1992

M.L. Kersten, F. Kwakkel

Design and implementation of a DBMS performance assessment tool

Computer Science/Department of Algorithmics and Architecture Report CS-R9270 December

CWI is het Centrum voor Wiskunde en Informatica van de Stichting Mathematisch Centrum
CWI is the Centre for Mathematics and Computer Science of the Mathematical Centre Foundation

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Design and Implementation of a DBMS Performance Assessment Tool

M.L. Kersten
CWI, P.O. Box 4079
1009 AB Amsterdam
The Netherlands
{Martin.Kersten@cwi.nl}

F. Kwakkel
CWI, P.O. Box 4079
1009 AB Amsterdam
The Netherlands
{Fred.Kwakkel@cwi.nl}

Abstract

The increasing number of advanced database management systems offered on the market requires tools to quickly assess their performance and to assure their quality. Performance measurement involves running a set of representative workloads, such as benchmarks, and quality assurance, which involves extensive testing. The SOFTWARE TESTPILOT [1] described in this paper greatly simplifies both jobs by enabling a compact specification of the workload search space, a flexible mechanism to interact with a system under study, and a fast algorithm to expose the performance bottlenecks or software instabilities.

1991 CR Categories: **Testing and debugging (D.2.5)** *diagnostics,* **Database systems (H.2.4)** *performance assessment,* **Installation management (K.6.2)** *benchmarks.*
Keywords and Phrases: Software testing, quality assurance, benchmarking.

1 Introduction

Database technology for single processor architectures has matured from file processing by a large number of individual programs to sound data models and data manipulation schemes applicable in many environments. The DBMSs are delivered as portable, single processor implementations with many techniques to obtain good performance. However, new (prototype) systems often exhibit low quality and low performance due to lack of extensive field tests in the application areas they intend to support. Inclusion of new techniques into existing architectures also hamper from unpredictable side-effects on system performance and stability. Yet, from an economic point of view it is mandatory to predict their performance and stability long before the hardware and software are installed.

The common approach to obtain performance characteristics of a DBMS is by offering it a set of carefully chosen queries and to observe its behavior. Database benchmarks such as Wisconsin[3, 2, 5] and AS3AP[8] primarily highlight the internal strengths and weakness of a system implementation. The TPC-A and TPC-B benchmarks[9] have the advantage of being based on a real application (a bank teller network), but have become divorced from the real applications requirements. They can only be used to provide a simple comparative measure of a small part of the system being tested. Current TPC work is directed towards providing benchmarks for more complex traditional application domains (e.g. Order-Entry, Decision Support) with an emphasis towards conventional systems.

The role of benchmarking to assess performance and as a means to improve software quality is limited. The known benchmarks do not directly aid users to determine the DBMS effectiveness for a particular domain. Domain-specific benchmarks are a response to this diversity of computer system use. They specify a synthetic workload characterizing a typical application problem domain.

[1] The work reported here is funded by ESPRIT-III Pythagoras project (P7091)

Report CS-R9270
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The OO1 [1] benchmark is a step in this direction, which addresses the perceived performance critical aspects of a Computer-Aided Design (CAD) application.

The benchmarks are complemented by test suites geared at exposure of weaknesses in the system implementation. Together they provide the basis for technical quality assessment. Extension with software quality assessment techniques, such as ISO9000, further improve the organization of software production. Finally, the alpha- and beta- test sites form the final barrier for systems to reach the market.

A major drawback of the benchmarks and test-suites, addressed in this paper, is that they represent only a few points in the workload search space. Therefore, a DBMS engineer or user with a workload characteristic slightly off those measured may face a badly performing system. Moreover, system implementors may be inclined to provide good performance on the published benchmarks. In particular, attaining good performance on the notorious complex operations may ignore the workload characteristics of the application domain. Likewise, test-suites may be biased by isolated implementation challenges, such as novel data structures or query optimization techniques, and neglect possible side-effects.

The SOFTWARE TESTPILOT described in this paper is a tool to aid the DBMS engineer and user to explore a large workload search space to find the slope, top, and knees of performance figures quickly. The approach taken is based on specifying the abstract workload search space, a small interface library with the target system, and a description of the expected behavior. Thereafter, it is up to the SOFTWARE TESTPILOT to select the actual workload parameter values and to execute the corresponding DBMS transactions, such that the performance characteristics and quality weaknesses are determined with minimal cost (=time).

