UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Adaptive Selection of Query Processing Strategies

José I. Icaza
Data Structuring Group

Research Report
CS-87-46

August, 1987

Adaptive Selection of Query Processing Strategies

by

Jose I. Icaza

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1987

Abstract

Approximate cost models that are traditionally needed to evaluate query execution strategies are sometimes unavailable or too unreliable for particular environments. Our new approach for strategy selection is based on feeding back the *actual* costs of query execution under different strategies, rather than on assumptions-loaded *estimates* of these costs. We propose three different methods for using these feedback costs, under a common general framework for adaptive systems.

In *optimal selection* we define the problem of designing an *optimal decision policy* for a single query class, where the policy specifies which strategy to execute for each query. This problem is mapped to some versions of the *bandit problem* in statistics, which can be solved using bayesian statistics and dynamic programming. We present and analyze a program that derives the optimal policy.

In *approximate selection* we use a learning automaton to select strategies. The learning algorithm by Thathachar and Sastry is modified to handle the dynamic environments of databases. This adaptive algorithm is tested using existing databases and query loads, showing how this approach determines the best execution strategies over time, and changes which strategies are selected as the environment changes.

Finally, the method of *query class partitioning* considers the problem of evolving optimal query partitionings, in which each partition includes all queries that have a common optimal strategy. We present a novel adaptive classification scheme that uses several learning automata.

Acknowledgements

It is a pleasure to acknowledge the help of my supervisor, Frank Tompa. I specially appreciate his flexibility and patience in adapting to several changes of research topic; his availability, guidance, and his human warmth. The other members of my examination committee contributed useful suggestions: Professors Stavros Christodoulakis, Christopher Genest, J. Howard Johnson and Clement Yu. Professor C.C. Gotlieb of the University of Toronto and Dr. W.A. Walker of Ontario Hydro kindly provided the tape of query measurements used in Chapter 5. Informal conversations with R. Baeza, J. A. Blakeley, P. Celis, P.A. Larson and M. Mota often resulted in useful insights. My wife Mariau endured numerous changes in my plans, to the detriment of her own professional development, and provided me with constant support, encouragement and generous help in the last stages.

I am grateful for financial support received from the Natural Sciences and Engineering Research Council of Canada under grant A9292; the Canadian Secretary of State Centres of Specialization Program; and the University of Waterloo.

To Mariau, Marifer and Leonardo

Contents

List of Tables

List of Figures

Chapter 1

The need for adaptivity in query processing

One of the primary tasks of a query processor is the selection of efficient *execution strategies* for queries. This involves choosing access paths to the required data and selecting algorithms to be used to answer the query.

Typically, the selection algorithm incorporates a *cost model* that enables it to predict the approximate cost of each strategy. This model is usually based on numerous assumptions about the physical data structures, the nature of the data and the characteristics of the query load. When the assumptions do not hold, the model becomes unreliable in predicting costs, and the selection of strategies may not be adequate. Further, there are situations in which no suitable cost model is available at all.

We propose a new approach where the selection of strategies is based on the *actual past performance*, rather than on the predicted performance of strategies. This brings in the benefit of adaptation: as the performance varies due to changes in the environment, the selection of strategies adapts to these changes. Further, as we will see, this adaptive approach is based on very weak assumptions, providing great generality and simplicity.

1.1 The traditional approach to query processing

The process of answering queries from a database can be naturally divided into two parts: strategy selection and query execution (see figure 1.1). The strategy selector will typically parse and transform the query in various ways, and eventually decide on an execution strategy for the query; the query executor will then follow this strategy to fetch the answer.

Often, there is more than one way to answer a query; the selector must then decide which strategy to use based on estimated costs of each strategy. These costs are often computed using a mathematical model of the storage structures of the database; see for instance Selinger [80] and Dayal and Goodman [31] for typical cost models.

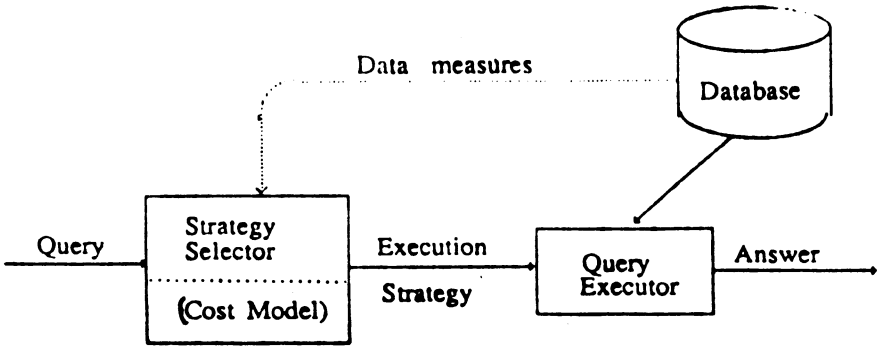


Figure 1.1: Traditional query processing

Models usually assume that some *data measures* are available, including data statistics (sizes, distributions), data placement (clustered or non-clustered) and access structures (availability and type of indexes and links). This is represented as a dotted line in figure 1.1. Models vary widely in the amount and kind of data measures that they assume available.

For tractability, numerous simplifying assumptions are usually made about selectivities, distribution of records in blocks, query load characteristics and so on. The model will then produce a cost figure that will be exact only when these assumptions are precisely met; more often, the cost model will produce a pessimistic value [26].

In the next two sections, we will present two general situations where the traditional approach is inadequate or cannot be used. Section 1.2 describes in general the kind of errors that can be made when typical assumptions are partially or totally invalid. Section 1.3 considers cases where there is no suitable cost model available.

1.2 Unreliability of cost model

Christodoulakis [25,26] has observed that the simplifying assumptions used in cost models often do not hold; the models may then produce grossly inaccurate cost estimates and potentially wrong choices of strategies. The models are usually designed for an "average" environment, but a particular database environment may be far from average, far from the assumptions made by the cost model.

As a simple example of how things can go wrong, consider the set of single-relation queries of the form "print records where X=a and Y=b", where the attribute names X and Y are the same for all the queries but the desired values a and b can vary between queries. Suppose that both attributes X and Y are indexed, and the query processor must decide which index to use; so the execution strategies are:

1. Use index on X to retrieve all records with X=a. On each of these records, check whether Y=b; if so, print the record.

2. Use index on Y to retrieve all records with Y=b. On each of these records, check whether X=a; if so, print the record.

Let the cost be measured by the number of disk blocks that have to be accessed.

We will use the data measures and assumptions employed by System R [80]. The only measures available are the total number of records and blocks occupied by the relation and the number of distinct values in indexes. Suppose there are 100 records in 10 blocks, 10 distinct values of X (from 1 to 10) and 20 of Y. Assuming uniform and independent distributions of the attribute values *in the stored data*, we can *expect* 10 of the 100 records to have X=a, and 5 to have Y=b. The second assumption is that the records are uniformly distributed in the blocks; that is, each qualifying record has equal probability to land in any of the 100 available spots. With this assumption, we get an *expected* cost of 6.66 for strategy 1 and 5 for strategy 2 [1], assuming the cost of using the index is negligible. Since strategy 2 is cheaper, it will be selected for *all* queries of this type.

Suppose now that the first assumption is violated for attribute X: the first half of the X values (from 1 to 5) only have one record occurrence each, and the other half have 19 occurrences each. By bad luck, if all the queries happen to ask for X values between 1 and 5, then the *real* costs for each of these queries will be 1 for the first strategy, and 5 for the second. Clearly then, the cost model will make the wrong strategy choice all the time.

Modelers therefore often make yet a fourth assumption that is not usually made explicit: attribute values are uniformly distributed *in the query load.* If satisfied, the right strategy choice will be made half of the time —with good luck, perhaps more often. For more complicated examples involving complex selection conditions and access to several relations, even more assumptions have to be made.

Several attempts have been made to improve on this situation. One obvious improvement is to have more data measures available. For the particular example considered, the indexes might keep exact selectivity values; this incurs time and space overheads. Piatetsky-Shapiro and Connel[70], and Kamel and King [55] give methods to compute approximate selectivities of individual values. Christodoulakis [25,24] suggests the fitting of multivariate probability distributions to the data, in order to improve record and block selectivity measures. These papers are limited to single-relation queries. Demolombe [36] gives a precise cost model that requires numerous detailed measures, including selectivities and correlations within and across relations. It is not clear that the additional work required to obtain and update data measures pays off in better selection of strategies.

Database management systems are general packages; one hopes that, over the life of the system in thousands of databases, the right choice of execution strategies will be made more often than the wrong choices, but there is no guarantee that this will hold for any particular environment. What is really needed is the ability of the selection mechanism to *adapt to its environment.*

[1] This is based on the formula of Bernstein et al. [13] that approximates Yao's exact formula [90].

1.3 Unavailability of suitable cost models

There are situations where the traditional approach cannot be used because suitable analytic cost models are not available.

As an example, consider a high-level *database management system independent* query language that must run on top of a large number of database management systems of diverse characteristics. Queries expressed in this language would be translated to the query language or data manipulation language of the underlying system. The different translations possible for a given query correspond to our execution strategies. How can the system choose the best translation?

It is not feasible to have an analytic cost model for each possible circumstance. The target systems may be commercial packages whose internal structures are not fully known —all we know is their interface languages. Also, the systems may be running on a wide variety of hardware, or even remotely, making the cost-model approach impractical.

1.4 A new approach

The situations considered in the previous sections have in common an uncertain or unpredictable environment, which is where adaptive approaches best fit. We can realistically assume that the exact cost of executing a query is available *after* the query has been executed. Our approach explores ways of making use of these real costs to improve the selection of strategies. What we propose is to form a feedback loop between the strategy selector and the query execution engine, as shown in figure 1.2.

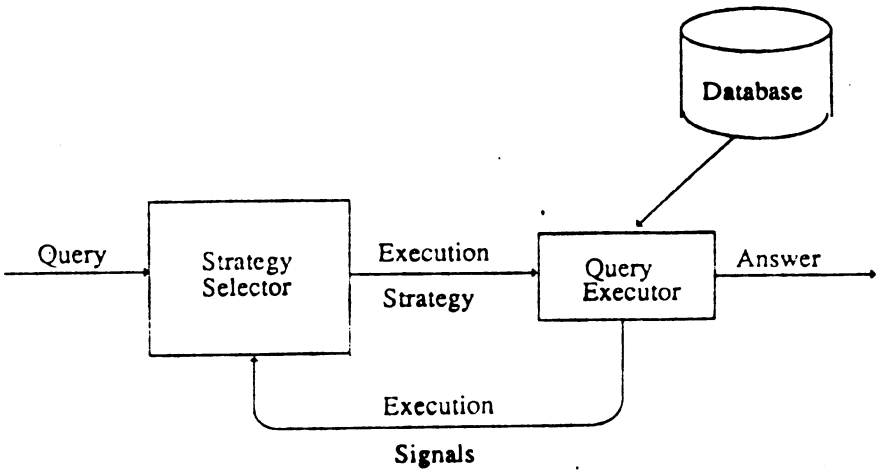


Figure 1.2: Adaptive query processing

The selector then receives some *signals* from the execution of the query; these signals enable it to learn about its environment over time and improve the choice

of strategies. In particular, we will consider extensively the case where the only such signal is the cost of query execution. For the trivial example of section 1.2, the adaptive approach will quickly notice that, *for the query load that the system is experiencing*, strategy 1 is better than strategy 2 and begin choosing the first one almost exclusively. The algorithms we propose will be able to adapt to changes in the query load or other aspects of the environment.

Saridis [78], writing about adaptive control systems, observed that adaptive controllers have been successfully applied to practical systems only in a few cases where the uncertainty of the environment has prevented the use of standard controllers. The reason for this is obvious: no system that has to infer part of its environment over time will improve upon a system that knows it beforehand. Therefore, we do not expect the adaptive approach to beat the traditional approach when the latter is based on a carefully tailored cost model that makes meaningful assumptions in environments that satisfy the assumptions. Rather, we address our work to the situations where suitable cost models are not available, environments change frequently, or assumptions cannot be met.

1.5 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 establishes a general framework for adaptive systems and includes a review of adaptivity in several fields, including databases and query processing. The formal framework is used to describe the components of a particular kind of adaptive query processing: adaptive strategy selection. This is further particularized in chapters 3 to 6, where we consider *cost based* adaptive strategy selection. Chapter 3 addresses the problem of determining optimal adaptive strategy selection policies; this turns out to be computationally impractical. Chapter 4 considers approximate policies. Simulation results using real query loads are presented in chapter 5. Chapters 3 to 5 assume that queries have been classified according to equivalence of strategies; in chapter 6 we consider a further sub-classification scheme that partitions queries adaptively according to patterns of attribute values and the associated costs of strategies. The last chapter presents conclusions and suggestions for further research.

Chapter 2

Strategy selection as an adaptive system

The term "adaptive" has been used with various meanings. Whether a system is considered adaptive depends on one's definition of adaptivity; further, the boundaries of the system have to be carefully stated, since something can be adaptive on one level and not on another level.

In this chapter we will develop a formal framework for adaptive systems based on the work of Holland [53]. This framework is grounded in General Systems Theory; its purpose is to identify characteristics of adaptivity that are common to all kinds of adaptive systems, natural or man-made. It specifies the components of these systems; it also suggests useful measures of effectiveness of adaptive systems and defines the *adaptive systems design* problem. The framework is then particularized to adaptive strategy selection in query processing. We then review applications of adaptive systems in various disciplines, including computer science and in particular database systems.

2.1 Adaptive systems: definition and framework

An adaptive system is first of all a *system.* The first order of business is then to distinguish the system from its environment, a distinction that depends on the point of view.

The system may not be able to observe all of its environment at once; we speak of a set of *signals* through which the system observes the environment.

Adaptivity implies change: modification of some part of the system. Likely, not all of the system changes, so we must identify which parts of the system undergo changes relevant for adaptation. In computer systems this may include algorithms and data structures.

The purpose of change is to improve performance. The next requirement is then to identify some measure of performance that improves over time as the system adapts to its environment.

Finally, there must be some agent of change, a component internal to the system that observes its performance and modifies its structures accordingly.

This all leads to the following definition:

> An adaptive system is a *system* that is able to *modify part of its structure* in response to *signals* from its *environment*, so as to improve some *measure of performance* over time. The rules for structural modification are called an *adaptive plan.*

Formally, an adaptive system is a quintuple $< \mathcal{E}, \mathcal{A}, \mathcal{I}, \tau, \mu >$, where:

- $\mathcal{E}$ is the set of possible environments. At any particular time the system inhabits one of these environments, $e \in \mathcal{E}$. It may happen that the environment changes over time; if this is the case, the rules of this change have to be specified as well. The changes in the environment may be caused by the adaptive system, or they may be independent of the actions of the adaptive system.

- $\mathcal{A}$ is the set of possible forms that the structures undergoing modification may take. At any time these adapting structures have one particular form, $a \in \mathcal{A}$. Usually the structures retain some memory of past interactions of the system with the environment.

- $\mathcal{I}$ is the set of values of signals that the system may receive from the environment.

- τ is the adaptive plan. It is responsible for generating new structures that are better adapted, based on the signals received. Thus, it can be interpreted as a mapping: $\tau : \mathcal{A} \times \mathcal{I} \rightarrow \mathcal{A}$.

- μ is a set of measures of performance. Little can be said in general about these measures, but we may be interested in the following:

  - For a fixed environment $e \in \mathcal{E}$, the performance of the current structure on the environment, $\mu_e(a)$. As a evolves to make the system better adapted to e, $\mu_e(a)$ changes also. Sometimes this performance measure may approach an asymptotic value.

  - For environments that change over time, we may be interested in average measures, given a specification of the kind of changes, and in the *speed of adaptation* of the system to the changes in its environment. This applies to cases where the structures a converge to a fixed structure over time for each possible environment. When the environment changes, the transient behavior of the a's produced by the adaptive plan and the time of adaptation to the next fixed structure may be of interest.

The conflict between exploiting and exploring

There is an interesting conflict inherent in the operation of adaptive plans [53,44]. At any point in time, the plan τ may know what is the best structure a so far, and use it over and over —we call this alternative "exploitation", to exploit the best known structure. However, new, untried structures may explore unknown parts of the environment, or might simply be better than the structures known so far —but we might not know unless we give them a chance. We might call this "exploration". Clearly, both must go on to some degree. We cite examples of this conflict and possible resolution techniques later.

Adaptive systems design

A particular adaptive system is specified by the quintuple $< \mathcal{E}, \mathcal{A}, I, \tau, \mu >$ outlined in the previous section. An *adaptive system design* problem consists of discovering what is the best system for a particular application. Hence we need *criteria* to enable the comparison of different systems; this set of criteria is denoted as χ.

The measures of performance mentioned above then become functions of parts of the system itself; for instance, there might be adaptive plans that exhibit excellent average performance but poor speed of adaptability to changes in the environment, or the other way around. χ weighs each factor to enable comparison of systems.

2.2 Adaptive strategy selection

In this section, we apply the framework to describe our adaptive strategy selectors. See Holland[53] for other examples of application to very diverse disciplines.

2.2.1 A general adaptive query processor

Figure 2.1 shows a diagram of a general adaptive query processor as a data flow diagram [35], where each bubble represents a process and each arrow is a data flow.

There are two processes: an *execution strategy selector* and a *query executor*. The selector chooses an *execution strategy* s_k for the input query q; using this strategy, the executor obtains the answer to the query from the database.

Let Q be the set of all queries, and $S = \{s_1, s_2, \ldots s_K\}$ be the set of all strategies.

