the remaining queries by shading their bins in Figures 11 and 12. The lines in these two figures show the decreasing cumulative histograms corresponding to the shaded portion of the histogram. The decreasing cumulative histograms allow us to read in Figure 12 that 17% of the queries had an actual improvement of 10 times or more, while 2% of the queries had an actual improvement of 100 times or more.

The decreasing cumulative histograms described above can be superimposed for comparing estimated and actual improvements of more than one configuration as well. Figure 13 compares the estimated and actual improvements for the 1C and B_NREF3J_R configurations relative to the P configuration. The graph shows that the actual relative improvement of configuration B_NREF3J_R is very close to the estimated improvement observed by the index recommender. However, the actual improvement achieved by the 1C configuration is far greater than what is estimated; this means that even if the 1C configuration was considered by the index recommender, its benefits would have been underestimated. Figure 14 shows a similar analysis for the three configurations tested with System C for the SkTH3J family (recall Figure 9 on page 10). Note that in this case, the benefits of both 1C and C_SkTH3J_R as seen by the index recommender are grossly underestimated. A similar behavior is observed for the configurations using the uniform TPC-H data as well (Figure 15).

6 Conclusions

This paper introduced a novel framework for a qualitative assessment of the performance of index recommenders. We propose a broader notion of workload performance, large and diverse query workloads described by families of similar queries, and we identify comprehensive single column indexing as a very useful baseline comparison configuration. Using this framework, we describe three benchmarks using real

and synthetic data and several classes of families which are used to provide the first assessment of current commercial index 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. Conducting extensive experimental evaluations are a first step towards assessing and improving the effectiveness of existing relational recommenders. The value of these experiments extends also to recently proposed XML-based recommender tools [12].

We can also regard the experimental data collected from our experiments as the missing *observation* step in the *observe-predict-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.

Finally, our use of histograms to characterize performance goals highlights the value of recommender systems that can accept *quality of service* goals specified as performance histograms.

Acknowledgments. 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.

References

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

[2] 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.

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

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

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

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

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

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

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

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.