The novelty of our SOFTWARE TESTPILOT is its use of a behavioral model of the target system to drive this process. The system uses the performance hypothesis and performance data gathered to identify candidate workspace parameters with high probability to (dis-) proof the hypothesis. Among the candidates it selects one of high value and low cost to reach. Moreover, the SOFTWARE TESTPILOT adjusts the behavioral model when measurements disprove the models' assumptions. Thus, the outcome of a session is both improved statistical knowledge on the performance characteristics and a (possibly) modified behavioral model of the target system.

Although testing and benchmarking [7] have received considerable attention over the last decade we are not aware of similar attempts to automate performance quality assessment at this level.

Test environments mostly focus on test-data generation, path coverage testing, or they explore the programming language properties to proof (type) assertions. The predominant pragmatic approach is to use shell scripts to capture the repetitive nature of the test runs or to rely on a small test space. The major drawback is that it often results in a brute force approach, running the script over night to obtain a few points of the performance function without adjustment or error recovery.

Performance monitors, built into the DBMS or operating system environment, are passive. They merely provide measurement data, statistical analysis, and visualization tools. Instead, the SOFTWARE TESTPILOT provides a mechanism to partially automate the interpretation of the results to prepare for and to perform the next test run.

A recent paper on I/O benchmark performance analysis [6] has some aspects in common with our approach. Like the SOFTWARE TESTPILOT , they attempt to identify the knee of a performance function as quickly as possible. However, their approach is focussed on a particular problem area. It has not been generalized into a general technique to support a larger application domain or to exploit the knowledge gained from a hypothesis.

The remainder of this paper is organized as follows. Section 2 introduces the scope of the problem, the approach taken by the SOFTWARE TESTPILOT , and the textual Test-suite Specification Language. Section 3 introduces the architecture of the SOFTWARE TESTPILOT and its core algorithms. We conclude with an outlook on future research issues.

2 A Test-suite Specification Language

In this Section we introduce the Test-suite Specification Language (TSL) for the SOFTWARE TESTPILOT . A TSL program, an object-based extension of the underlying implementation language Prolog, consists of object descriptions and their refinements. Each object type has a small number of attributes with system defined defaults. The program can be prepared with a text editor or interactively as illustrated in Figure 9.

2.1 The performance assessment problem

To obtain a better appreciation of the role of a SOFTWARE TESTPILOT in performance assessment, we use the Wisconsin Benchmark for relational DBMSs [5] as our running example. This benchmark has been designed in the early eighties to assess the performance of relational systems. It has been gradually extended to become a standard benchmark for assessing new implementation techniques [3] for relational systems.

The Wisconsin benchmark has a fixed set of 32 queries focused on single user query processing performance. One aspect being measured is the response time to access part of a table. After running many cases, two representatives were chosen. Namely, retrieval of 1% and 10% of a table with 10K tuples. Since, these numbers have become less relevant for new DBMS implementations. First, the table size has been increased to better reflect current database sizes and the hardware platform [4]. Second, selection of 1% out of a 1.000.000 tuples is probably too optimistic and the fraction may have to be scaled down as well.

This performance assessment problem can be summarized as the task to identify the shape of the performance function Q : *table_size* $\times$ *selection_size* $\rightarrow$ *response_time*. The table size and selection size are the input variables or experimental **factors**. The selection time is the output or **response** variable. This performance function is obtained by careful selection of input parameters, setting up the environment for the experiment, and to run the experiment until a statistical stable response value has been obtained.

The SOFTWARE TESTPILOT is designed to automate a major portion of this manual task as follows. The user reformulates the task as a performance test program in terms of factor and response parameters and actions on the target system. The action list includes descriptions to initialize the DBMS and to change its state incrementally. Given a test program, the task of the SOFTWARE TESTPILOT is then to select and execute a sequence of actions, such that the performance relationship(s) between the factors and their responses are determined with minimal cost.

The novelty of the SOFTWARE TESTPILOT is its use of an anticipated behavioral model to drive this process. That is, the user provides a **hypothesis**, i.e. a continuous function, that describes the expected performance relationship as accurately as possible. It is used by the SOFTWARE TESTPILOT to locate a point in the test space that is likely to disqualify the hypothesis given.

In general, multiple runs are required to obtain a reliable response function. If a large deviation is observed then the hypothesis is discarded by the system. Thereafter, a new hypothesis is derived from the points already measured. For example, failure to observe linear behavior may cause the SOFTWARE TESTPILOT to attempt a logarithmic distribution instead. Furthermore, a session is automatically terminated when a sufficient approximation of the hypothesis, in terms of cost and deviation, has been established.