We notice two feedback arrows in the diagram. The lower arrow carries a set of *signals* I_x from the executor to the selector. When the selector *tries* an execution strategy, it receives feedback from the execution of the query with that strategy: this feedback is named I_x. The selector may also obtain other signals directly from the database, independently of the execution of any query. These other signals are depicted in the upper feedback arrow as I_d and may include things like cardinalities

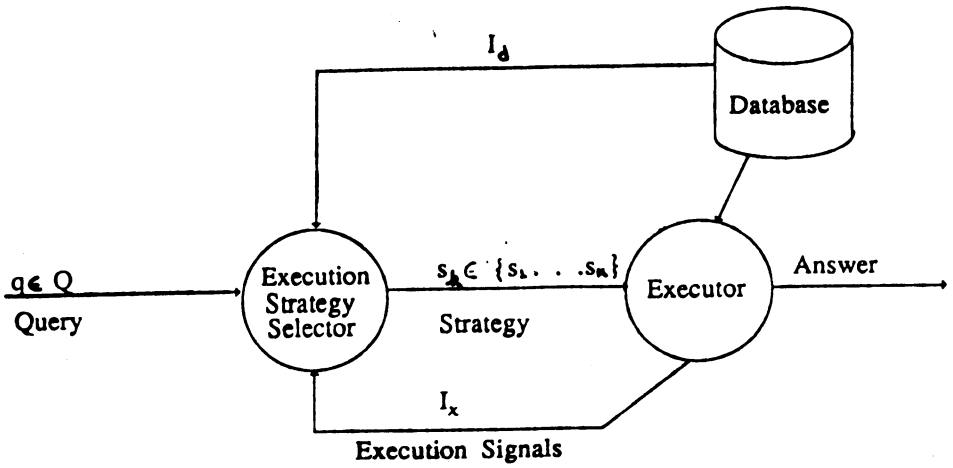


Figure 2.1: Adaptive query processor

of relations, selectivities, etc. The objective of the system is to execute queries as cheaply as possible.

Note that the query processor does not affect the database. Updates are assumed to take place somewhere else, outside of the system; of course, updates will affect the signals I_x and I_d that the selector receives. It is also possible that the query portion of an update, if any, is routed through the adaptive selector.

An even more general version of an adaptive query processor would include adaptive storage structure modification; another arrow would then go from the selector to the database. This possibility will not be further elaborated.

2.2.2 Formal model

Following the framework of section 2.1,

- The *system* is the box labeled "Execution strategy selector".
- The *environment* e of the system is given by the triple <database schema, database state, query load>. The database state is an instance of the data in the database that satisfies the schema plus associated constraints. The query load may be characterized in several ways; for instance, as a probability distribution over Q, the set of all queries.

  The adaptive system will be able to adapt to a certain environment e within a set of possible environments $\mathcal{E}$. It may also be able to adapt to changes from one environment to another within $\mathcal{E}$. The most frequent environment change is probably the query load. The database state also evolves as updates are made to the database.
- The system views its environment through the signals $< I_x, I_d >$.

- The *structures undergoing adaptation* may include a *strategy selection policy* P and an *environment model* M. The model M may contain whatever the system currently knows about the three components of the environment. It may also include part of the history of interactions of the system with its environment. M is obviously limited by the amount and type of signals received; information not available may have to be filled in with assumptions. Let $\mathcal{M}$ be the set of all possible states of M.

  The policy P is what dictates strategy selection. The strategy selected depends on the current model of the environment and the query; P can then be depicted as a mapping, $P : Q \times \mathcal{M} \rightarrow S$. This mapping can be stochastic, that is: $P : Q \times \mathcal{M} \rightarrow \pi_K$, where π_K is a probability distribution over S. The actual strategy tried is then determined by a random trial on this distribution. Let $\mathcal{P}$ be the set of all possible policies.

- The adaptive plan τ is responsible for updating the model M and possibly the policy P in response to the signals I: $\tau : \mathcal{P} \times \mathcal{M} \times I \rightarrow \mathcal{P} \times \mathcal{M}$. This updating is intended to improve some measure of performance.

- The performance measure μ evaluates how well the current policy P is doing with respect to the objective of answering queries cheaply. A typical measure is the average query execution time; we may also be interested in the speed of adaptation of M and P to different kinds of changes in the environment; and others. A more precise definition of these measures depends on the particular kind of adaptive selector considered; thus this definition will be postponed to chapters 3 and 4.

  Figure 2.2 depicts the operation of τ and the query selection.

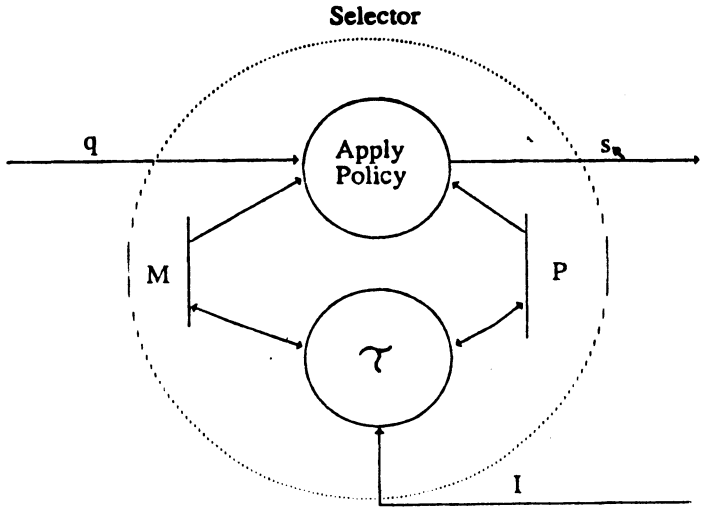


Figure 2.2: τ, in response to I, updates M and P

2.2.3 Traditional and adaptive approaches revisited

Using this formal model, we can visualize the difference between traditional query processors and the adaptive processors to be considered in this work. In a traditional query processor, the lower feedback arrow is usually missing: no measures are usually taken *during* the execution of the query nor *immediately after* the query is executed. The model M determines the expected cost of query execution, based on numerous assumptions about the database and query load. This model is updated with some data measures I_d as the database is updated. The policy P selects the best strategy based on the approximate cost predictions from M; the policy itself is never updated.

In this thesis, we will mostly consider query processors in which the upper feedback arrow is missing and the lower feedback arrow carries a single measure, the *cost of query execution.* This single important measure takes into account all the factors that can affect query execution cost, such as record blocking, selectivities, contention or whatever. It is also sensitive to variations in the query load.

2.2.4 Division of queries into classes.

In general, it is too much to expect that individual queries be optimized adaptively, as there are too many queries, each of which is reposed at infrequent intervals. All the possible queries will be divided into a number of classes, and the adaptive optimization will apply to all the queries within each class.

The process named "execution strategy selector" of figure 2.1 is now subdivided into several subprocesses, as shown in figure 2.3. There is now a classifier module,

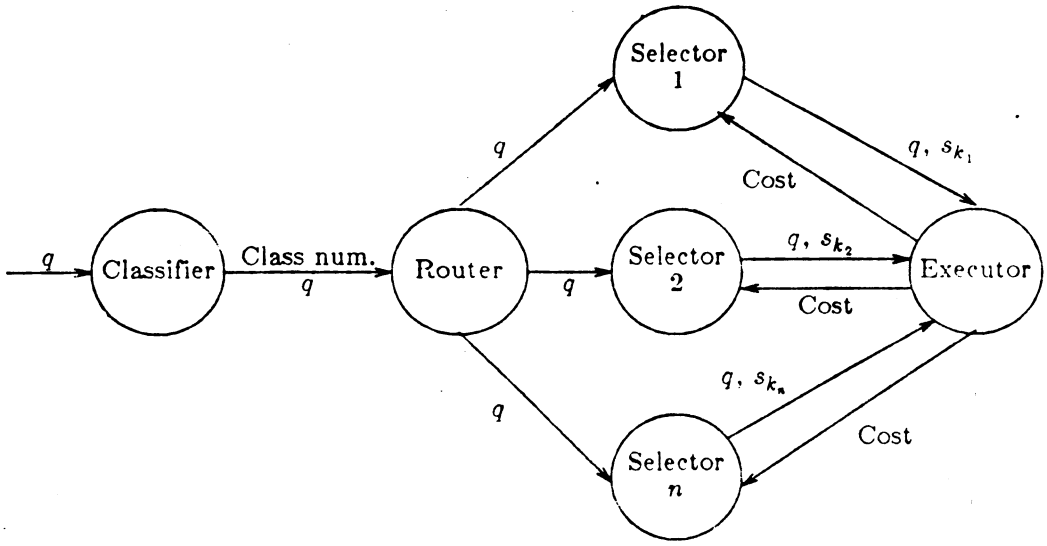


Figure 2.3: Query classification

which determines the query class from the query text, and several selector modules,

one for each class. Each selector carries an environment model M and a decision policy P for the queries in its class. The adaptive plan τ can be different for each selector, although in the systems that we will consider the same plan will be used for each. The kind of strategies selected are also tailored for the class: *all the queries in a class should be executable using any member of a common set of strategies for that class.*

Each class has a performance measure μ_c that could be the average execution time for all queries within the class; and there is an overall performance measure μ for all the queries. As the individual τ's strive to improve the μ_c's, the overall μ improves also.

How the queries are subdivided into classes is of no concern to the individual strategy selectors; all that matters is that the executor be able to execute the query given the query text, class number and strategy number. We consider below two extreme classifications of queries and other more reasonable classifications. Without loss of generality, we illustrate the options in terms of querying a relational database.

- All queries belong to the same class. Consider two strategies: "use sort/merge to carry out any joins", and "use nested loops". There have been various studies that compare these two and other join methods under several assumptions [91]. Instead of making assumptions, the adaptive approach would determine over time the best overall strategy *for the particular environment* (database state and query load) that the system happens to be inhabiting, and would change the strategy selected should changes in the environment make this profitable.

- Query class corresponds to *query type*: all syntactically equivalent queries where attribute values have been replaced by variables. Strategies are tailored for each query type.

- All queries that refer to the same subset of relations belong to the same class. Different strategies may then correspond to different orders of performing joins.

- Classes partition queries by the set of indexed attributes that appear in the selection clause of the query. Thus two queries belong to different classes if they select on different indexed attributes. If a class has n indexed attributes $a_1, a_2, \ldots a_n$, strategy number i for the class can be "use the i_{th} index to retrieve all records that match the condition on a_i. On each of those records, verify the rest of the condition"

We stress that the fundamental query classification criterion is *equivalence of strategies*: we cannot have a class where some subset of queries can be executed with some of the strategies and another subset with other strategies. What is a good classification scheme and what strategies to use depends on the particular problem addressed; the judgement of the database designer and database administrator must be used here. In chapter 6 we will consider an approach whereby queries in a class are further partitioned adaptively into several subclasses, taking into account the execution signals fed back from the database.

2.3 Review of adaptive systems

This section is a brief survey of some applications of adaptive systems to several areas.

First of all, we would like to mention that our definition of adaptive system is by no means universally accepted; in fact, dozens of definitions of the terms *adaptive* and related terms such as *self-organizing, self-improving, auto-modifying*, and so on, are known. Several authors emphasize a feature common to most adaptive approaches, which is *reduction of uncertainities over time.* That is, performance is improved by using information acquired on-line about uncertain parts of the environment. It is possible that whatever we call adaptive according to our definition will not be so called by other people and the other way around.

Adaptive controllers are reviewed comprehensively in the books by Saridis [78] and by Harris and Billings [50]. Most of these approaches consider plants describable by sets of partial differential equations, whose nature may be partially unknown. The controller then infers values for the unknowns over time. One particular class of controller, based on variable-structure automata [60], does not assume any particular form of the plant equations.

In statistics, adaptive sampling techniques have been used under the name *sequential sampling.* The idea here is to use the actual data values that have been sampled in deciding on the fly whether the sampling should continue or can stop. See [81] for a review of this work. Other relevant work in statistics will be reviewed in chapter 3.

Within computer science, John H. Holland and his associates at the University of Michigan have been most active in research related with *genetic algorithms*[53] for the solution of a wide variety of problems, including adaptive control [32], learning classifiers [46] and function optimization [15]. These computer algorithms are based on analogies with biological adaptation. A coded representation of the evolving structures a is chosen. New a's are then generated and modified using genetic laws similar to the ones perfected by Nature. Thus the a's are married with each other, their substructures crossed-over and mutated, and reproduced according to their success in the environment. Surprisingly, this results in a rather efficient exploration of uncertain environments that rapidly converges to the best structures. As an example, Grafenstette et al., [47], use as a an encoding of a traveling salesman tour, such that the genetic operations mentioned above result in valid tours. The performance measure is simply the tour length, and the system eventually evolves short tours. The proceedings of a recent conference describe applications in several areas of Computer Science[1].

In numerical integration, there are algorithms that automatically adapt the integration step size so as to improve accuracy on functions that have diverse and changing slopes [72]. Adaptive algorithms have also been used for routing of messages in networks [59]; in this case, the best routes are assigned based on performance measures of the existing routes. Adaptive data structures such as move-to-front lists [73] have proved very useful.

Most of the *learning systems* considered in artificial intelligence are adaptive according to our definition. The checkers player program by Samuel [77] is an early example. See [37] for a recent survey.

In databases, the most common application of adaptive techniques is the adaptive modification of physical data structures so as to improve performance. An early paper by Stocker and Dearnley [83] described the general approach. Hammer and Chan [49] considers the adaptive creation and destruction of indexes. Yu and Chen [92,95] discuss adaptive clustering methods that put together database records that tend to be accessed as a group. This is put in the form of a general adaptive framework that modifies an internal data structure one query at a time.

In query processing, some attempts have been made to alter the execution strategies dynamically, as the execution proceeds. Clausen [27] uses an adaptive method to process relational joins in which the join selection condition is modified *on the fly* as records are accessed. Rowe and Stonebraker [76] and Yu et al. [93] do not preselect the order of all joins before the query is executed, as implied by our traditional model, but can partially determine the join order at run time as the cardinalities of internal results are generated. In these examples, the adaptation takes place *during* the execution of a particular query, rather than over the execution of many queries. Yu et al. [94] suggests a simple modification of some of the cost formulae, based on the difference between the estimated cost and the last actual cost obtained with each formula. He does not present analytic or simulation results of using this approach, however.

None of the approaches considered so far in adaptive query processing utilizes the actual *cost history* of query execution to guide the adaptation, which is our main contribution.

Chapter 3

Optimal strategy selection policies

In this chapter we analyze the optimal design of *one* of the selectors shown in figure 2.3. Thus, all the queries to be considered belong to one class. At first we assume that the environment of the system remains constant; this restriction will be relaxed in the next chapter. The signal I_x is the cost of execution.

The computational requirements for optimality are too high for practical purposes. Nevertheless, the study of optimal policies is interesting for the following reasons:

- It establishes the need for heuristic, approximate policies with reduced computational requirements.
- It provides a measure to determine the worth of approximate algorithms for strategy selection.
- The behavior of optimal policies lends insight into the design of approximate ones.
- The problem is mathematically interesting in itself.

In the next section, we will formally state the optimal policy design problem. Section 3.2 presents the solution for a very general case and a program that derives the optimal policy. Section 3.3 briefly discusses other less general cases with more reduced computational requirements.

The problem turns out to be related to a wide class of problems that turn up in different guises in several disciplines, including *bandit problems* in statistics[14] and *multi-stage decision problems* in decision theory[71]. These related problems are reviewed in section 3.4.

3.1 An optimal sampling selector

Let $S = \{s_1, s_2 \ldots s_K\}$ be the set of strategies, Q be the set of queries that belong to the class, and Λ be a sequence of queries, the *query load*, $\Lambda =< q^{(1)}, q^{(2)}, \ldots >$

where $q^{(t)} \in Q$ represents the query posed at time t. Λ is unknown in advance to the adaptive system. We recall that the same K strategies are available for any of the queries.

The actual cost of executing a query is a function of the query, the execution strategy, and the database state. Assume for the moment that the database state is fixed. We can imagine a function C, unknown to the strategy selector, that gives the cost of each strategy for each possible query:

$$C : Q \times S \rightarrow \Re$$

Now consider C_k, the restriction of this function to strategy s_k. Under the query load Λ, C_k gives rise to a sequence of costs for strategy s_k, $\rho_k =< x_k^{(1)}, x_k^{(2)}, \ldots >$, where for all t, $x_k^{(t)} = C(q^{(t)}, s_k)$. In this chapter, we will assume that each cost in the sequence is an independent random observation from an unknown probability distribution X_k that does not change with time; this assumption will be relaxed in the next two chapters. Thus X_k is the p.d.f. of the cost of s_k.

A simple example with three queries and two strategies follows:

$$Q = \{q_a, q_b, q_c\};\ K = 2;\ S = \{s_1, s_2\}$$

C :

q	s_1	s_2
q_a	10	20
q_b	20	10
q_c	10	25

$$\begin{aligned} \Lambda &= < q_a, q_a, q_b, q_a, q_c, q_c, q_c, q_b \ldots > \\ \rho_1 &= < 10, 10, 20, 10, 10, 10, 10, 20 \ldots > \\ \rho_2 &= < 20, 20, 10, 20, 25, 25, 25, 10 \ldots > \end{aligned}$$

When the adaptive selector chooses strategy s_k, the actual cost of execution of the query on this strategy is fed back; thus after each query, the selector has available an extra sample cost that comes from the unknown distribution X_k. Figure 3.1 shows the operation of the selector. At each time t, the selector decides on a strategy $s_k^{(t)} \in S$. The query executor can be idealized as a black box that contains K unknown probability distributions $X_1, \ldots X_K$. The executor responds to $s_k^{(t)}$ with a sample observation $x^{(t)} = x_k^{(t)}$ from distribution X_k. The selector maintains an internal model M of these distributions; the model gets enriched after each sample point is received. The selector incorporates a *decision policy* P that specifies which strategy to choose for every state of the model, $P : \mathcal{M} \rightarrow S$, where $\mathcal{M}$ is the set of all model states.