For example, assume that the SOFTWARE TESTPILOT has already obtained responses for Q as shown in Figure 1. The straight line denotes the hypothesis for Q given by the user. To verify this hypothesis, a new point must be chosen with maximal probability to dis-qualify the claim made. A strategy is to take a point with the maximum distance from any two adjacent points already measured. In this case, the next point to be tested would be a table size of 80K (= (100+60)/2). Before this can be done 20K tuples must be inserted.

Thus, the test suite program should include actions to change the state of the database, i.e. to support insertion and deletion of dummy tuples. These actions are used by the SOFTWARE

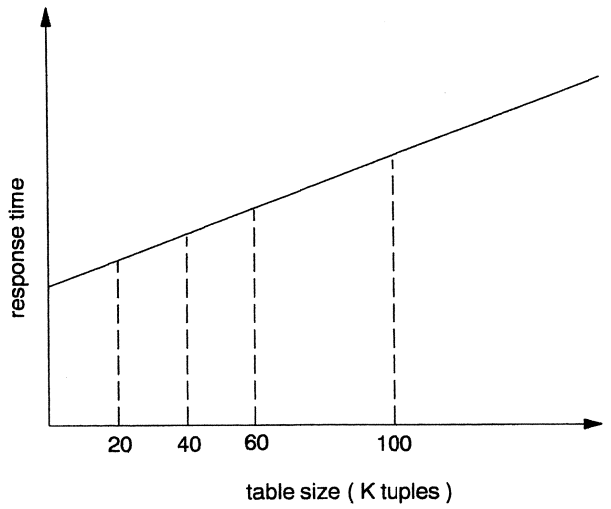

Figure 1: Performance hypothesis and points measured

TESTPILOT to move the DBMS into the required state. Once the state is reached, the query can be executed to (dis) proof the hypothesis.

2.2 Factor and response definition

The basic building blocks of a test suite specification are the **factor** and **response** variables. They require a symbolic name for reference within the TSL program and a type name from the collection *(integer, symbolic, time, float, string, char, boolean)*. The type can be further constraint by specifying a sorted list of value ranges (e.g. [0..10,990..1000]) or an enumeration of categories and symbolic constants (e.g. [max,min,sum]).

The type values can also be described by a generator, i.e. a unary Prolog predicate activated upon system restart to produce a finite term list. For example, a generator can be used to produce terms for the distinct n-way joins over a relational schema.

Object refinement can be used to simplify a large test space description. A refinement is indicated by the attribute **like** bound to another variable. The attributes of the dependent variable are taken as default until they are explicitly redefined.

The mechanism to select the next value for a factor is controlled with the attribute **apply**, which is an element from the collection {**cyclic**, **random**, **optimize**}. These items can be overruled with a Prolog clause in cases where more precise control is needed on the exploration of the test space.

The partial specification shown in Figure 2 illustrates some definitions for the Wisconsin benchmark.

2.3 Focus definition

The test space spanned by the variables is often too large to be investigated with a single run. Therefore, the user can identify a subset as the prime **focus** of attention. Each focus bears a name for reference within the TSL programming environment and a listing of the factor and response variables.

The variables outside the focus remain fixed at their initial value and performance relationships outside the current focus are only inspected if they involve no additional cost. The relative importance of different foci is administered by a **weight** attribute (>0), which causes the SOFTWARE TESTPILOT to balance the effort investigating the foci.

```
factor => [     name:           table_size,
                type:           integer,
                apply:          optimize,
                range:          [ 0 .. 1000k ]
].

factor => [     name:           selection_size,
                type:           integer,
                range:          [ 0 .. 1000k ],
].
response => [   name:           selection_time,
                type:           time,
].
focus => [      name:           dbfocus,
                factors:        [table_size, selection_size],
                responses:      [selection_time:T],
                pattern:        [ table_size, selection_size, selection_time ],
                check:          (T < 1000)
].
hypothesis => [ factors:        [table_size:X, selection_size:Y],
                responses:      [selection_time:Z],
                guess:          Z = linear(X,Y,1)
].
```

Figure 2: Sample TSL specification

The optional attribute **check** takes a predicate to limit the space considered for experimentation. Each experiment leads to a measurement vector that can be sent to a post-processing system for further analysis. The content of the vector is described by the attribute **pattern**.

Figure 2 contains a single focus identifying interest in the (partial) performance functions over the space `table_size`$\times$ `selection_size` $\rightarrow$ `selection_time`. The constraint T<1000 indicates that the SOFTWARE TESTPILOT must avoid actions that exceed 1000 time units.[2]

2.4 Hypothesis definition

The dependency of a response variable on its factors is described with a **hypothesis** introduced by the user or deduced by the SOFTWARE TESTPILOT from earlier experiments. The current implementation only supports the former, because induction of a hypothesis from data gathered is a statistical problem initially best addressed using a (separate) statistical package.

The envisioned performance function is described by the attribute **guess**, which contains a linear, logarithmic, or exponential equation. The applicable sub-domain is described with a condition in attribute **check**. How the hypothesis can be used to select points for exploration is described in Chapter 3.2. In Figure 2 we expect `dbselect` to be linear dependent on `table_size` and `selection_size`.

2.5 Action definition

An **action** describes an experiment on the target system or a way to change the state of the target system to prepare for the next experiment. An action is characterized by its name, factor and response variable(s). Although all factors in the TSL program determine the response value, only

[2]An attribute can be tagged with a variable name, i.e. an identifier starting with an uppercase, to denote its value in other expressions within the same object.

```
action => [     name:           dbselect,
                factors:        [ selection_size:S],
                responses:      [selection_time:T],
                body:           sql_select(S,T)
].
action => [     name:           dbinsert,
                factors:        [table_size:X],
                step:           [table_size:1],
                body:           sql_insert(X)
].
action => [     name:           dbdelete,
                factors:        [ table_size:X],
                step:           [ table_size:N],
                cost:           (N),
                body:           sql_delete( X, N )
].
```

Figure 3: Action objects

those used as action parameters or those changed as a side-effect in an action should be explicitly mentioned.

For example, consider the action `dbselect` in Figure 3. Its body contains a call to a parameterized SQL query. This SQL command does not critically depend on the `table_size`, which is considered a constant factor taken from the focus setting.

A possible side-effect of an action is a modification of the factors to reflect the new state of the target system. These side-effects are described with the attribute `step`, which contains factors tagged with an arithmetic expression to derive the destination point in the test space. They are crucial for the SOFTWARE TESTPILOT to infer the state of the target system and, thereby, to determine the course of actions taken.

The variable and focus constraints can be refined with the attribute **check** to allow for actions covering different cases. For example, the TSL program may contain two actions to insert tuples in the relation. One could generate a single tuple and another generates a batch of 100 tuples.

The **body** describes the interaction details with the target system. To support a wide range of target systems and experimentation techniques its value is a Prolog term. Interfacing the target system then depends on the foreign language facilities of the underlying Prolog implementation of the SOFTWARE TESTPILOT . A small library can be used to capture the interface requirements for a class of target systems, such as the SQL-server library currently in use.

In some cases, such as during system restart and error recovery, the SOFTWARE TESTPILOT should enquire the target system on its status. Enquiry actions are recognized by omission of **step** and **response** properties. They can be applied unconditionally at any position in the test space.

Actions come with an estimated **cost** defined by the user or inferred by the system from previous experiments. They are used to plan the course of actions.

Figure 3 illustrates the framework of several actions. The action `dbselect` represent our prime performance interest; the others are used by the SOFTWARE TESTPILOT to change the state of the DBMS. The details on how the actions interact with the SQL server are not shown.

2.6 Monitoring and control

The SOFTWARE TESTPILOT comes with a graphical user interface to develop TSL programs and to monitor progress of an experimental session (See Figure 9). In particular, a monitor can be opened to inspect the performance relationship for any variable combination being measured. A form-based interface supports browsing of the TSL objects and their on the fly modification. A

```
monitor => [    factors:        [table_size, selection_size],
                response:       selection_time,
                display :       true,
                rate:           1
].
controls => [   speed:          1,
                buffer:         30,
                plans:          30,
                startup:        [table_size:0],
                selftest:       true,
                simulate:       false,
                distort:        0.2
].
```

Figure 4: Monitor and session controls.

TSL program developed interactively and the performance functions displayed can be reproduced in textual form for inclusion in other documents.

A SOFTWARE TESTPILOT session can be stopped and re-started at any time. Upon restart the system automatically attempts to obtain the actual state of the target system when **selftest:true**. Variables whose initial value can not be determined this way take a default value described in **position**.

The behavior of the SOFTWARE TESTPILOT can be further controlled by redefinition of system properties as illustrated in Figure 4. They are explained in more detail in the remaining sections.