The goal is to drive the selector to choose the cheapest strategies for the query class; or more precisely, *to devise a decision policy that minimizes the expected value of the total query execution cost:*

$$minimize\ \mu(P) = E\left(\sum_{t=1}^{h} x^{(t)}\right)$$

h is a known quantity called the *time horizon.* Note that μ is a function of the policy P.

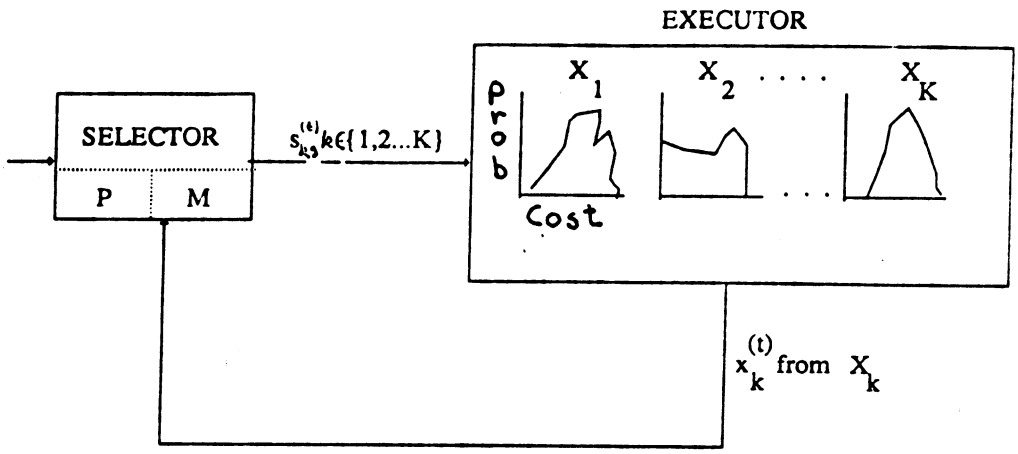


Figure 3.1: Optimal sampling selector

A finite time horizon is reasonable for this problem, because in database systems the constancy of the environment cannot usually be guaranteed for very long; h would then be the expected amount of time before the environment (and therefore the distributions X_k) change. These changes and therefore the proper value of h can be determined adaptively by means of *change point detectors*, as discussed in the next chapter; in this chapter we will consider that h is a given constant quantity.

If the distributions X_k were totally known, the task would be trivial: the optimal policy would be to choose the strategy whose distribution has the least mean. Unfortunately, the distributions are presumed to be unknown.

A simple idea would be to use as model M a set of K averages $\bar{x}_1, \bar{x}_2, \ldots$ where $\bar{x}_k$ is the running average of the values observed from X_k. The policy then may be to sample from the distribution that has the minimum current average. This is called a *myopic* or *greedy* policy, and it is not optimal in general; in many occasions, it is better to observe a strategy that is not the current best, in order to gain information that can be used in the future to make better decisions. This is an example of the familiar conflict between exploitation and exploration mentioned in chapter 2.

We will now pose the problem more formally using a bayesian approach [79,33]. This approach provides an excellent framework for expressing uncertainty and accounting for learning as time passes. The uncertainty about each distribution X_k is represented by a *distribution family* F_k and a *prior distribution* $\pi_k^{(0)}$. The distribution family is a finite or infinite set of either continuous or discrete distributions, but we will only consider discrete distributions. The actual distribution X_k is one of the members of F_k; the uncertainty about *which* member is expressed by $\pi_k^{(0)}$. The prior distribution is a distribution over the family members; it specifies the decision maker's assessment of the initial probabilities that the actual X_k is equal to each family member, *before any sampling has taken place*:

$$\pi_k^{(0)}(f) = Pr(f \equiv X_k) \text{ for all } f \in F_k$$

If the prior distribution is *uniform*, it is called a *flat prior*.

Learning is realized by obtaining a *posterior distribution* as a function of the prior distribution and sampling evidence. The posterior distribution is obtained by means of Bayes' theorem as will be illustrated later. In the posterior distribution, family members that appear more likely after considering the sample observations get assigned higher probabilities.

As an example, consider a finite family of two discrete distributions with a non-flat prior [1]:

$$F_2 = \{3/5\delta_1 + 1/5\delta_2 + 1/5\delta_3, 1/3\delta_1 + 1/3\delta_2 + 1/3\delta_3\}$$
$$\pi_2^{(0)}(3/5\delta_1 + 1/5\delta_2 + 1/5\delta_3) = 2/3,$$
$$\pi_2^{(0)}(1/3\delta_1 + 1/3\delta_2 + 1/3\delta_3) = 1/3$$

The formal statement of the *optimal policy design* problem —actually, a class of problems— is then as follows:

- Given:
  1. K —the number of strategies
  2. $F_1 \ldots F_K$ —K distribution families
  3. $\pi_1^{(0)} \ldots \pi_K^{(0)}$ —K prior distributions over the families.
  4. h —the time horizon
- Find:
  1. A model M that will evolve through states $M^{(0)}$, $M^{(1)}$ ...
  2. The initial state of the model, $M^{(0)}$
  3. τ —A rule for obtaining $M^{(t+1)}$ from $M^{(t)}$ after a cost value has been obtained from the execution of a strategy. Let X be the set of possible costs, and recall that $\mathcal{M}$ is the set of all model states:
  $$\tau : \mathcal{M} \times S \times X \rightarrow \mathcal{M}$$
  4. $P : \mathcal{M} \rightarrow S$ —the decision Policy
- Such that: If $M^{(t)}$ is the state of the model at time t, and $x^{(t)}$ is a value sampled from the distribution corresponding to strategy $P(M^{(t)})$ (the strategy dictated by the policy), then the policy performance measure $\mu(P)$, defined as
  $$\mu(P) = E\left(\sum_{t=0}^{h} x^{(t)}\right),$$
  is minimum among all possible policies,
  $$\mu_P = \min_{p \in P} \mu(p)$$
  The expectation above is taken over realizations of the process *and* over the initial prior distributions.

[1] The notation $p_1\delta_{v_1} + p_2\delta_{v_2} + \ldots$ represents a probability distribution where value v_1 is obtained with probability p_1, value v_2 with probability p_2 and so on.

A few words are in order about the relationship of this formal statement of the problem and the formal framework for adaptive selectors of section 2.2.2. There, we said that the environment e of the system included the schema, database state and query load. Here we have abstracted the database and query load as a set of cost distributions, and thus $e = X_1 \ldots X_K$. The set of environments $\mathcal{E}$ that the system can inhabit is now characterized by the distribution families, $F_1 \ldots F_K$; thus $\mathcal{E} = F_1 \times F_2 \times \ldots F_K$ and a particular environment $e \in \mathcal{E}$ consists of one cost distribution from each family, corresponding to the possible costs of a particular strategy. The adaptive plan τ modifies only the model M; the policy P remains fixed. Note that the performance measure μ_P is an average over all possible environments $\mathcal{E}$; on a *particular* environment $e \in \mathcal{E}$ the system may achieve a cost sum smaller or greater than μ_P. Finally $\mathcal{X}$, the criterion to compare different adaptive systems, is optimality: M, τ and P are to be chosen in such a way that μ_P is minimal.

In the following section, we consider a subclass of problems for optimal policy design wherein all strategies are associated with arbitrary families of discrete cost distributions. Other distributional assumptions will be briefly considered in section 3.3. The solution is obtained by dynamic programming within the bayesian framework. We will also present an algorithm for the solution and analyze its computational requirements. This case and its analysis has not been described before in the literature.

3.2 Finite families of discrete distributions

3.2.1 Problem statement

Here we assume that the cost values are integers, for instance number of disk blocks accessed or number of cents charged to answer the query. We also assume that the lower and upper cost values are bounded and known for each strategy. Let these be l_k and u_k for strategy s_k, and let $N_k = u_k - l_k + 1$. Thus each cost distribution X_k is an unknown probability distribution over the integers between l_k and u_k, and each corresponding distribution family F_k is a known *finite* set of distributions over the same set of integers.

The solution that we will obtain is general, but for clarity we will develop the solution in parallel with an example of a problem instance. The example has $K = 2$ strategies, the cost limits for the first strategy are $l_1 = 1, u_1 = 3$, and for the second strategy are $l_2 = 2, u_2 = 4$. The distribution families can be conveniently represented as matrixes and the prior distributions as column vectors, for instance:

F_1

$\pi_1^{(0)}$	1	2	3
0.5	0.50	0.25	0.25
0.5	1/3	1/3	1/3

F_2

$\pi_2^{(0)}$	2	3	4
1/36	0.8	0.1	0.1
1/36	0.7	0.2	0.1
...	...	...	...
1/36	0.1	0.8	0.1
1/36	0.6	0.2	0.2
1/36	0.5	0.3	0.2
...	...	...	...
1/36	0.1	0.7	0.2
1/36	0.4	0.3	0.3
...	...	...	...
1/36	0.1	0.1	0.8

Here F_1 consists of two distributions; the first assigns probabilities of 0.50, 0.25, 0.25 to the values 1,2,3 respectively, and the second assigns equal probabilities of 1/3 to each value. F_2 consists of 36 distributions; as can be seen, probabilities are multiples of 0.1, in all possible combinations that add up to 1.0. Both prior distributions are flat in this example.

Let $|F_k|$ be the number of strategies in the family F_k. F_k can then be visualized as a matrix of $|F_k|$ rows and N_k columns, where each row represents one of the distributions in the family. Thus $F_k[i, x]$ is the probability of the value x in the ith member of the family. [2] The prior probability $\pi_k^{(0)}$ is a vector of $|F_k|$ elements, where the ith element is equal to the initial prior probability that the ith distribution of the family is the actual cost distribution X_k.

3.2.2 Solution

This completes the specification of the "given..." part of the problem class of section 3.1. We must now find a model M, a way of updating it τ, and the optimal decision policy P. Briefly, M is a set of counters that keep track of the number of times that each value has been observed on each strategy; τ increments the proper counter; and P will be obtained by a dynamic programming approach, where the *stages* correspond to time $t = 1$, $t = 2$, $\ldots t = h$ and the *states* correspond to all possible states of M.

[2] Square brackets will be used to represent subindexes; row index $i \in \{1 \ldots N_k\}$; column index $x \in \{l_k \ldots u_k\}$

M and τ

For the example above, suppose that at time $t = 14$ the first strategy s_1 has been executed five times, and s_2 nine times. On s_1 the cost value 1 has been obtained twice, 2 has been observed once and 3 twice. On s_2 each value has been observed 3 times. Then $M^{(14)}$ would be:

$$\begin{array}{cc} & M^{(14)} \\ M_1^{(14)} & \qquad M_2^{(14)} \\ \boxed{2\ 1\ 2} & \qquad \boxed{3\ 3\ 3} \end{array}$$

Thus M is a set of K vectors, $M = M_1, M_2, \ldots M_K$, where for each k, M_k has N_k elements. The value of M at time t is denoted as $M^{(t)} = M_1^{(t)} \ldots M_K^{(t)}$. $M_k^{(t)}[x]$ counts the number of times that the value x has been observed on strategy s_k, up to time t. The initial state of the model, $M^{(0)}$, is all zeros.

The function $\tau(m, k, x)$ where m is some state of the model, k identifies a strategy s_k, and x is a cost value, $l_k \leq x \leq u_k$, returns a *new* model by incrementing the counter $m_k[x]$ in the given model and leaving all other counters untouched.

Posterior distributions

Bayes' theorem provides a way of combining the prior distribution of a family $\pi_k^{(0)}$ with the sampling evidence $M_k^{(t)}$, in order to derive a posterior distribution $\pi_k^{(t)}$. In the posterior distribution, family members that seem more likely after some sampling will get assigned higher probabilities. For the case that we are considering, the theorem can be stated algebraically as follows:

For all $f \in F_k$:

$$\begin{aligned} \textit{posterior}(f) \;=\; \pi_k^{(t)}(f) \;&=\; Pr(f \equiv X_k \mid \textit{prior}(f), \textit{observed } M_k^{(t)}) \\ &=\; Pr(f \equiv X_k \mid \pi_k^{(0)}(f), M_k^{(t)}) \\ &=\; \frac{\pi_k^{(0)}(f) \times Pr(M_k^{(t)} \mid f \equiv X_k)}{\sum_{f \in F_k} \pi_k^{(0)}(f) \times Pr(M_k^{(t)} \mid f \equiv X_k)} \end{aligned}$$

To derive $Pr(M_k^{(t)} \mid f \equiv X_k)$ = probability of observing $M_k^{(t)}$ *given that* $f \equiv X_k$, consider our example above: let $f = 0.50\delta_1 + 0.25\delta_2 + 0.25\delta_3$ (the first distribution of F_1), and let $M_1^{(14)} = (2, 1, 2)$. The probability that *two* 1's and *one* 2 and *two* 3's have been observed if $X_k = f$, is $\binom{5}{2,1,2} 0.50^2 \times 0.25^1 \times 0.25^2$. Using the matrix and vector notation, this is equal to

$$\prod_{x=l_1}^{u_1} \binom{N_1}{l_1, \ldots u_1} F_1[1, x]^{M_1^{(14)}[x]}$$

Generalizing, substituting in Bayes theorem, and expressing the computations in procedural form leads to the following algorithm to compute the posterior distribution:

1. Compute likelihood $l(i)$ for $i = 1 \ldots |F_k|$:

$$l[i] = \pi_k^{(0)}[i] \prod_{x=l_k; M_k^{(t)}[x] \neq 0}^{u_k} F_k[i,x]^{M_k^{(t)}[x]} \tag{3.1}$$

2. Compute posterior distribution:

$$\pi_k^{(t)}[i] = \frac{l[i]}{\sum_{m=1}^{|F_k|} l[m]}, \; for \; i = 1 \ldots |F_k| \tag{3.2}$$

The posterior distribution allows us to determine what is the expected probability of obtaining any value x between l_k and u_k when executing strategy s_k:

$$Pr(x \mid using \; s_k \; at \; time \; t) = Pr(x|\pi_k^{(t)}) = \sum_{i=1}^{|F_k|} \pi_k^{(t)}[i] \times F_k[i,x] \tag{3.3}$$

The optimal policy by dynamic programming

Now define $V_*^{(t)}(M^{(t)})$, the value of the optimal decision policy at time t, as the expected value of the sum of all future observations when the state of the model is $M^{(t)}$ and optimal decisions are taken at every stage. We would like to find

$$\mu_P = \sum_{t=1}^{h} x^{(t)} = V_*^{(0)}(M^{(0)}),$$

and to identify all the optimal decisions. Note that, for $t > 0$, $V_*^{(t)}$ is a function of the state $M^{(t)}$ (that is, a function of what x-values have been observed so far). $V_*^{(0)}$ however is unique, since at time 0 there is just one state value: all zeros.

Consider the last decision at time h denoted as $s^{(h)}$, $s^{(h)} \in S$. Let $M^{(h)}$ be *one of the possible states* at time h. We can depict each possible decision and each consequence of each decision as a graph, as shown in figure 3.2. Suppose strategy s_k is selected. We can use equation 3.3 to compute the probability that the outcome is l_k, $l_k + 1$ and so on. The expected value of this last decision is then:

$$E(deciding \; s_k \; at \; time \; h \; when \; state \; is \; M^{(h)}) = V_{(s^{(h)}=s_k)}(M^{(h)}) = \sum_{x=l_k}^{u_k} x \times Pr(x|\pi_k^{(h)}) \tag{3.4}$$

and the value of the best decision is:

$$V_*^{(h)}(M^{(h)}) = \min_{s \in S} V_{(s^{(h)}=s)}(M^{(h)}) \tag{3.5}$$

The optimal decision at time h is the value of s that minimizes the last equation.

Now consider the previous stage, at $t = h-1$. Assume that the values of optimal decisions *at all possible states* on the last stage, $V_*^{(h)}(M)$, are known. Let $M^{(h-1)}$ be one of the possible states at time $h-1$, and recall the definition of the function τ

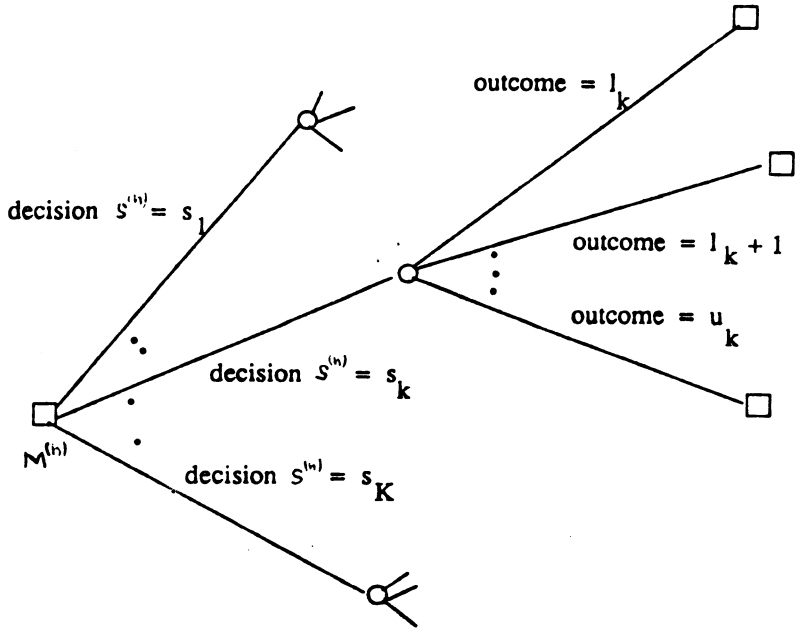


Figure 3.2: Decision tree at last stage

(page 21). Let us now compute the expected value of executing s_k at time $h-1$; this value now depends not only on the expected probabilities from equation 3.3, but also depends on the values of future optimal decisions at time h already computed:

$$V_{(s^{(h-1)}=s_k)}(M^{(h-1)}) = \sum_{x=l_k}^{u_k} Pr(x|\pi_k^{(h-1)}) \times \left[x + V_*^{(h)}(\tau(M^{(h-1)}, k, x))\right] \tag{3.6}$$

and the value of the optimal policy at time $h-1$ is

$$V_*^{(h-1)}(M^{(h-1)}) = \min_{s \in S} V_{(s^{(h-1)}=s)}(M^{(h-1)}) \tag{3.7}$$

Finally consider an arbitrary stage t. Assume that the values of optimal policies at all possible states on stage $t+1, V_*^{(t+1)}(M)$, are known. The formulas are now:

$$V_{(s^{(t)}=s_k)}(M^{(t)}) = \sum_{x=l_k}^{u_k} Pr(x \mid \pi_k^{(t)}) \times \left[x + V_*^{(t+1)}(\tau(M^{(t)}, k, x))\right] \tag{3.8}$$

and the value of the optimal policy at time t is

$$V_*^{(t)}(M^{(t)}) = \min_{s \in S} V_{(s^{(t)}=s)}(M^{(t)}) \tag{3.9}$$

The preceding equations can be solved backwards, obtaining first $V_*^{(h)}(M)$ from equations 3.5 and 3.4, then $V_*^{(h-1)}(M), V_*^{(h-2)}(M) \ldots$ and so on from equations 3.9 and 3.8, and so on until obtaining the value of the optimal decision at time 0, $V^{(0)}(M^{(0)})$. For each state, the decision that leads to the minimum expected value is remembered.