3 Design of the Software Testpilot

The SOFTWARE TESTPILOT is designed around four processes that communicate through data pools. A system overview and design rationale is given in Section 3.1. Then, the algorithm to select candidate points from the test space (Section 3.2) and the algorithm to construct a flight plan (Section 3.3) are described in greater detail. The Chapter concludes with the selection and execution of the best plan.

3.1 The Software Testpilot's Processing Cycle

The processing cycle for the SOFTWARE TESTPILOT is a repetition of flight plans during which information on the performance relationship(s) is gathered. This process stops when all interesting points have been inspected, when a fatal-error occurs, or when the user terminates further exploration of the test space.

A flight starts with selection of a target point in the test space where measurements should be taken. From the large collection of possible points the SOFTWARE TESTPILOT identifies a few candidate that provides the best information to (dis-) qualify the hypotheses applicable within the focus of interest.

Before the measurements can be taken, the SOFTWARE TESTPILOT should move the DBMS into the required state, that is, tuples must be inserted/deleted. The action(s) to reach this state is (are) inferred from the **factor** and **step** attributes of action descriptions. Often, several feasible plans exist and a choice should be made to avoid time consuming plans.

For instance, consider a TSL program to investigate tuple selection from table sizes between 0 and 100 tuples. To (dis-) qualify a linear hypothesis quickly it could investigate table sizes in the order: 0, 100, 50, 25, 75, 13, 38 etc. That is, table sizes lie far away from those already inspected. However, the corresponding flight plans are also expensive. For, after the first tuple selection

```
position → [      selection_size:10 , table_size: 30 ].

pointDone → [     selection_size:10 , table_size: 100, selection_time: 1.67].
pointDone → [     selection_size: 90, table_size: 100, selection_time: 1.45].
pointDone → [     selection_size: 25, table_size: 50, selection_time: 0.46].

candidate → [     selection_size: 30, table_size: 100, weight: 0.95].
candidate → [     selection_size: 15, table_size: 50, weight: 0.65].
candidate → [     selection_size: 5, table_size: 50, weight: 0.73].

flightPlan → [    point : [ selection_size:20, table_size:50 ],
                  actions : [ dbdelete(100,50), dbselect(50,30) ],
                  cost : (50+30)*0.95 ].
flightPlan → [    point : [ selection_size:15, table_size:50 ],
                  actions : [ dbdelete(100,50), dbselect(50,15) ],
                  cost : (50+15)*0.65 ].
flightPlan → [    point : [ selection_size:5, table_size:50 ],
                  actions : [ dbdelete(100,50), dbselect(50,5) ],
                  cost : (50+5)*0.73 ].
```

Figure 5: The data pools with state information

100 tuples are inserted. Then again one tuple is selected and 50 tuples are deleted to bring the database to a state of 50 tuples, etc..

A way out of this dilemma is to consider several candidate points and to apply the cheapest flight plan. For instance, if three candidate points are handled together then the execution order becomes: 0, 50, 100 .. 75, 25, 13, etc. The first batch (0,100,50) is sorted on the cost (=unit steps) to reach the point. Then the next three points are generated (25,75,13) and sorted. Now 75 is the 'cheapest' point because its distance to 100 is the minimum of the three points. The execution order becomes: 75,25,13 etc.

If we carry this idea further and allow the SOFTWARE TESTPILOT to generate all candidate points at startup then the sequence becomes: 0, 1, 2, 3 etc.. This is something we do not want either, because it provides only slow insight into the shape and validity of the performance functions. Therefore, we use a buffer of candidates and give the user a choice between an emphasis on the execution time, the 'cost' of the flight plan to reach that point, and emphasis on selecting interesting points, i.e. the 'weight' of a point (See Section 3.4).

The internal architecture of the SOFTWARE TESTPILOT uses four data pools for interprocess communication. Their roles is as follows:

- *position*, which describes the current position of the SOFTWARE TESTPILOT in the test space.
- *pointDone*, which contains the response values obtained and the setting of all TSL input variables (Figure 5.top).
- *candidate*, which contains the buffered candidate points. Each point consists of the focus input values and a **weight** of interest for this point.
- *flightPlan*, which contains a flight plan for each candidate point. A flight plan consists of the target point, a sequence of actions, and the estimated cost.

The four processes involve selection of candidate points, generation of flight plans, selection of a single flight plan, and its execution. They are described in more detail in subsequent sections.

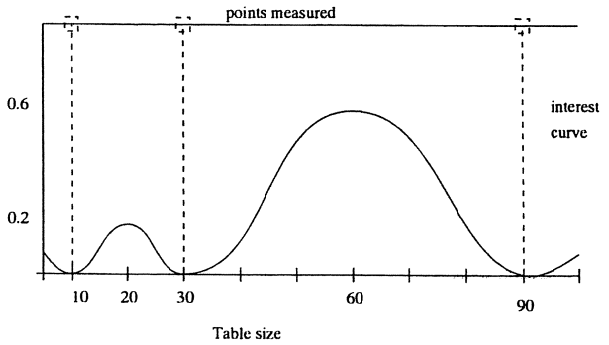

Figure 6: The interest curve for candidate points

3.2 Selection of candidate points

The first process deals with the selection of a reasonable number of candidates for inspection. Considering too many candidate makes subsequent selection of the target point expensive. Conversely, considering only a few candidates may lead to time consuming experiments, because it becomes less likely to inspect the more interesting point, as indicated in the previous section.

To solve this resource management problem, we limit the candidate pool to those with a (normalized) weight exceeding a system defined threshold. This threshold is lowered when the system runs out of good candidates. The result is a system behavior that gradually explores less-and-less interesting points up to a user interrupt or reaching a predefined termination condition. The mechanism to determine the weight is introduced by an example.

Assume that class *pointDone* contains the points depicted in Figure 6 on the first line. The challenge is to find a metric that identifies the interesting points. This depend on how we intend to use the SOFTWARE TESTPILOT . If we aim for identifying the shape of a performance function then the candidates close to an already measured point are of less interest, due to expected measurement errors. Conversely, if we are interested in robustness of the target system points then near the minimum and maximum of the domain are of high interest. To support a wide range of application domains, the user can specify their interest function.

In our example, we focus on identifying the shape of the continuous performance function. The interest curve for the linear hypothesis is shown in Figure 6, which favors a candidate X_i exactly in between two points X_{i-1} and X_{i+1} identified before. It is a weight proportional to the distance of its neighbors, i.e. $weight(X_i) = D * sin((X_i - X_{i-1})\pi/2D)$ with $D = X_{i+1} - X_{i-1}$. This leads to considering 20 and 60 tuples as candidates for inspection.

This idea has been generalized to deal with several hypotheses as follows. Let C denote the set of old candidates and points investigated, let H denote the set of hypothesis defined, and F the focus. Then, the algorithm in Figure 7 generates for each factor A_i in a hypothesis a candidate point between any two known points (*pointDone* and *candidate*). Factors that do not appear in a hypothesis definition do not contribute to the collection of candidates. To make the algorithm work, initial 'corners' of the test space must be stored in the candidates pool upon system initialization. The algorithm can be improved by adding points to the buffer pool after each measurement.

3.3 Generation of flight plans

The second process generates a flight plan for each candidate in two phases. The first phase deals with moving the target system into the test space position identified. The second phase extends the flight plan with the actual measurement actions. At the same time the SOFTWARE TESTPILOT determines the cost and expected response time of the plan. The cost is used to order the batch

for each hypothesis $h : f_l \times .. \times f_r \rightarrow r_o$ **do**
Let $c \in C$ with $c = (a_1..a_{l-1}, a_l..a_r, a_{r+1}..a_n)$
where $a_l..a_r$ are values for the hypothesis parameters $f_l..f_r$
and $a_1..a_{l-1}$ and $a_{r+1}..a_n$ are values for the remaining factors.
for each $f \in \{f_l..f_r\}$ **do**
group C by f
def: $G_i = (a_1..a_{i-1}, a_i, a_{i+1}..a_n) \mid a_1..a_{i-1}$ and $a_{i+1}..a_n$ are constant.
for each group $G_f[k]$ **do**
sort $G_f[k]$ with key f
for each subsequent pair **do**
generate a new candidate point using the interest curve.

Figure 7: Generation of candidates

process makeplan()
clear flight plan FP and set cost C to zero
PHASE I:
Let $Q(u_i..u_l)$ be current database state.
Let $Q'(v_i..v_l)$ be the target database state.
$S(s_i..s_l) := Q(u_i..u_l)$
repeat
Find an action $A = A(s_j..s_k)$ that brings S closer to Q'.
Determine the new state S' using the **step** attribute.
$FP := FP{+}{+}\ A(s_j..s_k)$
$cost := cost + estimate(A(s_j..s_k))$
until $S = Q'$
PHASE II:
for each response r **do**
find an action A with $A(f_c..f_e) \rightarrow r$
substitute the relevant focus values from Q'.
$flightplan = flightplan{+}{+}\ A(v_c..v_e)$

Figure 8: Flight plan generation algorithm

of flight plans for subsequent execution. The response time is returned to an interactive user, who can use it to intervene (or have lunch). A sketch of the algorithm is shown in Figure 8.