3.2.3 Algorithm and analysis

We now proceed to analyze the computational requirements of the dynamic programming approach. Pseudocode of an iterative algorithm could be as follows:

```
for t = h ... 1
    for each distinct state M^(t) possible at time t
        for each of the K possible decisions at state M^(t)
            compute π_k^(t) with equations 3.1, 3.2
            for each of the N_k possible outcomes
                compute Pr of each outcome with equation 3.3
            compute expected value of decision with equations 3.4 or 3.8
        compute value of best decision with 3.5 or 3.9
        store the best decision and its value.
```

We will first compute S_t, the number of states of M possible at time t. Recall that M is a set of counters; let N be the number of counters, $N = \sum_{k=1}^{K} N_k$. At time t, t queries have been answered, and each query answer increments one counter (corresponding to the cost observed for a single strategy), therefore all these counters must add up to t. Thus the number of possible states of M_t is the number of different ways that N counters can add up to t. This can be obtained via a related formula for the number of *ordered partitions* of an integer t into i parts, which is $\binom{t-1}{i-1}$ [69, pp. 67]. For example there are 3 partitions of the number 4 into 2 parts: 1 3, 2 2, and 3 1. Each of these can be assigned to the sequence of N counters in $\binom{N}{i}$ different ways; thus we get $\binom{N}{i}\binom{t-1}{i-1}$ ways of having i non-zero counters and $N - i$ zero counters add up to t. Then

$$S_t = \sum_{i=1}^{N} \binom{N}{i} \binom{t-1}{i-1}$$

This can be simplified. First the second combination is changed to $\binom{(t-1)}{(t-1)-(i-1)}$. This leaves the resulting summation formula in a known form [56, sec 1.2.6 eq. 21] that simplifies to:

$$S_t = \binom{N+t-1}{t}$$

The total number of states from $t = 0$ until $t = h - 1$ is then

$$\sum_{t=0}^{h-1} \binom{N+t-1}{t} = \binom{N+h-1}{h-1} \text{[56, 1.2.6 eq. 10]}$$

We will now compute the number of operations required *per state*. Computing the posterior distribution with equations 3.1 and 3.2 for a given state $M^{(t)}$ and decision s_k takes $O(N_k|F_k|)$ operations. $O(N_k|F_k|)$ operations are needed for the computation of the probability of all the outcomes plus the expected values with equations 3.3, 3.4 and 3.5. Assuming that all the distribution families are of size bounded by $|F|$, the number of operations per state is $O(N|F|)$.

Therefore the algorithm takes

$$\binom{N+h-1}{h-1} \times O(N|F|)$$

For N large and $h >> N$, this is $O[(h/N)^N N|F|]$.

This is an exhaustive enumeration of all the states of M, all the decisions at each state and all the outcomes of each decision. However, this is still faster than an exhaustive enumeration of all the *policies*, for this will also require computing the posterior distribution at all possible states *plus* h additional operations to evaluate each policy. The number of possible policies can be as big as the number of mappings of the form $M \to S$; therefore an exhaustive enumeration of policies would be of the same order as the dynamic programming approach, but with a larger constant associated with the order term.

3.2.4 Computer program and sample run

A recursive version of the preceding algorithm was implemented in the language Maple [21,22]; the code is in appendix A. Input for this program includes the number of strategies K, time horizon h, minimum and maximum cost values per strategy, the distribution families $F_1 \dots F_k$ and the prior distributions $\pi_1^{(0)} \dots \pi_K^{(0)}$ (by default, these are assumed *flat*). The program prints the expected value of the optimal policy $V_*^{(0)}$, and a table that indicates the optimal decision(s) for any state that can be reached when following optimal policies. There can be more than one optimal policy; in this case, there will be some state(s) in which more than one decision is optimal —all of them are then listed.

The computational requirements are very high, as shown by the preceding analysis; therefore, we only present results for small values of the time horizon and the total number of possible cost values N. The results are however sufficient for observation of the behavior of optimal policies in general.

Input./ $K = 2$ strategies, $h = 8$. Each distribution family consists of two distributions, as follows:

	F_1			
$\pi_1^{(0)}$	1	2	3	Mean
0.5	0.1	0.1	0.8	2.7
0.5	0.1	0.5	0.4	2.3
				2.5

	F_2			
$\pi_2^{(0)}$	2	3	4	Mean
0.5	0.8	0.1	0.1	2.3
0.5	0.4	0.3	0.3	2.9
				2.6

The values were chosen in such a way that, considering the four possible ways of taking one distribution from each family, in two cases the first strategy is better than the second one, in one case the second is better than the first, and in the remaining case the two are equally good. Both prior distributions are flat, so each of the four cases is equally likely.

Since the first strategy looks on average better than the second one, a *simple approach* would choose the first always, achieving an expected cost sum of $2.5 \times 8 = 20.0$

A *random policy* that chooses strategies at random, would get $2.55 \times 8 = 20.40$.

An *impossible approach* that would start by knowing at time 0 which of the four cases apply, would obtain 2.3×8 in three of the four cases, and 2.7×8 on the fourth, giving an expected cost sum of: 19.20.

Figure 3.3 shows the beginning of the raw output of the Maple program. We

```
                          input
                          K := 2
                          h := 11
                       xmin := [1, 2]
                       xmax := [3, 4]
                        N := [3, 3]
          F[1] := [[.1, .1, .8], [.1, .5, .4]]
          F[2] := [[.8, .1, .1], [.4, .3, .3]]

              S0 := [[0, 0, 0], [0, 0, 0]]

                          output
        expected value of any optimal strategy:, 19.77563520

            optimal choices following optimal policies:

1   [[0, 0, 0], [0, 0, 0]]   1
2   [[0, 0, 1], [0, 0, 0]]   2   *
2   [[0, 1, 0], [0, 0, 0]]   1
2   [[1, 0, 0], [0, 0, 0]]   1
3   [[0, 0, 1], [0, 0, 1]]   1
3   [[0, 0, 1], [0, 1, 0]]   1
3   [[0, 0, 1], [1, 0, 0]]   2
3   [[0, 1, 1], [0, 0, 0]]   1
3   [[0, 2, 0], [0, 0, 0]]   1
3   [[1, 1, 0], [0, 0, 0]]   1
3   [[1, 0, 1], [0, 0, 0]]   2   *
3   [[1, 1, 0], [0, 0, 0]]   1
3   [[2, 0, 0], [0, 0, 0]]   1
4   [[0, 0, 2], [0, 0, 1]]   1
4   [[0, 1, 1], [0, 0, 1]]   1
```

Figure 3.3: Value of optimal policy and first part of optimal policy table

can see that the value of the *optimal policy* obtained by dynamic programming

is 19.77563520. There is only one optimal policy in this example. Table 3.1 was produced from part of the program's output; it gives the optimal decision for each state that can be reached while following the policy, from $t = 1$ to the first states at $t = 5$. The table shows the current time step, the state vector of counters, and the number of the strategy to select.

	State						
	s1			s2			deci-
t	1	2	3	2	3	4	sion
1	0	0	0	0	0	0	1
2	0	0	1	0	0	0	2 *
2	0	1	0	0	0	0	1
2	1	0	0	0	0	0	1
3	0	0	1	0	0	1	1
3	0	0	1	0	1	0	1
3	0	0	1	1	0	0	2
3	0	1	1	0	0	0	1
3	0	2	0	0	0	0	1
3	1	1	0	0	0	0	1
3	1	0	1	0	0	0	2 *
3	1	1	0	0	0	0	1
3	2	0	0	0	0	0	1
4	0	0	2	0	0	1	1
4	0	1	1	0	0	1	1
4	1	0	1	0	0	1	1
4	0	0	2	0	1	0	1
4	0	1	1	0	1	0	1
4	1	0	1	0	1	0	1
4	0	0	1	1	0	1	1
4	0	0	1	1	1	0	1
4	0	0	1	2	0	0	2
4	0	1	2	0	0	0	1
4	0	2	1	0	0	0	1
4	1	1	1	0	0	0	1
4	0	2	1	0	0	0	1
4	0	3	0	0	0	0	1
4	1	2	0	0	0	0	1
4	1	1	1	0	0	0	1
4	1	2	0	0	0	0	1
4	2	1	0	0	0	0	1
4	1	0	1	0	0	1	1
4	1	0	1	0	1	0	1
4	1	0	1	1	0	0	2
4	1	1	1	0	0	0	1
4	1	2	0	0	0	0	1
4	2	1	0	0	0	0	1
4	2	0	1	0	0	0	2 *
4	2	1	0	0	0	0	1
4	3	0	0	0	0	0	1
5	0	0	3	0	0	1	1
5	0	1	2	0	0	1	1
5	1	0	2	0	0	1	1
5	0	1	2	0	0	1	1
5	0	2	1	0	0	1	1
5	1	1	1	0	0	1	1
5	1	0	2	0	0	1	1
5	1	1	1	0	0	1	1
5	2	0	1	0	0	1	1
5	0	0	3	0	1	0	1
5	0	1	2	0	1	0	1
5	1	0	2	0	1	0	1
5	0	1	2	0	1	0	1
5	0	2	1	0	1	0	1
	— etc —						

Table 3.1: Optimal decisions table

We notice that the optimal decision at $t = 1$ is to execute strategy 1. Then, if a 3 is obtained, the second row indicates executing strategy 2; otherwise sampling $s1$ again is optimal. At $t = 3$ the table indicates that $s1$ should be chosen in all but two of the 9 possible states that can be reached, and so on.

An asterisk is placed besides the states in which the decision is not *myopic*: A

myopic decision would only look at the current expected values for each strategy, $\sum_{x=l_k}^{u_k} x \times Pr(x \mid \pi_k^{(t)})$, and choose the strategy with the least expected value. Such a decision does not weigh the possible futures resulting from it, as the optimal decision does.

Table 3.1 was then input to a C program that uses the table to take optimal decisions in several experiments. The program receives as another input two numbers that indicate which of the two distributions is to be used from each family. The program initializes $M^{(0)}$ to zeros, and then repeatedly selects a strategy using the table, draws a random sample from its distribution, updates the run-time model M, and repeats the cycle h times.

Let $x^{(t)}$ be the cost obtained at time t. The *expected reward* at time t, $R^{(t)}$, is defined as the value of $x^{(t)}$ averaged over several experiments. Below we list the value of $R^{(t)}$, averaged over 10000 experiments, for each of the four possible choices of distributions from each family. Time runs horizontally from 1 to 8.

s1: 0.1 0.1 0.8; s2: 0.8 0.1 0.1
R(t): 2.700 2.378 2.411 2.463 2.431 2.425 2.442 2.438

s1: 0.1 0.1 0.8; s2: 0.4 0.3 0.3
R(t): 2.700 2.861 2.781 2.735 2.762 2.746 2.726 2.729

s1: 0.1 0.5 0.4; s2: 0.8 0.1 0.1
R(t): 2.300 2.300 2.300 2.300 2.300 2.300 2.300 2.300

s1: 0.1 0.5 0.4; s2: 0.4 0.3 0.3
R(t): 2.300 2.541 2.419 2.348 2.362 2.346 2.327 2.325

Consider the first case. In this case strategy 1 has a mean cost of 2.7, and strategy 2, of 2.3. The cost starts at 2.7 as expected, since the optimal policy indicates strategy 1 as the choice at $t = 1$. A 3 is obtained with probability 0.8; since the optimal decision is $s2$ when 3 is obtained at $t = 1$, at $t = 2$ we can expect to sample $s2$ 80% of the time. Thus the cost goes down to 2.378, then up again and so on. We notice that the optimal policy never settles down to sample the real optimal strategy, $s2$, exclusively; this is because the optimal policy is all the time an average over all possible futures, and even at the last decision there is still uncertainity: the last part of the optimal decisions table (not shown) indicates that in 75 states out of the possible 3^7 states that can be reached at t=8, the optimal decision is s_1. Similar comments can be made about the other three cases.

This is the average of the four cases at each value of time:

2.500 2.5205 2.47809 2.46153 2.464 2.45427 2.44897 2.44776

These numbers add up to 19.77482, in close agreement with the expected value of the optimal policy computed by the Maple program. Here we can see how the optimal policy averages over all four possible environments; on any of the four particular

environments the expected cost sum might be smaller than the expected value for the *optimal policy*. The optimal policy is the overall best when the likelihood of each particular environment obeys the prior distributions.

This example will be used again in the next chapter to compare optimal with approximate policies.

3.3 Other distributional assumptions

Although we illustrated the dynamic programming approach for a particular subclass of problems, using finite families of discrete distributions, the approach can be employed with other distributional assumptions with minor modifications. As an example, we could consider binomial distributions. The cost values considered are still integers in a known range $l_k \ldots u_k$ for each strategy s_k. Each distribution X_k is assumed to be $Binomial(N_k - 1, p_k)$, and the corresponding family F_k is an infinite family of binomial distributions. Different members of the family are obtained by varying the parameter p_k in the interval [0,1]. The prior distribution is then a distribution over p_k; a convenient choice (see for instance DeGroot [34]) is a $Beta(\alpha_k^{(0)}, \beta_k^{(0)})$ distribution. A *flat prior* in this case would be a uniform distribution over p_k, which is obtained by setting $\alpha_k^{(0)} = 0$ and $\beta_k^{(0)} = 0$ on the *Beta*. After a single value x is obtained from strategy s_k, Bayes' theorem obtains a posterior distribution

$$Beta(\alpha_k^{(0)} + x - l_k, \beta_k^{(0)} + N_k - 1 - (x - l_k))$$

The state M no longer needs to keep track of counters for each of the values $l_k, l_k + 1 \ldots u_k$; the sum of the values obtained so far and the number of observations is sufficient state information. After obtaining a sum $M_k^{(t)}$ in n_k observations, the posterior distribution is

$$Beta(\alpha_k^{(0)} + M_k^{(t)} - n_k \times l_k, \beta_k^{(0)} - M_k^{(t)} + n_k(N_k - 1 + l_k)).$$

Note that the number of different state values that can occur is much smaller now: For strategy s_k at time t, it is the number of different sums that can be obtained by adding up t times any of the integers $l_k \ldots u_k$, that is $t \times N_k$; for all strategies, the number of possible states at time t is just $t \times N$. The computational requirements of a program that derives the optimal strategy based on the binomial distributional assumption would then be more reasonable; nevertheless, the exponential nature of the dynamic programming approach eventually limits the time horizon values that can be practically considered.

It is difficult to justify any simple distributional assumption about costs received from arbitrary queries on assorted database systems. Multi-parameter distributions can be considered —computational complexity then creeps in again. As often happens in any modeling effort, there are complex trade-offs between realism, mathematical tractability and computational complexity.

3.4 Related work

Bandit problems

The optimal policy design problem is very similar to some versions of the *multi-armed bandit problem* in statistics [14]. The name originated from the *bandit machines* of gambling casinos; these machines either take your coin returning nothing or return a given amount of money with fixed but unknown probability. How should h coins be allocated to K machines so as to maximize expected gain? Allocation decisions can be influenced by results obtained along the way. Numerous variations of this problem have been described in the literature; they all involve Bernoulli distributions, with a few exceptions dedicated to Normal distributions. Berry and Firstedt [14] have written a recent book dedicated to the study of bandit problems. The book includes a comprehensive annotated bibliography that proved invaluable in this research.

Thompson [86] posed the first bandit problem, the gambling version for two machines. He proposed a heuristic in which an arm is sampled with a frequency proportional to the posterior probability that the arm is better than the other one. There are numerous variations of the bandit setting. Arms can be independent, or dependent [40]; they can be all unknown, or some of them known [9,16]; time can be discrete, or continuous [23]; and so on.

A dynamic programming solution for bandits originated with Bellman [12,11], who set up the optimality equations for the case of two arms, one arm known, and discounted costs. Horowitz [54] shows the dynamic programming approach for the more general case of two unknown Bernoulli arms. Berry and Firstedt [14] in their book explain the general approach and justify the equations on the basis of measure theory.