For example, the candidate selected in the previous section requests a selection experiment of 5 tuples from a table with 90 tuples. Assume that in the current position the table size contains 60 tuples.

Then, the first phase must insert 30 tuples to reach the desired point. The eligible action is `dbinsert`, whose **factors** and **step** attributes permit table_size increments. Thus, the flight plan can be initialized with the request `dbinsert(30)`. The second phase identifies the actions whose response variable is mentioned in the focus of interest. They are added to the flight plan. In our example, we merely add the request `dbselect(90,5)`, but in general several selections are performed at once.

3.4 Selection of the best flight plan

The third process deals with selecting a flightplan for execution. This choice depends on two factors: the weight of the measuring point and the flight plan cost. The user can balance these

factors using the *speed* property in the control object, which takes a value in the range 0.0 to 1.0.

A speed factor 0.0 causes the flightplan with the highest weight to be chosen; the cost to execute the flight plan is ignored. The system tries to (dis) proof the hypothesis with a minimal number of test runs. A speed of 1.0 leads to selecting the cheapest plan, but may delay insight in the precise shape of the performance function.

Actually, let F_i denote the plans under consideration then the plan taken into execution minimizes the formula:

$$'value' = speed * \alpha + (1 - speed) * \beta$$

where *alpha* and *beta* are normalized costs and weights, defined as

$$\alpha = (cost(F_i)/maxcost) \text{ and}$$
$$\beta = (weight(F_i)/maxweight).$$

3.5 Flightplan execution

The flight plan selected above is executed on the DBMS and the results are kept for inspection by the SOFTWARE TESTPILOT . The interface between SOFTWARE TESTPILOT and target system is kept as open as possible, because it permits non DBMS tools to be considered as well, e.g. a event-queue simulation package. Moreover, we expect generic modules to defined that encapsulate interaction with a particular target system.

To aid design and debugging of a TSL program, we have included a dummy target system. This dummy uses the information within the SOFTWARE TESTPILOT to mimic the behavior of a real system. In particular, it uses the hypothesis to predict the outcome of an action with some distortion.

4 Summary

In this paper we have described a novel approach to assess the performance quality of database management systems. The technique consists of specifying a potential large test space through which the SOFTWARE TESTPILOT navigates using a behavioral model of the anticipated performance characteristics. The system automatically selects the best actions to move the DBMS into the proper experimental state by insertion/deletions. Thereafter, it performs the measurements.

The current implementation of the SOFTWARE TESTPILOT is written in SWI-Prolog [10]. It runs on Silicon Graphics workstations using the XPCE toolkit for monitoring the performance functions.

The SOFTWARE TESTPILOT is currently used for assessing the performance gains to be expected for browsing session using new query optimization techniques. This requires particular strength in the generation of symbolic data. Moreover, a prototype implementation for parallel execution of flight plans has been constructed. The SOFTWARE TESTPILOT is delivered with an SQL server module and a dummy database generator.

Acknowledgments

The authors wish to thank the members of the Database Research Group of CWI and partners in the Pythagoras project for advice on earlier versions of this systems. In particular, we thank R. Ulenbelt for their contribution on the implementation of the prototypes.

References

[1] Cattell, R. "Engineering and Database Benchmark", Technical Report, Database Engineering Group, Sun Microsystem, April 90.

[2] D. Bitton, D.J. DeWitt, and C. Turbyfill, *Benchmarking Database Systems: A Systematic Approach*, CS Report #526, Univ. of Wisconsin at Madison, Dec 1983.

[3] D. Bitton *The Wisconsin Benchmark* in Readings on Database Systems, Editor: M. Stonebraker, Morgan-Kaufmann, 1988.

[4] R.J. DeWitt, S. Ghandeharizadeh, D. Schneider, R. Jauhari, M. Muralikrishna, A. Sharma, *A Single User Evaluation of the Gamma Database Machine*, in Database Machines and Knowledge Base Machines, M. Kitsuregawa (ed.), Kluwer Academic Publ. 1988, pp. 370-386.

[5] D.J. DeWitt, *The Wisconsin Benchmark*, in The Benchmark Handbook for database and transaction processing systems (by J. Gray), Morgan Kaufmann Publishers Inc., 1991.

[6] P.M. Chen and D.A. Patterson, *A New Approach to I/O Benchmarks- Adaptive Evaluation, Predicted Performance* Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).