One version of the bandit problem has attracted considerable attention. In this version, the time horizon is infinite, and future costs are assumed to be discounted. The objective is then to find a policy that minimizes the discounted total cost; see for instance Gittins and Jones [45]:

$$\mu_P = E\left(\sum_{t=1}^{\infty} \beta^t x^{(t)}\right),$$

where β $(0 < \beta < 1)$ is the *discount factor.*

Several authors have proposed heuristic sampling strategies. Most of the strategies apply only to the Bernoulli setting; for instance, Robins [74] first proposed a "play the winner, switch from a loser" strategy that was later widely studied. Bather [8] suggested a variation of the myopic strategy, in which the running averages are corrupted with noise factors that decrease exponentially with time. This provides for explore/exploit balance. Colton [28] was the first researcher to suggest a two stage approach for two normal distributions, in which both arms are equally observed in the first stage, and the best of them exclusively observed in the second stage. He proved that the optimal length of the first stage should be about the square root of the length of both. This spawned a great deal of related research.

A different way to handle bandits was illustrated by Vogel [89] who used a *minimax* approach. The objective here is to design policies that minimize the maximum (worst) possible value that can be obtained.

As we mentioned before, most of the bandits literature deals with the Bernoulli setting; the only exceptions that we found treat Normal distributions as exemplified in the paper by Day [30] which considers two normal distribution families with known variance and a normal prior over the mean, and the multi-stage approach advocated by Colton [28] and followers. Since the papers come from researchers in statistics, computational analysis and complexity is rarely addressed.

Related statistical problems

Various other areas of statistics also develop plans for sampling from several distributions. *Medical trial problems*, for instance, deal with choosing between alternative medical treatments based on applications of the treatments to sample patients. Most of this work deals with *Bernoulli distributions*, that is, a treatment is considered "successful" or "not successful", and the usual objective is to select the best treatment with a given low probability of committing a mistake in the selection. Medical trials conducted in practice allocate a fixed number of patients to each treatment, but several authors have proposed *sequential* procedures, in which the result of the samplings so far influence future allocations. This usually involves *stopping rules* that specify when to stop sampling all or some of the distributions. See Hoel, Sobel and Weiss [52] for a survey and Armitage [3] for a discussion.

Ranking and selection theory is concerned again with selecting the best distribution and sometimes with ranking distributions in order of goodness. This work usually considers normal populations under a variety of assumptions. *Multistage selection* divides the sampling job into a number of *stages*; a sampling rule is specified for each (usually, equal allocation of observations to distributions). At the end of each stage, a decision is taken whether to continue with further stages or to stop sampling and select the best so far. In the former case the distributions to be considered for the next stage may be a subset of the original ones. The references in Finney [41] summarize this work up to 1985.

These three areas of statistics are closely interrelated and papers are difficult to assign to a certain category. As an example, Colton [28] in a medical trial paper, addresses the problem of maximizing the number of successful applications of treatments (a bandit concern) using a variation of multi-stage selection. We may note however, that a sampling plan that selects the best distribution with a given confidence does not necessarily maximize the sum of the observations; these are different problems.

Decision theory

The area known as *multi-stage decision problems* is also related with bandits and the dynamic programming solution. These problems also involve decision trees where each decision can have several outcomes and lead to other decisions. The

end states are assigned a *utility* and the problem is to design a decision *policy* that maximizes expected utility. The policy must specify what is the best decision for every possible state that the tree might reach when the policy is actually applied. These problems do not usually involve cost sums, and the assumptions involve only probability estimates of the various outcomes. See the book by French [42] for an introductory presentation and the one by Raiffa [71] for a more mathematical one.

3.5 Summary

In this chapter we have mapped the optimal policy design problem for strategy selection to other similar problems in statistics and decision theory. Our contributions are first, a clear formal statement of the problem viewed in the framework for adaptive systems introduced in the previous chapter; second, a solution for a very general distributional assumption that has not been considered before; and third, a general computer program that obtains the optimal policy and an analysis of its computational requirements.

Chapter 4

Approximate strategy selection

As we saw in the preceding chapter, the computational requirements needed to design optimal strategy selection policies are excessive for practical purposes. Approximate algorithms that evolve *good*, sub-optimal policies over time then become a practical necessity. We also saw that the optimal design problem requires the specification of distribution families and prior distributions; these may be difficult to obtain for database systems. Further, the algorithms in chapter 3 are limited to fixed environments.

This chapter presents approximate algorithms for fixed or changing environments. As in chapter 3, we are again concerned with a single selector receiving queries from a single query class. Our algorithms are based on learning automata. These automata are reviewed in the next two sections, using the framework for adaptive strategy selectors developed in chapter 2.

We next review various learning algorithms that have been proposed for learning automata; we choose a recent algorithm developed by Thathachar and Sastry [85] as a basis for our procedures. In order to apply this algorithm to the fixed environment database case, we need to understand its behavior in several situations. This behavior cannot be inferred from the analytical results and simulations presented in Thathachar and Sastry's paper; we therefore present further simulations to clarify these aspects. We also present an original comparison of a strategy selector based on learning automata with a selector based on the optimal policies of chapter 3. Our simulations consider strategy costs derived from simple uniform distributions, and discrete distributions over a small range of values. Simulations with realistic data-base costs will be presented in chapter 5. In section 4.4 we present an original extension to the learning algorithm to handle changing environments. Finally in the last sections we briefly consider alternative learning automata algorithms and other approximate algorithms based on principles other than automata.

4.1 Learning automata

Learning automata belong to the class of stochastic automata, in which transitions between states are probabilistic. When the transition probabilities can vary over

time, the automata are called *variable structure stochastic automata*, or *learning automata* for short.

These automata were first introduced by Varshavskii and Vorontosova [87] to model the learning behavior of biological organisms. In this context, an automaton models an organism, repeatedly choosing one out of several *actions* in a stochastic environment. The environment provides a binary *reward* (0) or *penalty* (1) response which is fed back to the automaton. The probability of obtaining a reward is initially unknown and may be different for each action. The automaton must learn over time what are the best actions (i.e. the ones that produce many rewards and few penalties).

In the most common formulation, the automaton has a state corresponding to each action, and the transitions between states are governed by a vector of probabilities; the i'th element of the vector is the probability that the automaton makes a transition from the current state to state number i (or alternatively, the probability that the next action chosen is action i). Probabilities of actions that turn out to be "good" are reinforced (incremented), whereas probabilities of "bad" actions may be penalized (decremented). The objective is to drive the automaton towards choosing the best of the actions exclusively.

This simple model was later taken up by cyberneticians and control theorists, among others, who proceeded to construct systems based on several variations of the basic model and to apply it to several areas. Narendra and his associates [61,62,63] have written three comprehensive survey papers which cover the main theoretical developments and applications. Briefly, the main variations include various *learning schemes* to update the probability vector [18,20,68], the consideration of several kinds of more general environments [4,64], and several ways of interconnecting automata [7,57]. Some of this work will be reviewed in other parts of this chapter.

4.2 Formal model

In this section we will specify the components of an adaptive strategy selector based on a learning automaton, using the framework of chapter 2. We will specify the environment model M, the policy P, the environment e, the adaptive plan τ and several performance measures. $S = \{s_1, s_2, \ldots s_K\}$ is the set of available strategies, which correspond to the actions of the automaton.

Model, policy and environment

The environment model M includes as principal component a probability vector p where p_k indicates the probability of executing strategy $s_k, k = 1 \ldots K$. p varies over time, $p = p^{(t)}$; the initial value of p is set to $p_k^{(0)} = 1/K$ for all k whenever there is no a-priori information about the goodness of some strategies over others. M may also include other components according to the specific kind of automaton considered.

The policy P chooses a strategy by drawing a random observation from the discrete probability distribution denoted by $p_1\delta_1 + p_2\delta_2 + \ldots p_k\delta_k$. If the outcome of the observation is k, $1 \leq k \leq K$, strategy s_k is chosen.

After the automaton selects a strategy, the query is executed using this strategy, and the database management system then feeds back a cost of query execution. For convenience, this cost is normalized by the range of cost values observed so far, obtaining a cost $x \in [0,1]$ (note that this is a generalization of the simple reward-penalty models of early automata). This response is stochastic: the environment e is characterized by a set of K unknown distributions $X_1 \ldots X_K$, where X_k is the conditional distribution of x given that the strategy chosen was s_k. Let μ_k be the mean of X_k, and let s_l be the strategy that corresponds to the minimal mean distribution: the *optimal strategy*. For notational simplicity we will assume that there is only one optimal strategy; the algorithms remain unchanged if there are several. For now we assume that the distributions do not change over time; this restriction will be lifted in section 4.4

The adaptive plan

The adaptive plan τ is in charge of updating the probability vector and other components of M according to the responses of the environment in such a way that performance improves. Thus $\tau : M \times S \times [0,1] \rightarrow M$.

As an example, consider the *linear reinforcement inaction* (L_{R-I}) scheme, originally proposed by Bush and Mosteller [18] for simple reward–penalty environments ($x \in \{0,1\}$). In this scheme, the only component of M is the probability vector p. The probabilities are updated as follows, where s_k is the strategy executed last and x is the corresponding response from the environment:

$$\tau_{L_{R-I}}(M,k,x) \; where \; M =< p >$$

$$\begin{aligned} p_j^{(t)} &= p_j^{(t-1)} - (1-x)p_j^{(t-1)}\lambda_R \quad for\ all\ j \neq k, \\ p_k^{(t)} &= p_k^{(t-1)} + \sum_{j \neq k}(1-x)p_j^{(t-1)}\lambda_R \end{aligned}$$

λ_R $(0 \leq \lambda_R \leq 1)$ is a *learning parameter*

Note that under a penalty response (x=1), the probabilities are left unchanged; under a reward response (x=0), the probability of the selected strategy is incremented, and all others linearly decremented in such a way that all the probabilities still add up to 1. The *learning parameter* λ_R is fixed beforehand.

Figure 4.1 shows the pseudocode of strategy selection based on a learning automaton.

Performance criteria

The performance criteria are related to the asymptotic behavior of the probability vector p. Given a particular updating scheme, will the automaton eventually converge to the optimal strategy? (That is, will $p_l^{(t)} \rightarrow 1$ as $t \rightarrow \infty$?) How quickly?

initialize(M) (including probability vector p)
repeat forever:
  Get next query q
  $k := P(M)$ —apply policy P (probabilities) to choose a strategy s_k
  $x := execute(q, k)$—execute query with strategy s_k. Get feedback cost x
  $M := \tau(M, k, x)$ —update state of the automaton

Figure 4.1: Strategy selection with a learning automaton

How reliably? For the L_{R-I} scheme, Norman [67] proved that the automaton always converges to a single strategy, not necessarily the optimal one; however, if λ_R is small enough, it will always converge to the optimal strategy, at the expense of an increased convergence time. Viswanathan and Narendra [88] then showed that the L_{R-I} scheme also works well for the case where $x \in [0, 1]$, and that the convergence result still holds. There are no reported analytical results regarding convergence time.

We will use several performance measures in order to evaluate adaptive plans and parameter values. Typically, an experiment is repeated several times with a particular set of distributions $X_1 \ldots X_K$; the following is then computed:

- The expected number of steps until convergence of p_l to a given value. This is computed as the average over all experiments of the number of steps until convergence. The range and standard deviation of the number of steps are also of interest.

- The *accuracy*, an estimate of the probability of convergence to the optimal strategy. This is computed as the quotient of the number of times that the automaton converged to the optimal strategy, over the number of experiments.

- The value of the *average expected reward* over time. The *expected reward* at time t is defined as $R^{(t)} = \sum_k p_k^{(t)} \times \mu_k$; clearly the minimum possible value of the expected reward is c_l, and is achieved at the same time that the automaton converges to the optimal strategy. The average expected reward at time t is the average of $R^{(t)}$ over all the experiments. In general, the evolution of $R^{(t)}$ may be more representative of an automaton's worth than the evolution of the probabilities. To see this, consider the following 3-strategy example, where the X_k are uniform distributions with a small range (say, 0.1), and we have: $c_1 = 0.2, c_2 = 0.201, c_3 = 0.8$. A typical learning algorithm will quickly eliminate s_3 from consideration (that is, $p_3^{(t)}$ will reach a value of almost 0 for a small value of t), but it may take a very long time to decide the better between s_1 and s_2, because their means are so close. Convergence time to the optimal strategy may then be very long. However, the value of $R^{(t)}$ will very quickly reach a value close to its minimum 0.2, since c_1 and c_2 are widely separated from c_3.

4.3 Learning algorithms for fixed environments

Several learning algorithms have been proposed for these automata. The L_{R-I} algorithm is actually a special case of the L_{R-P} (linear reward-penalty) algorithm; in the more general algorithm, a second parameter λ_P is introduced. The probabilities are now updated both in the case of reward and in the case of penalty response. Nonlinear schemes have also been proposed by Chandrashekaran and Shen[20], and others; these converge faster under certain circumstances, usually at the expense of accuracy.

The emphasis of the literature on automata is on proofs of convergence. For the L_{R-I} scheme, bounds on the probability of convergence to the optimal action have been obtained by Norman [66].

Although the schemes were designed for simple reward-penalty environments for the most part, Viswanathan and Narendra [88] showed that the proofs of convergence also apply for the more general case where the response x is in the interval $[0,1]$. As we mentioned, if the observations from the distributions do not lie in the interval $[0,1]$, the observations are normalized by dividing $(x-l)$ by $(u-l)$, where u and l are the upper and lower bound on the values of observations. If these bounds are not known a priori, they can be estimated concurrently with the operation of the automaton with good results[88]; this is the approach that we follow in chapter 5 to handle cost values from existing databases.

Thathachar and Sastry's algorithm

Recently, Thathachar and Sastry [85] have introduced a learning algorithm that is claimed to converge six to ten times faster than previous algorithms. Quick convergence is specially important for our purposes: for fixed environments, the faster the selector begins choosing the best strategy, the lower will be the accumulated cost for large values of time; and for the changing-environment extension that we will propose in section 4.4, quick convergence allows better tracking of the changing environment.

In this algorithm, the model M includes the vector p , and also for each strategy k, the running sum of the observations σ_k, the number of the observations n_k, and the running average $\bar{x}_k$. After receiving a feedback cost x from the execution of strategy s_k, M is updated as follows:

Algorithm I

$$\tau_I(M,k,x) \text{ where } M =< p,\sigma,n,\bar{x} >$$

$$\begin{aligned}
n_k &= n_k + 1; \sigma_k = \sigma_k + x; \bar{x}_k = \sigma_k/n_k \\
p_j &= p_j - \lambda(\bar{x}_j - \bar{x}_k) \times \left(if\ \bar{x}_j > \bar{x}_k\ then\ p_j\ else\ \frac{p_k(1-p_j)}{K-1} \right) \quad for\ all\ j \neq k; \\
p_k &= 1.0 - \sum_{j \neq k} p_j
\end{aligned}$$

First note that if the probabilities add up to 1 before updating, they will also add up to 1 after updating. Probabilities of strategies that look currently worse than s_k (higher current average) are decremented; those strategies that look better than s_k (lower current average) have their probabilities incremented. The factors that appear inside the if-expression ensure that no probability is ever decremented below 0 or incremented beyond 1.

To start up the process, a number of *training observations* are made on each strategy, in order to compute initial values for the averages; or, in other words, if η denotes the number of training observations, the first $\eta \times K$ queries will be executed using each of the K strategies η times.

Thathachar and Sastry prove in their paper that, regardless of the environment, there exists a value of λ that will make $p_l^{(t)} \rightarrow 1$ in probability as $t \rightarrow \infty$. If λ exceeds this critical value, the automaton might converge to a sub-optimal action. They also show the results of one simulation of Algorithm I running against ten uniform distributions. The means of the distributions are randomly spaced; the widths are such that one extreme of the distribution touches either 0.0 or 1.0. Using a value of $\lambda = 1$, the automaton converges to the best strategy with a probability of 0.99 in 98 steps on average, versus 784 steps employed by the L_{R-I} scheme.

There are several important aspects that are not answered in the paper. It is not clear what is the effect of the following items on the performance measures mentioned in section 4.2:

- the value of λ: The authors use the maximum value of λ for the simulation reported in the paper. It is not clear under which circumstances the value should be different.
- the number of strategies: How does convergence speed vary as a function of the number of strategies?
- the proximity of the means of adjacent cost distributions.
- the overlap between cost ranges of adjacent distributions.

In order to answer these questions, we performed a new series of simulations which we are reporting in the following section.

4.3.1 Simulation results

Simulation 1

Objective. The purpose of this simulation is to determine convergence speed under favorable conditions, and to observe the variation of convergence speed as a function of the number of strategies.

Input. The most favorable conditions are clearly non-overlapping cost distributions; in this case, a single observation on each distribution would suffice to determine the best one. We would like to see how much worse than that the automaton performs.

There are K uniform distributions with means evenly spaced in the interval $[0,1]$; their ranges are such that each distribution just touches its neighbors. For example, for $K = 2$ we would have one distribution centered at 0.25, the other at 0.75, and both ranges equal to 0.5. The parameter values are $\eta = 1$ and $\lambda = 1$. Table 4.1 lists the time to convergence to a probability $p_1 = 0.98$ for several values of K. We list the minimum, maximum, average and standard deviation of the time over 1000 experiments.

K	Time			
	Min	Max	Avg.	Std. dev.
2	4	17	8.418	2.351
3	7	28	13.739	3.507
4	9	35	19.207	4.805
5	12	44	24.662	6.277
6	13	57	30.208	8.284
7	16	66	34.749	9.863

Table 4.1: Convergence time for K evenly spaced uniform distributions

Results. We observe that the average time to convergence increases linearly with K. The variance of the time to convergence also increases with K.

Simulation 2

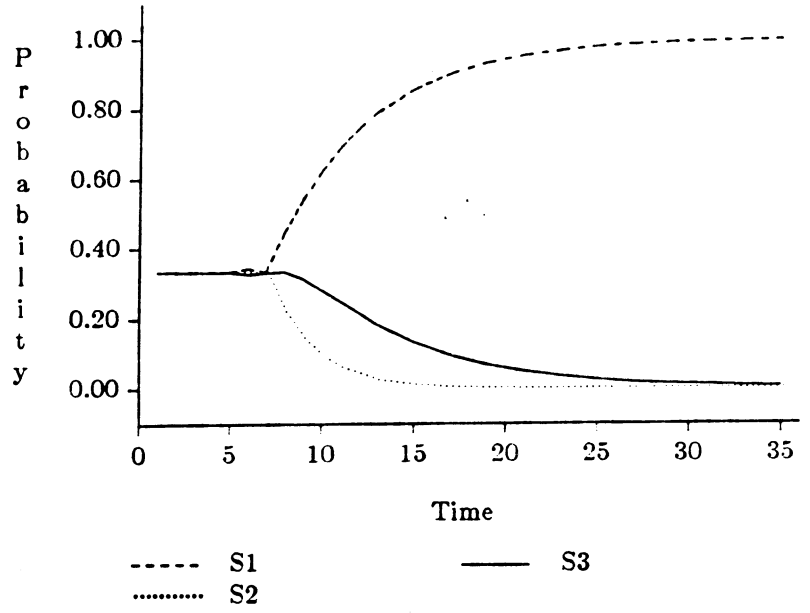
Objective. To observe accuracy and convergence speed as a function of the proximity between the two smallest means

Input. There are $K = 3$ uniform distributions. Distribution 1 ranges from 0 to 0.5, distribution 2 from 0.5 to 1, and the third distribution overlaps the other 2. Each row of Table 4.2 corresponds to a different position of distribution 3, starting when μ_3 is centered at 0.5 and then moving to the left. All three distributions have a range of 0.5.

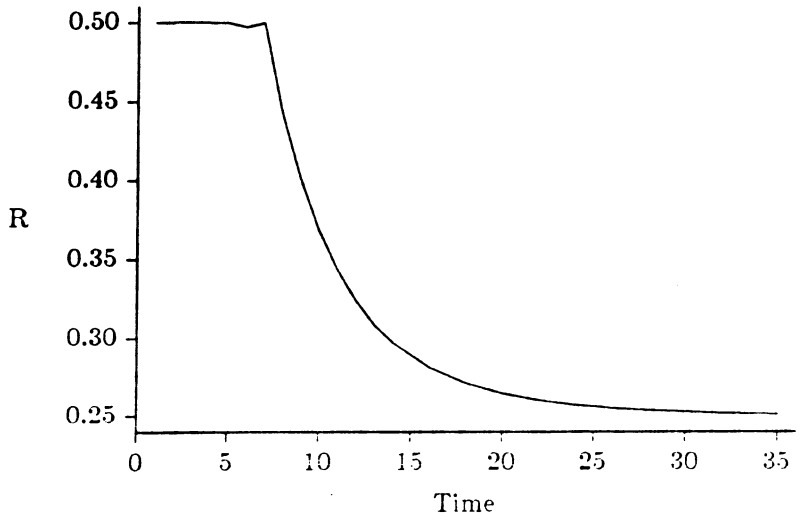
Results. As the two smallest means get closer, the average time to convergence increases; the standard deviation of the time to convergence also increases. There is a point when accuracy drops below 100%. The last row shows that lowering the value of λ restores accuracy back to 100%.

Figures 4.2 and 4.3 show the evolution of the three probabilities and the average expected reward for two rows of the table.

The data used to produce the figures are averages over 1000 experiments. Figure 4.2 corresponds to the first row, with $\mu_3 = 0.5$; figure 4.3 corresponds to row 5, the first row where accuracy drops below 100%. In this second case the average expected reward tends towards 0.26 rather than its minimum 0.25, which is still quite acceptable. When distributions are extremely overlapped, small values of λ

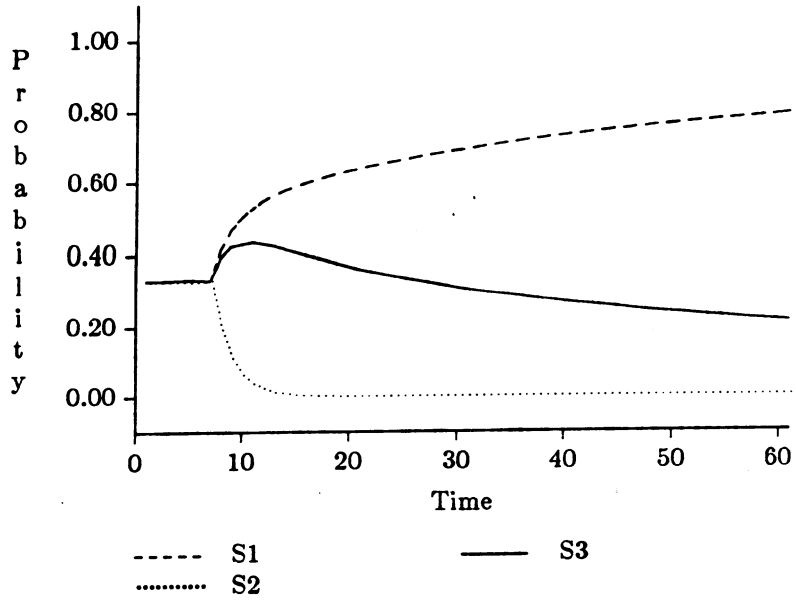


(a) Probabilities over time

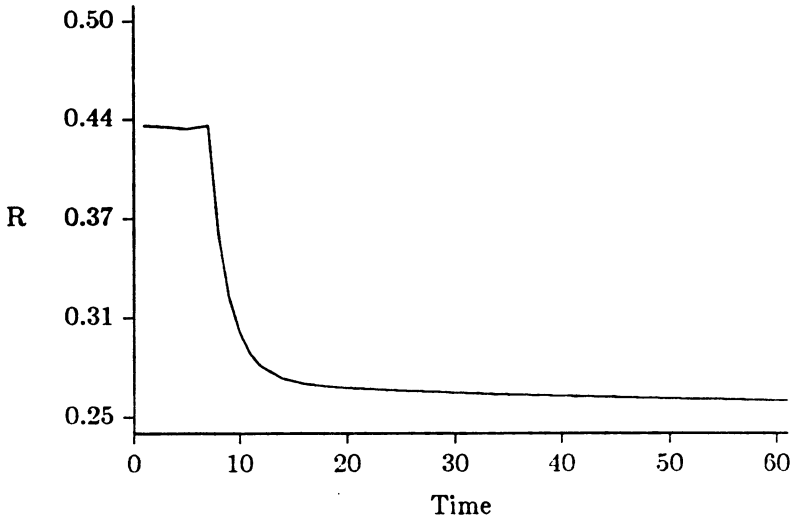


(b) Average expected reward

Figure 4.2: Moderately overlapped distributions.



(a) **Probabilities over time**



(b) **Average expected reward**

Figure 4.3: Highly overlapped distributions

μ_3	λ	Time				Accuracy
		Min	Max	Avg.	Std. dev.	
0.5	1.0	11	53	21.608	6.3739	100%
0.45	1.0	11	78	26.833	10.2013	100%
0.40	1.0	12	140	34.553	16.7630	99.8%
0.35	1.0	13	238	52.372	33.5763	99.1%
0.30	1.0	15	574	95.355	77.7107	92%
0.30	0.3	44	1402	300.962	185.8660	99.6%

Table 4.2: Convergence time and accuracy as two smallest means get closer

may be required for accuracy. A test was made with $K = 2$, $\lambda = 1$, the first distribution from 0.0 to 1.0, and the second from 0.2 to 1.0. This leaves the two means at the same distance as in row 5 of the previous table; however, this time accuracy drops to 88%, and $\lambda = 0.2$ is required for 100% accuracy (1000 experiments). For database purposes, however, a convergence to the best between two extremely close distributions may not be essential; a value of $\lambda = 1$ should be suitable in most circumstances.

4.3.2 Comparison with optimal policy

Here we use the approximate selector with the same example used at the end of chapter 3 to illustrate optimal policies. The time horizon is rather small ($h = 8$), and therefore the adaptive selector has hardly begun to converge to the best strategy when time is up. Nevertheless, its performance is acceptable compared with the optimal. The figures reported below are averages of 10,000 experiments.

We recall that the example involves two families of two discrete distributions each, as follows:

	F_1			
$\pi_1^{(0)}$	1	2	3	Mean
0.5	0.1	0.1	0.8	2.7
0.5	0.1	0.5	0.4	2.3
				2.5

	F_2			
$\pi_2^{(0)}$	2	3	4	Mean
0.5	0.8	0.1	0.1	2.3
0.5	0.4	0.3	0.3	2.9
				2.6

There are four possible ways of selecting one distribution from each family. For each of these, we list the average expected reward of the approximate selector on top of the one for the optimal policy.

		1	2	3	4	5	6	7	8
s1: 0.1 0.1 0.8	app	2.500	2.499	2.500	2.479	2.464	2.450	2.439	2.430
s2: 0.8 0.1 0.1	opt	2.700	2.378	2.411	2.463	2.431	2.425	2.442	2.438
s1: 0.1 0.1 0.8	app	2.800	2.802	2.800	2.795	2.791	2.789	2.786	2.784
s2: 0.4 0.3 0.3	opt	2.700	2.861	2.781	2.735	2.762	2.746	2.726	2.729
s1: 0.1 0.5 0.4	app	2.300	2.300	2.300	2.300	2.300	2.300	2.300	2.300
s2: 0.8 0.1 0.1	opt	2.300	2.300	2.300	2.300	2.300	2.300	2.300	2.300
s1: 0.1 0.5 0.4	app	2.600	2.591	2.600	2.553	2.523	2.500	2.483	2.468
s2: 0.4 0.3 0.3	opt	2.300	2.541	2.419	2.348	2.362	2.346	2.327	2.325

Consider the first case. This involves a distribution with mean 2.7 for strategy 1, and 2.3 for strategy 2. The first average reward for Algorithm I is 2.5, the average of the two means, since at $t = 1$ there is a training observation that uses each strategy with equal probabilities. For $t = 2, 3, \ldots$ we can see that the approximate algorithm begins to converge towards strategy 2, but does not reach convergence when time is up. The optimal policy starts sampling strategy 1 always; we saw in chapter 3 that this is the optimal first choice *considering all possible future outcomes of this decision.* The average rewards go up and down always according to optimal policy choices.

Since the prior distributions for both families are flat, all four cases are equally likely. Thus the expected value at each time step, averaging over all possible environments is:

2.55 2.54827 2.55 2.53185 2.51945 2.50994 2.50215 2.49566

These numbers add up to 20.20732. As we saw, the expected value for the optimal policy is 19.77563520. We can see that, on average, the learning automaton approach does better than random choice and but not so well as optimal. Unfortunately, the small value of the time horizon that we were able to process with the optimal policy design program, prevents us from making any accurate estimates about the difference in performance of the two approaches in general.

4.4 Changing environments

We say that the environment changes when the cost distributions X_k do not remain static while the automaton is in operation. The changes may involve shiftings of the value of the mean or other more major changes; and they can be gradual or sudden, and periodic or random.

There has been some work concerning changing environments for particular situations. Barto, Anadan and Anderson [6,7] have considered the case where the environment can be in one of a finite number of states, known to the automaton. An environment state vector is fed to the automaton, which then chooses an action and receives a reward-penalty response. They give an algorithm that enables the automaton to learn the best action for each state. Narendra and Viswanathan [65] have considered the case of periodic environments. The period is subdivided into

equal intervals, and a separate automaton handles each interval. Thathachar and Sastry [85] consider a Markovian switching environment, in which all distributions can switch between two states.

For database systems we need a more general approach, since it is very difficult to predict in advance the kind of changes that will occur. Since the automaton tends to "lock" to the best strategy, the change of most interest is a variation in the location of the distributions that causes the best strategy to lose first place. Other changes should ideally be ignored. We will utilize the following measures of performance:

- the speed of adaptation to a change: Assuming that the automaton has reached steady state and there is a *sudden* change, the automaton should adapt quickly to the new environment and pick the new best strategy.
- the *tracking ability*: This refers to smoothly evolving environments, where there is as gradual movement of distribution positions that eventually causes a change in the identity of the best. The automaton should track the best strategy accurately.

We introduce two modifications to Algorithm 1 that together have given satisfactory results. First, the averages $\bar{x}_k$ are now computed with a simple exponential smoothing scheme [17], as follows:

$$new\ average = (1-\alpha)\times\ old\ average + \alpha\times\ new\ observation$$

This has the effect of weighting the last observation by a factor α, and weighting previous observations at times $t-1, t-2 \ldots 0$ by factors $\alpha(1-\alpha)$, $\alpha(1-\alpha)^2$ and so on. Thus a high value of α places more weight on recent observations making the average more sensitive to change, although also more "jumpy". For very small values of α, the exponentially smoothed average tends towards the value of the conventional (equally weighted) average.

The second modification is to limit the minimum value that a probability may reach. In Algorithm I, this minimum value is zero. For changing environments however, it is not possible to detect changes which affect a strategy that is not being sampled at all; therefore the limiting value, $pmin$. Likewise, the maximum probability value cannot exceed $pmax = 1-(K-1)pmin$. Putting these two modifications together gives Algorithm II:

Algorithm II

$$\begin{aligned}
\bar{x}_k &= \bar{x}_k(1-\alpha) + x\alpha \\
p_j &= p_j - \lambda(\bar{x}_j - \bar{x}_k) \times \left(if\ \bar{x}_j > \bar{x}_k \ \ then\ (p_j - pmin)\right. \\
&\qquad \left. else\ \frac{(p_k - pmin)(pmax - p_j)}{(K-1)(pmax - pmin)}\right) \quad all\ j \neq k; \\
p_k &= 1 - \sum_{j \neq k} p_j
\end{aligned}$$

The maximum allowed value of $pmin$ is clearly $1/(K-1)$. The initial values of the $\bar{x}_k$ are again computed taking a fixed number η of observations on each strategy

and computing a standard average with them. This initial value is also used as the old value for the first computation of the exponentially smoothed average. The first part of the *if expression* ensures that, even when $\lambda = 1$ and $x_j - x_k = 1$, p_j will not be decremented to a value smaller than *pmin*; the second part ensures that, even if all the p_j's are incremented, none of them will go beyond *pmax*, and $p_k > pmin$. Thus all probabilities remain within the specified bounds.

4.4.1 Simulation results

Objective. To observe tracking ability of Algorithm II.

Input. In this simulation, there are two uniform distributions of range 0.1, initially located at 0.05 and 0.5 respectively. From the start of the experiment, the distribution of strategy 1 begins to oscillate back and forth between its initial location at 0.05 and location 0.95, far to the right of distribution 2. The distribution moves at a constant speed v, measured in distance per time step. The second distribution remains static. We used $\lambda = 1$, $\alpha = 0.9$, and a minimum sampling probability *pmin* of 0.1. Again, the figures show averages of 1000 experiments.

Results. Figure 4.4 shows the results for a speed of 0.025. The figure shows

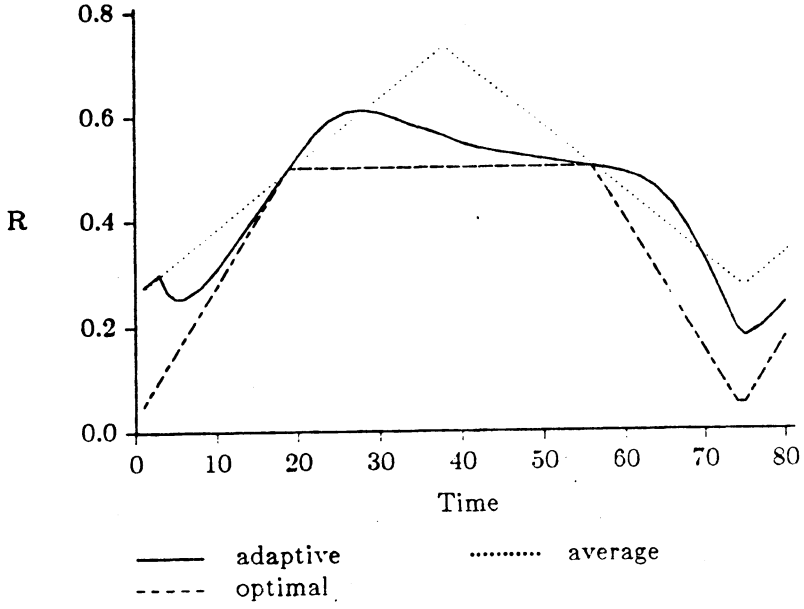


Figure 4.4: Oscillating distribution, speed 0.025

three curves: One curve, the optimal case, shows the value of the minimum of the two means; another the value of the average of the two; the third the value of the expected reward of the adaptive selector. Consider first the optimal curve. At $t = 0$ the optimal is 0.05. Then the curve goes up in a straight line as the distribution of strategy 1 moves right. At $t = 20$ the two distributions overlap; from that point onwards the optimal remains at 0.5. At $t = 40$ strategy 1 hits 0.95 and bounces

back to the left. At $t = 60$ there is another overlap, and the minimum curve begins to go down accompanying strategy 1 until $t = 78$ when a full cycle is completed. The average curve goes up and down in a sawtooth fashion.

If the adaptive selector were to track the minimum strategy perfectly, its curve would overlap the minimum curve. Instead, we can see that between $t = 0$ and $t = 20$ it lags the minimum a little and then it overshoots. The reason for the overshoot is that at $t = 20$ when the distributions cross, $s1$ is being sampled at a high rate, whereas the now-minimum $s2$ is only sampled 10% of the time. Sampling of s2 begins to increase little by little, detectably often at $t = 30$. On the way back there is a similar overshoot. The two hashed areas mark two small intervals of time where the adaptive selector does worse than a selector that would choose the strategies at random. At all other times it outperforms random selection.