[7] Gray,J., "The Benchmark Handbook", Morgan Kaufmann Publishers Inc., 1991

[8] Terbyfill C., *AS3AP - An ANSI SEQUEL Standard Scalable and Portable Benchmark for Relational Database Systems*, DB Software Corporation, 1989

[9] Transaction Processing Performance Council, "TPC Benchmark, A Standard", Omeri Serlin, ITOM International Corp. Nov. 89

[10] J. Wielemaker, SWI-Prolog Manual, University of Amsterdam, 1992.

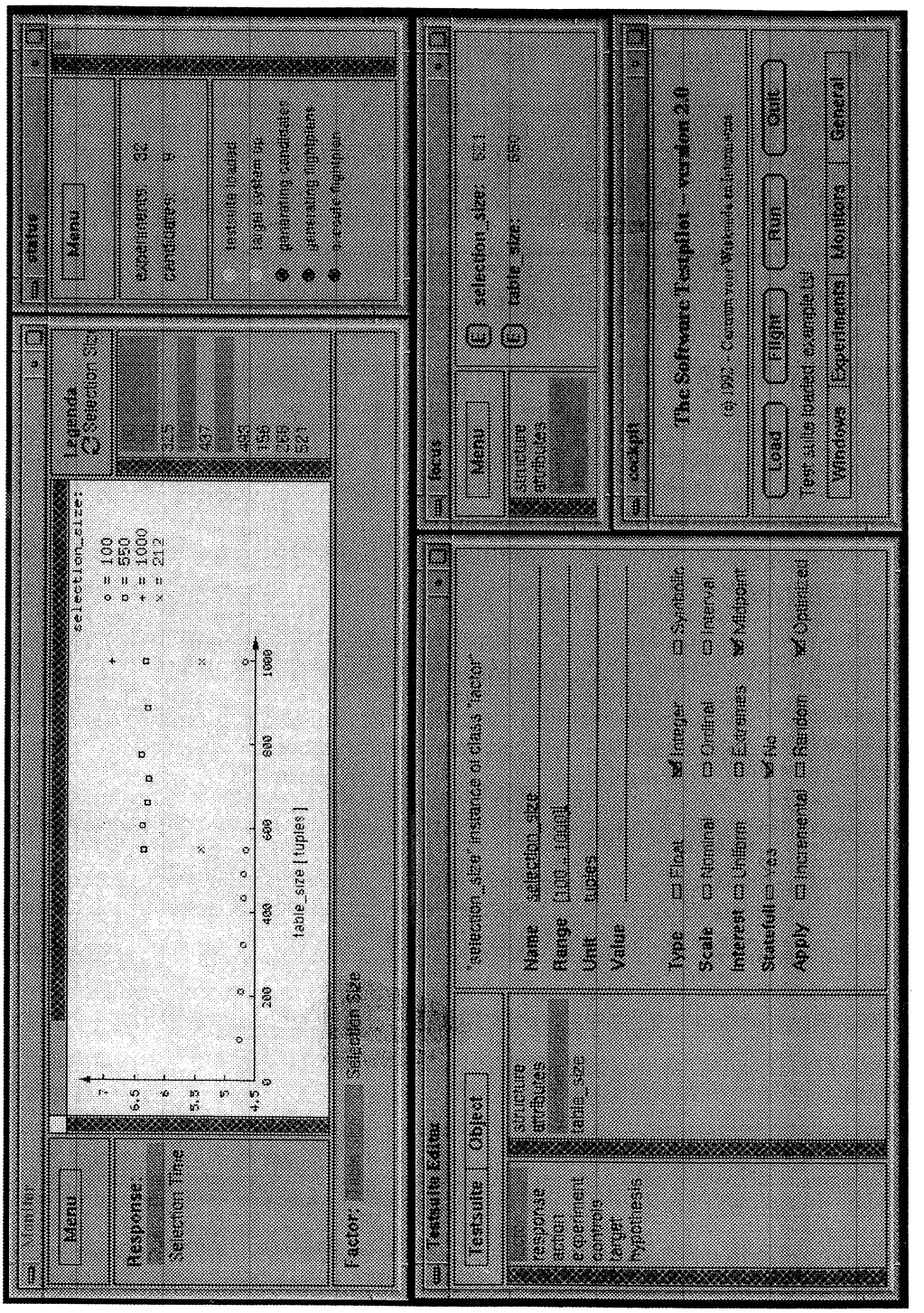

Figure 9: the cockpit