Consider now choosing a fixed strategy all the time. Selecting strategy 1 would produce an horizontal line at y=0.5. We can see that this would improve upon adaptive selection by a small amount from $t = 22$ to $t = 58$; the rest of the cycle, adaptive selection beats fixed strategy 1 selection by a larger amount, thus on average adaptive selection will be bether than fixed strategy 1 selection. Similar comments can be made about fixed strategy 2 selection.

Figure 4.5 repeats the experiment at double speed, $v = 0.05$. Two cycles are shown. The hashed areas are now bigger.

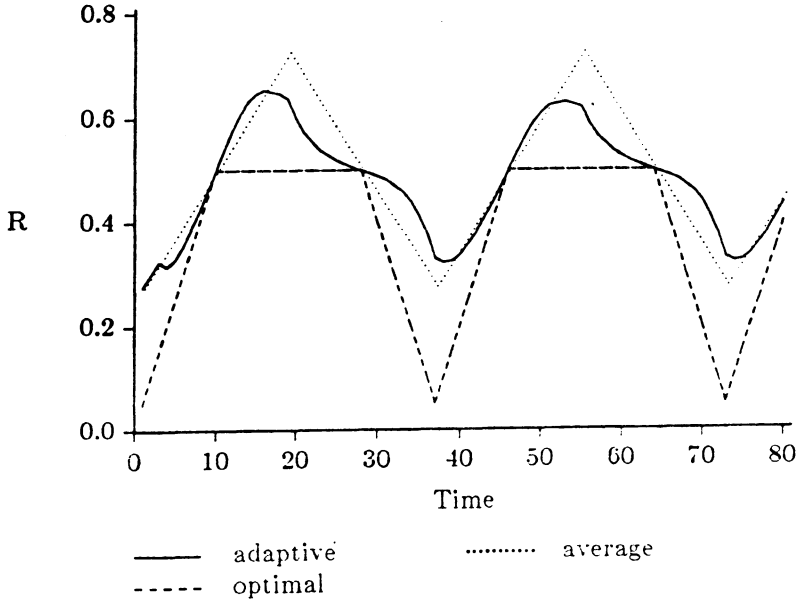


Figure 4.5: Oscillating distribution, speed 0.05

One might think that the performance of Algorithm 2 *in fixed environments* would deteriorate as the sensibility to change α is increased. This turns out to be the case only when the cost distributions are highly overlapped, and even here the deterioration is small.

To show this, we now consider three distributions with high overlaps: 0–0.7, 0.2–1.0, and 0.25–1.75. Algorithm I converges to probability 0.98 with 94% accuracy in 53 steps on average.

Algorithm II was run with weights equal to 1, 0.5 and 0.1. Note that a weight of 1 effectively ignores all previous observations; the "average" is equal to the last observation. Figure 4.6 shows the average expected reward curve for each case, and also the curve for Algorithm I. We see that Algorithm I beats Algorithm II by a

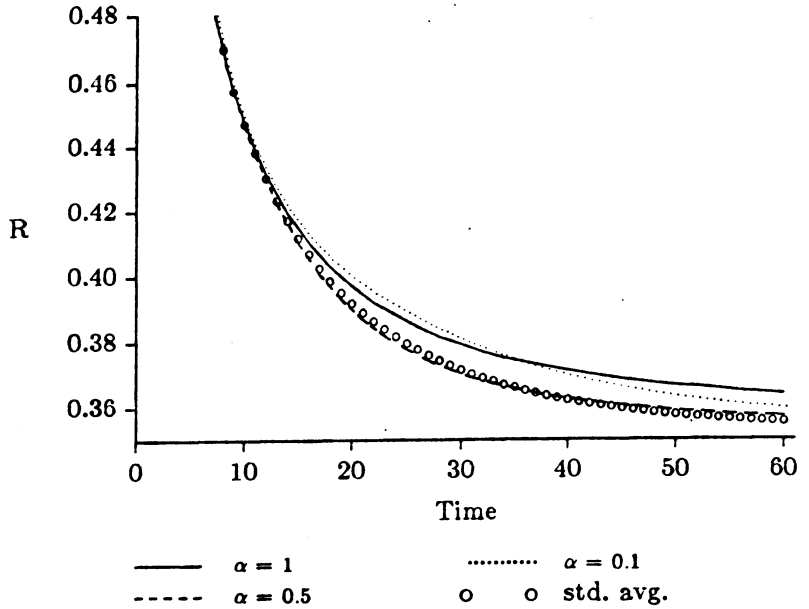


Figure 4.6: Smoothed average on constant environment. High overlaps.

small amount; for $\alpha = 0.5$ the two curves almost overlap. $\alpha = 0.1$ is the loser due to the high inertia of the exponentially smoothed average at very low values of α.

We can see that Algorithm II with a relatively high weight should provide good sensitivity and adaptability to change, without degrading performance for the fixed-environment case.

4.5 Other automata algorithms

Several attempts were made to improve the performance of Algorithm I by using measures of the reliability of the sample means $\bar{x}_k$ as estimates of the true means μ_k. The results were inconclusive. In one approach, designed for normal distributions with known variance, the term $(\bar{x}_j - \bar{x}_k)$ was replaced by $2(Pr(\mu_j - \mu_k) - 0.5)\lambda$ where $Pr(\mu_j - \mu_k)$ is the likelihood that μ_j is indeed less than μ_k given the sample evidence (Algorithm III). This likelihood can be computed as the area under the standard normal curve to the right of

$$\frac{\bar{x}_j - \bar{x}_k}{\sqrt{\frac{\sigma_j}{n_j} - \frac{\sigma_k}{n_k}}}$$

Algorithm III did turn out to converge faster than Algorithm I for cases where Algorithm I had 100% accuracy. For cases where Algorithm I had a lower accuracy however, the accuracy of Algorithm III was even lower, unless the value of λ was drastically reduced, in which case there was no appreciable difference between the two algorithms.

4.6 Other algorithms not based on automata

All the adaptive plans mentioned in this chapter are based on learning automata. We do not claim that these are the best possible plans; some other possibilities that we have not fully explored are the following:

- A plan could be designed based on the optimal policy design problem of chapter 3. This would require assumptions about the distribution families that can occur and initial prior distributions over these families. The time horizon can be set to the estimated time until the next environment change. Optimal policies for several values of the time horizon could be computed off-line and stored on disk. The on-line algorithm would consist of estimating the next time of change, loading the appropriate policy, and executing it. Environment changes could be detected by a separate change-point module that watches sample observations and continuously makes tests for a shift of mean on the distributions. A variety of statistical tests exist for the change-point problem; two good candidates for our purposes are the Man-Whitney non-parametric test [5], and a Hinkley detector [51]. The estimated time for the next change can be based on the average of previous change intervals.

- Rather than handling changes by exponential smoothing as we have suggested, it is also possible to use a change-point detector to control the amount of exploration done by Algorithm I. In this scheme, the probability vector would be reset to equal probabilities when a change is detected, causing the algorithm to explore all strategies equally and probably start converging to another one. The averages can also be "forgotten" when a change is detected.

- Plans can be devised based on approximate policies for bandit problems [74,8,28]. Several of these were reviewed in chapter 3. All of the ones known to us need distributional assumptions, usually Normal or Bernoulli; further, few of them appear to be sensitive to environment changes. Results to date are all reported for fixed environments.

- Finally, there are several schemes used by statisticians to find the best of K distributions [52,41]; some of these were also reviewed in chapter 3. They have the advantage of providing a measure of the reliability of a selection; however, none of them seems easily adaptable for the dynamic problem of *continuously* finding the current best of the distributions.

4.7 Summary

In conclusion, the policies that we have presented based on learning automata provide very satisfactory performance on fixed environments plus the ability to adapt to changing environments, which may be very important for database systems. No distributional assumptions are needed, and computational requirements are minimal. In chapter 5 we will present simulation results of these algorithms running against real database query loads.

Chapter 5

Simulations on existing databases

In the previous chapter we illustrated the application of the heuristic adaptive selectors using uniformly distributed query costs. The main purpose of this chapter is to show the use of these techniques with costs computed from real query loads.

First, we will consider a group of commands issued to an actual UNIX^{TM} system as queries to an imaginary database in which the code for each command has to be found. A simplified model of this database then provides to the adaptive algorithms the cost of searching this code using several strategies. It will be shown that the adaptive selectors are indeed able to choose the best strategy quickly.

Next, we run the adaptive selectors with query loads issued to an existing System 2000^{TM} database [29] . Magalhaes[58], as part of his doctoral dissertation, obtained a copy of this database and the query loads with measurements.

System 2000 applies a fixed strategy to answer queries; several other plausible strategies will be defined and evaluated with the help of a detailed model of System 2000. Again, we will show that the adaptive selectors do converge to the correct strategy under these real-life conditions, and that the selectors are able to adapt to changes in the data or the query load that cause a change in the identification of the best strategy.

5.1 The UNIX commands database

The UNIX operating system has to search for the code that implements a command; this code is then loaded to execute the command. We will assume (unrealistically) that the code for all commands is contained in a simple "database" that consists of three disk blocks. At system generation time, code for each command is stored in one of the blocks. When a command is later issued, these blocks have to be searched one by one until the command code is found; however, the *order* in which the blocks are searched can be varied. What is the best order to search the blocks? In our terminology, each one of the 6 possible orders corresponds to an execution strategy . The cost of execution of a strategy can be equal to 1, 2 or 3, indicating the number of blocks that need be searched to find the command code.

An analytic solution to this problem is possible only by making assumptions about the distribution of the commands in blocks and "queries". The typical *uniformity* assumptions would clearly lead to the conclusion that any strategy is equally good. Another possibility is to gather statistics in order to find probabilities of submission of each command and then derive the best overall strategy. But there is no guarantee that these probabilities would hold at all times. The adaptive approach derives the best current strategy "on the fly" without the need of analytic models nor explicit statistics gathering by query.

5.1.1 The database and the query load

A set of 400 commands was gathered one arbitrary day and time during one hour of execution, by using the *lastcomm* system utility. The commands were issued by all users that happened to be logged on at that time. This set of commands constitute our "query load"; the first few are listed in table 5.1. It turned out that there were 82 *distinct* commands in the query load. A hash function was then applied to determine the assignment of each command to one of the three disk blocks. Table 5.2 shows the resulting assignments. 28 commands landed in the first disk block, 33 in the second block and 21 in the third.

5.1.2 Strategies and costs

Now consider the six strategies in table 5.3(a). The cost of searching a command using each strategy depends on the block assigned to the command. For instance, searching a command in block 3 with strategy 2 ("search 1 then 3 then 2") would have a cost of 2. The costs for each of the three possible block assignments are shown in Table 5.3(b).

Table 5.4 shows the first part of the 400 commands in the query load, together with their block assignments, and the cost of searching each command using each strategy.

	(cont.)	(cont.)	(cont.)	(cont.)
ddumb	expr	soelim	ls	tbl
vi	pwd	ddumb	lpq	Mail
man	lpr	rn	lpq	more
more	typeset	whoami	sh	more
ls	rm	tset	csh	man
lpd	itcsh	searchpath	sendbatch	more
sendmail	troff	whoami	sh	typeset
local	eqn	hostname	*disperse*,	rm
csh	mv	sendmail	rm	itcsh
rm	pic	ps	square	troff
lpr	edit	lpq	square	eqn
sh	gts	w	awk	tbl
sendmail	csh	cat	sed	cat
sendmail	sh	grep	expr	grep
dcan.prefi	awk	wmi	tee	wmi
dcan	rn	typeset	dmesg	typeset
sh	csh	rm	tee	rm
cat	chmod	hostname	cat	sendmail
mv	sh	sed	cron	local
sendmail	atrun	typeset	cron	sendmail
local	cron	sendmail	lpq	sendmail
sendmail	daps	sendmail	lpq	sendmail
sendmail	sh	typeset	typeset	date
sendmail	aps5	Mail	rm	hostname
mv	sh	rm	itcsh	sed
mv	rsh	csh	troff	typeset
lpd	rsh	more	eqn	typeset

Table 5.1: Query load: commands in order of submission

Command	Block
Mail	3
square	2
aps5	2
apsview	2
archshell	1
atrun	3
awk	1
backup	3
basename	1
biff	2
cat	2
chmod	2
clear	1
comsat	3
cron	3
csh	2
daps	2
date	2
dcan	2
dcan.prefi	2
ddumb	3
diff	3
disperse_s	2
dmesg	2
echo	3
edit	2
eqn	3

Command	Block
expr	2
getquote	1
grep	1
gts	1
hgotf	2
hostname	2
hw	1
hw_lpf	1
hwprefilte	2
inews	2
isss	1
itcsh	1
lastcomm	2
local	1
lpc	1
lpd	2
lpf	1
lpq	3
lpr	1
ls	2
man	1
match	3
mkdir	3
more	3
mv	1
nerds	2
pic	1

Command	Block
ps	1
pwd	3
readnews	1
rlogin	1
rlogind	1
rm	2
rn	3
rsh	1
rshd	1
run	2
rwho	1
searchpath	2
sed	3
sendbatch	2
sendmail	2
sh	2
sleep	1
soelim	2
tbl	1
tee	3
troff	3
tset	2
typeset	2
unbatch	3
vi	3
w	3
whoami	1
wmi	2

Table 5.2: The database: distinct commands and block assignments

Number	Strategy
1	Search block 1, then 2, then 3
2	Search block 1, then 3, then 2
3	Search block 2, then 1, then 3
4	Search block 2, then 3, then 1
5	Search block 3, then 1, then 2
6	Search block 3, then 2, then 1

(a) Strategies

Block	Strategy					
	1	2	3	4	5	6
Number of 1's	1	1	2	3	2	3
2's	2	3	1	1	3	2
3's	3	2	3	2	1	1

(b) Costs per strategy

Table 5.3: Strategies and costs

Command		Strategy						Command		Strategy					
		1	2	3	4	5	6			1	2	3	4	5	6
ddumb	3	3	2	3	2	1	1	typeset	2	2	3	1	1	3	2
vi	3	3	2	3	2	1	1	rm	2	2	3	1	1	3	2
man	1	1	1	2	3	2	3	itcsh	1	1	1	2	3	2	3
more	3	3	2	3	2	1	1	troff	3	3	2	3	2	1	1
ls	2	2	3	1	1	3	2	eqn	3	3	2	3	2	1	1
lpd	2	2	3	1	1	3	2	mv	1	1	1	2	3	2	3
sendmail	2	2	3	1	1	3	2	pic	1	1	1	2	3	2	3
local	1	1	1	2	3	2	3	edit	2	2	3	1	1	3	2
csh	2	2	3	1	1	3	2	gts	1	1	1	2	3	2	3
rm	2	2	3	1	1	3	2	csh	2	2	3	1	1	3	2
lpr	1	1	1	2	3	2	3	sh	2	2	3	1	1	3	2
sh	2	2	3	1	1	3	2	awk	1	1	1	2	3	2	3
sendmail	2	2	3	1	1	3	2	rn	3	3	2	3	2	1	1
sendmail	2	2	3	1	1	3	2	csh	2	2	3	1	1	3	2
dcan.prefi	2	2	3	1	1	3	2	chmod	2	2	3	1	1	3	2
dcan	2	2	3	1	1	3	2	sh	2	2	3	1	1	3	2
sh	2	2	3	1	1	3	2	atrun	3	3	2	3	2	1	1
cat	2	2	3	1	1	3	2	cron	3	3	2	3	2	1	1
mv	1	1	1	2	3	2	3	daps	2	2	3	1	1	3	2
sendmail	2	2	3	1	1	3	2	sh	2	2	3	1	1	3	2
local	1	1	1	2	3	2	3	aps5	2	2	3	1	1	3	2
sendmail	2	2	3	1	1	3	2	sh	2	2	3	1	1	3	2
sendmail	2	2	3	1	1	3	2	rsh	1	1	1	2	3	2	3
sendmail	2	2	3	1	1	3	2	rsh	1	1	1	2	3	2	3
mv	1	1	1	2	3	2	3	soelim	2	2	3	1	1	3	2
mv	1	1	1	2	3	2	3	ddumb	3	3	2	3	2	1	1
lpd	2	2	3	1	1	3	2	rn	3	3	2	3	2	1	1
expr	2	2	3	1	1	3	2	whoami	1	1	1	2	3	2	3
pwd	3	3	2	3	2	1	1	tset	2	2	3	1	1	3	2
lpr	1	1	1	2	3	2	3	searchpath	2	2	3	1	1	3	2

(continuation)

Table 5.4: Query costs per strategy for UNIX commands database

Table 5.5 shows the distribution of block assignments and strategy costs for the 400 commands, summarizing the number of 1's, 2's and 3's found in each column of the complete Table 5.4. Figure 5.1 shows histograms of the cost distributions per strategy. We can see that strategies 3 and 4 are the best, having almost the same mean. The distributions remain approximately constant over time.

	Block	Strategy					
		1	2	3	4	5	6
1	98	98	98	205	205	97	97
2	205	205	97	98	97	98	205
3	97	97	205	97	98	205	98

Table 5.5: Distributions of blocks and costs

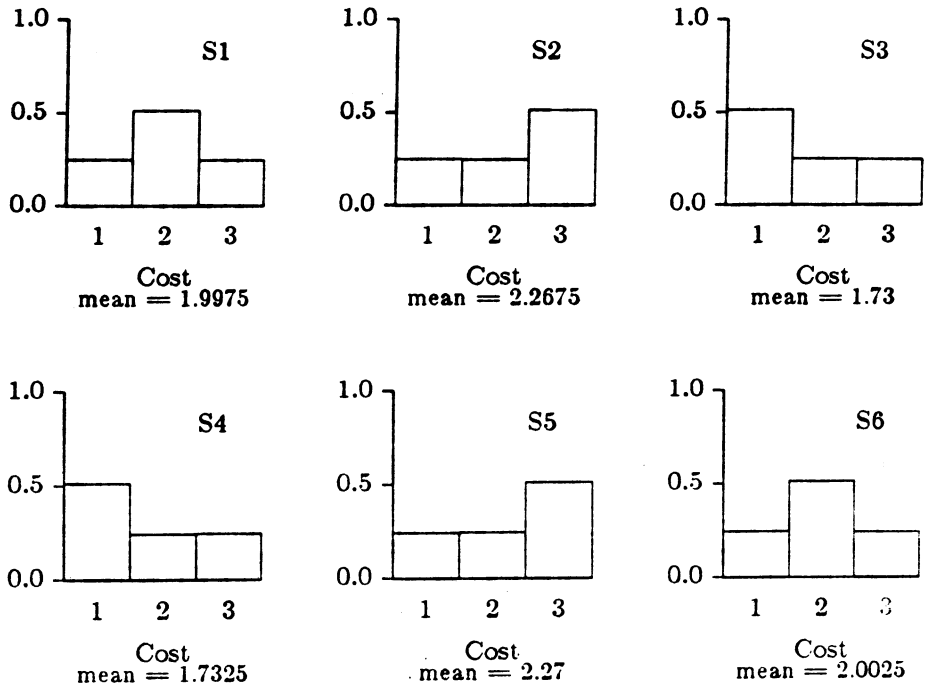


Figure 5.1: Cost distributions per strategy

5.1.3 The experiments

Adaptive selection

Table 5.4 can be conceptually put inside a black box that represents the executor module of chapter 2 (see figures 2.1 and 5.2). An adaptive selector feeds the black

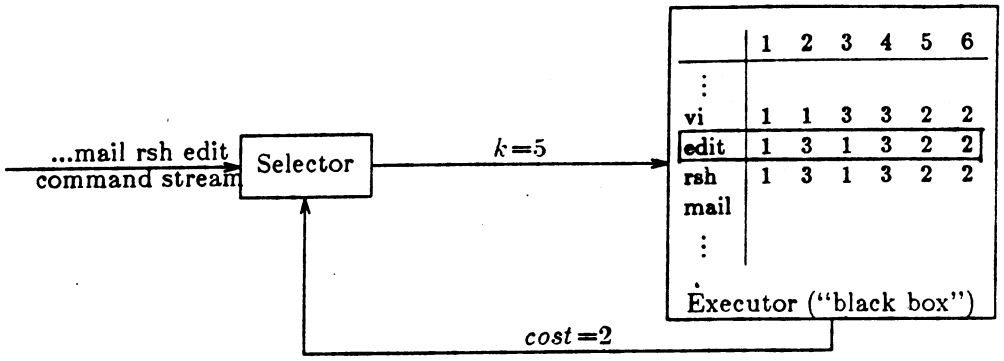


Figure 5.2: Adaptive selector on UNIX command costs

box a strategy number; the black box then accesses the corresponding column on the next row of the table, and sends the resulting cost to the selector. The selector cannot see the table; it only sees the cost fed back by the black box.

We applied Algorithm I of chapter 4 with $\lambda = 1.0$. Figures 5.3 and 5.4 show plots of the probability of executing strategies one to four as a function of time, for the first 200 of the 400 commands; after the 200th command, there is no appreciable change in the probabilities. The curves for strategies 5 and 6 are similar to those for 2 and 1 respectively. The first figure shows one arbitrary run of the selector against the black box; the second figure shows the average of 100 such runs. We notice that the most expensive strategies are quickly dismissed. On each run, the selector always ends up converging to strategy 3 or 4; which one is chosen depends on random factors.

Comparison with simpler approaches

We now plot in figure 5.5 the value of the average query processing cost over time for adaptive selection and several other approaches. Let $x^{(i)}$ be the cost obtained at time i. The value plotted is $\bar{x}^{(t)} = (\sum_{i=0}^{t} x^{(i)})/t$ for $t = 0 \ldots 200$. The first curve in figure 5.5 is for the adaptive approach. Each value plotted is itself the average of 1000 experiments. The remaining three curves are for approaches that select a fixed strategy from the beginning and stick to it until the end; we show the curves for the first three strategies.

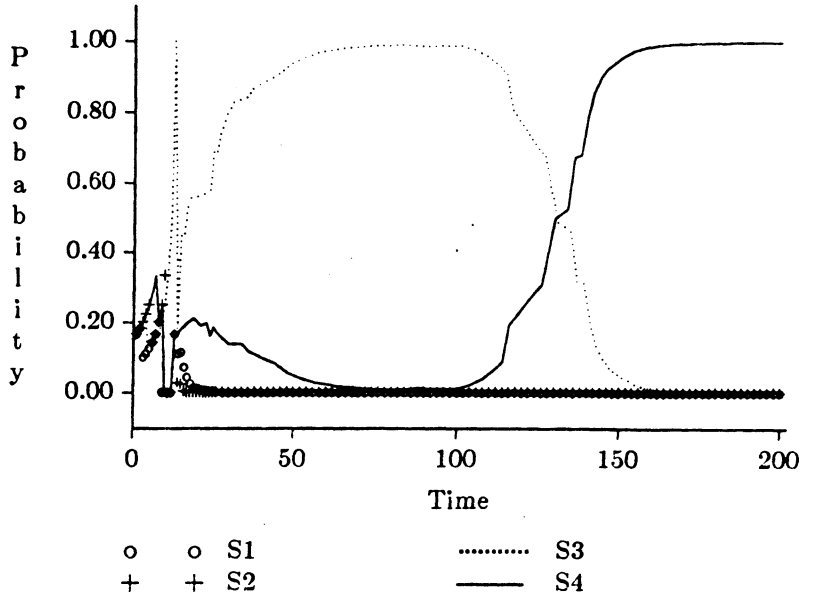


Figure 5.3: Probabilities, one sample run

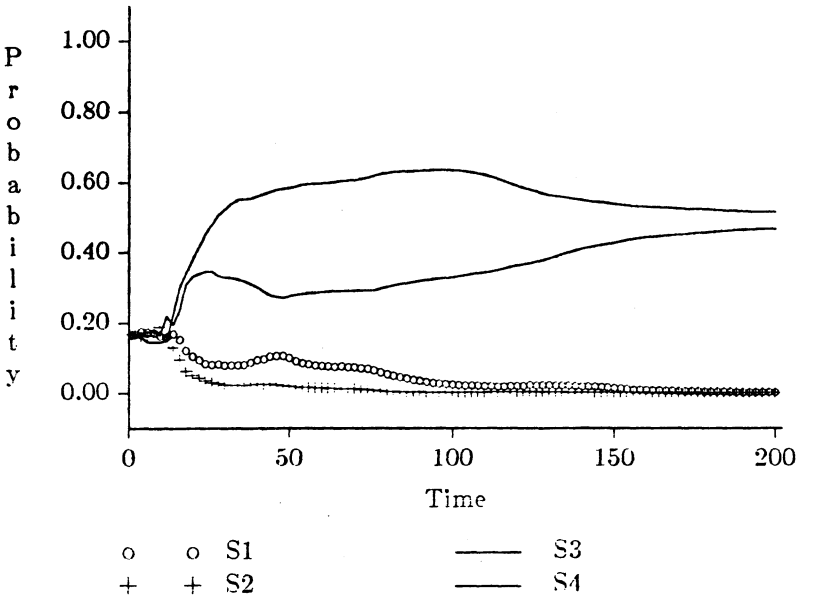


Figure 5.4: Probabilities, average of 100 runs

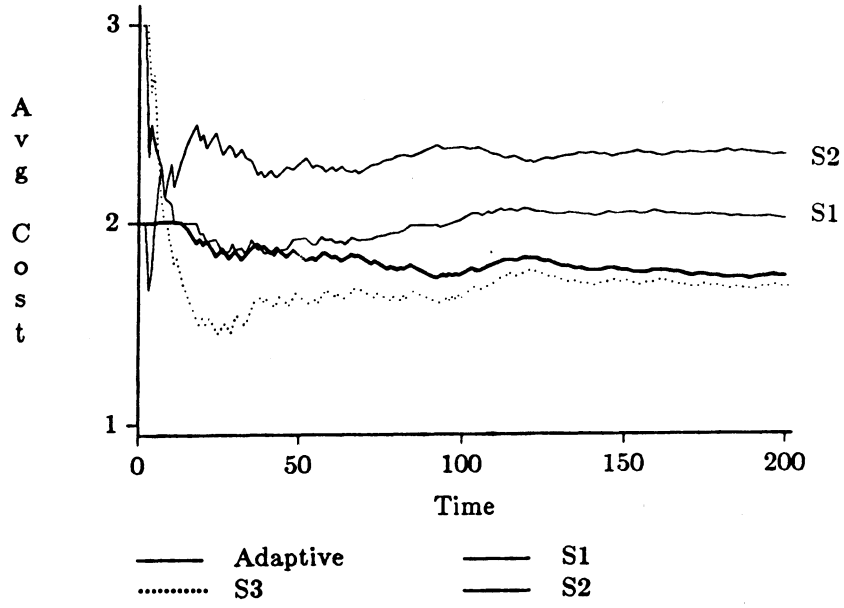


Figure 5.5: Average query processing cost over time

Consider now a random approach that throws a die to select a strategy at each time step. Since the average strategy cost for every command is 2, this would produce a straight horizontal line at *avg.cost* = 2. Similarly, an unachievable optimal approach that always selects a strategy of minimum cost would also be plotted as an horizontal line at *avg.cost* = 1. Note that an approach that selects a fixed strategy at random and then sticks with it, would perform on average just as the random approach above.

Comments

We showed that the adaptive approach does select one of the best strategies rather quickly: by approximately $t = 50$ one of the two best strategies has been selected. The overhead of the initial exploratory phase of the adaptive approach is reasonable; by $t = 100$ the average query processing cost has almost caught up with the fixed-strategy 3 approach. We note, however, that in general it is not possible to guess in advance what is the best fixed strategy to follow, since we are assuming that no cost model is available.

5.2 The System 2000 database

In 1980, Magalhaes [58] collected the transaction loads submitted during a four-day period to six operational System 2000 databases at Ontario Hydro. He also took a tape dump of the database's contents before the observation period. Then, in a test environment, he reloaded the data from the dump and re-ran the transactions

in order to take a series of measurements in controlled conditions. The databases were reorganized before the observation period in order to facilitate performance predictions.

A tape of transaction loads and measurements was kindly provided to us by Magalhaes' supervisor, Dr. C.C. Gotlieb of the University of Toronto. The tape also contained a full dump of one of the databases, "database B". The rest of the tape includes, for each transaction, the database identification, day, time, the text of the transaction and several measurements, including I/O and central processor costs per database management system module, and sequence of block accesses to the files that comprise the database. A sample page of this data is shown in Appendix B. The largest part of the resource consumption for all databases is attributable to Module "303", which handles record qualification. Some of the transactions are written using commands from System 2000 "natural language"; others are procedure calls to predefined sequences of individual commands written in the same language.

One of Magalhaes' findings is that for all databases, a few (six to ten) transaction types account for almost all resource consumption. This holds even for the databases that receive a great proportion of apparently "ad hoc" queries. This corroborates a similar finding by Rodriguez-Rosell [75] on a manufacturing enterprise.

We will be concerned with the transaction load to database B, which was the most used of the 6 databases. Most of the activity on database B is consumed by five predefined transactions; of these, the transaction labeled *insert1* is the one that consumes most resources. This is the transaction that we have chosen for further analysis and explanation in this chapter.

The rest of the section is organized as follows. In section 5.2.1 we provide a more detailed description of database B and its transactions. In section 5.2.2 we will provide a brief description of System 2000 data structures necessary to understand the execution strategies possible on this database management system. In section 5.2.3 we will describe the System 20000 model that was used to compute the costs of alternate strategies. Finally we will describe the experiments that were performed with the data.

5.2.1 The database

Database B is a half megabyte database used for room booking, among other things. Figure 5.6 shows the hierarchical structure of the database; indexed attributes are underlined. Table 5.6 shows some statistics about records and indexed attributes. Although the meaning of the record types and attributes was not provided to us, it appears that records "rec00", "rec20" and "rec60" represent a hierarchical subdivision of rooms from several buildings, "rec70" records are reservation entries for specific dates and times, and "rec80" records are used for special equipment that is sometimes needed with a reservation.

As we mentioned earlier, most of the resource consumption is due to five predefined transactions; Table 5.7 shows the average daily amount of database resources consumed by them. Transactions are identified by a label.

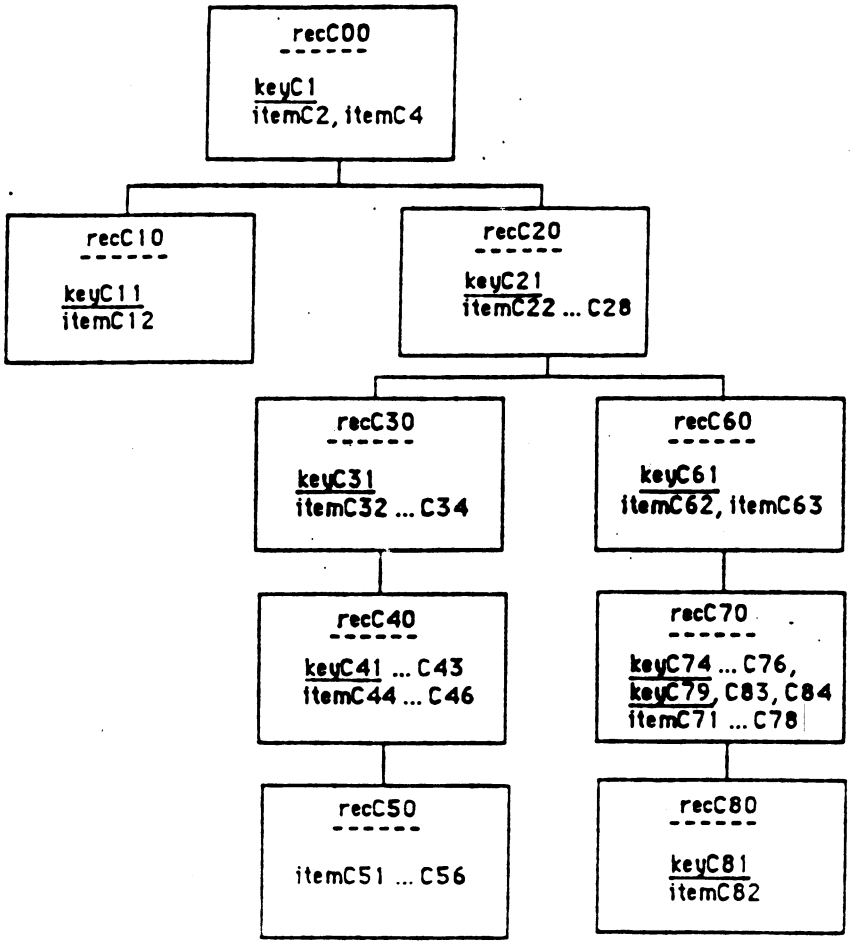


Figure 5.6: Database B

Magalhaes[58, 6.38]

Record	Length	Number of Occurrences
rec00	42	6
rec10	5	51
rec20	27	72
rec30	56	247
rec40	57	790
rec50	16	729
rec60	10	82
rec70	106	2796
rec80	23	258

Indexed Attr.	length	Distinct Values
c01	6	6
c21	6	40
c61	6	15
c74	6	290
c75	5	29

Table 5.6: Database statistics

Transaction	Frequency	Database i/o		Scratch i/o	
		avg	sdv	avg	sdv
insert1	152	95.6	14.2	2.0	0.4
list1	52	43.0	6.3	7.9	0.9
update1	26	33.6	6.3	8.5	1.9
list2	24	90.1	25.5	9.0	0.5
insert2	20	46.4	2.5	2.0	0.0
list3	5	112.5	22.5	5.0	0.0
list4	3	278.6	41.9	7.0	0.0

Table 5.7: Resources consumed by transactions

5.2.2 System 2000 data structures

In order to understand what transaction execution strategies are possible, it is first necessary to give a brief description of the internal storage structures of System 2000 (see Figure 5.7). A System 2000 database is stored in six *files*, which are called the Definitions Table (DEFIN), the Distinct Values Table (DVT), the Multiple Occurrences Table (MOT), the Hierarchy Table (HT), the Data Table (DT), and the Extended Values Table.

- The DEFIN file contains the schema definition: schema structure, attribute names, lengths and types, and so on.
- The DVT file contains a B-tree for each of the indexed attributes. The root of each B-tree is pointed to by entries in the DEFIN file. Each B-tree contains all the distinct values of each attribute. The leaf data contains a value plus a pointer. If the value occurs only once in the whole database, the pointer addresses the corresponding place in the HT file; otherwise it points to an inverted list in the MOT file.
- The MOT file contains inverted lists for each multiply-occurring value. A list is stored as a chain of blocks, where each block contains a set of pointers to the entries in the HT file corresponding to the records where the value occurs.
- The HT file stores the detailed structure of the database by means of a multi-way tree of pointers; the record data itself is stored in the DT file. There is an entry for each record of each record type. One pointer in the entry points to the actual data in the DT file. Other pointers point to the parent, first child and right sibling entry. The entry also contains an identification of the record. Note that all children of an entry are stored together in a single sibling list, even if they belong to different record types; this is why the record type id is necessary in the entry.
- The DT file contains an entry for each record. The position of the records in this file always corresponds to the position of the corresponding entry in the HT file.

5.2.3 Strategies and costs

We constructed a half-analytic, half-simulation model of a part of System 2000 in order to derive precise costs of execution of several strategies for the transactions of database B. Here we will describe one of the transactions, the transaction *insert1*, then explain the strategy followed by System 2000, mention a few other strategies that could be followed, and finally describe the model.

The transaction *insert1*

insert1 inserts a *rec70* record and then lists several other *rec70* records. The transaction has the following form:

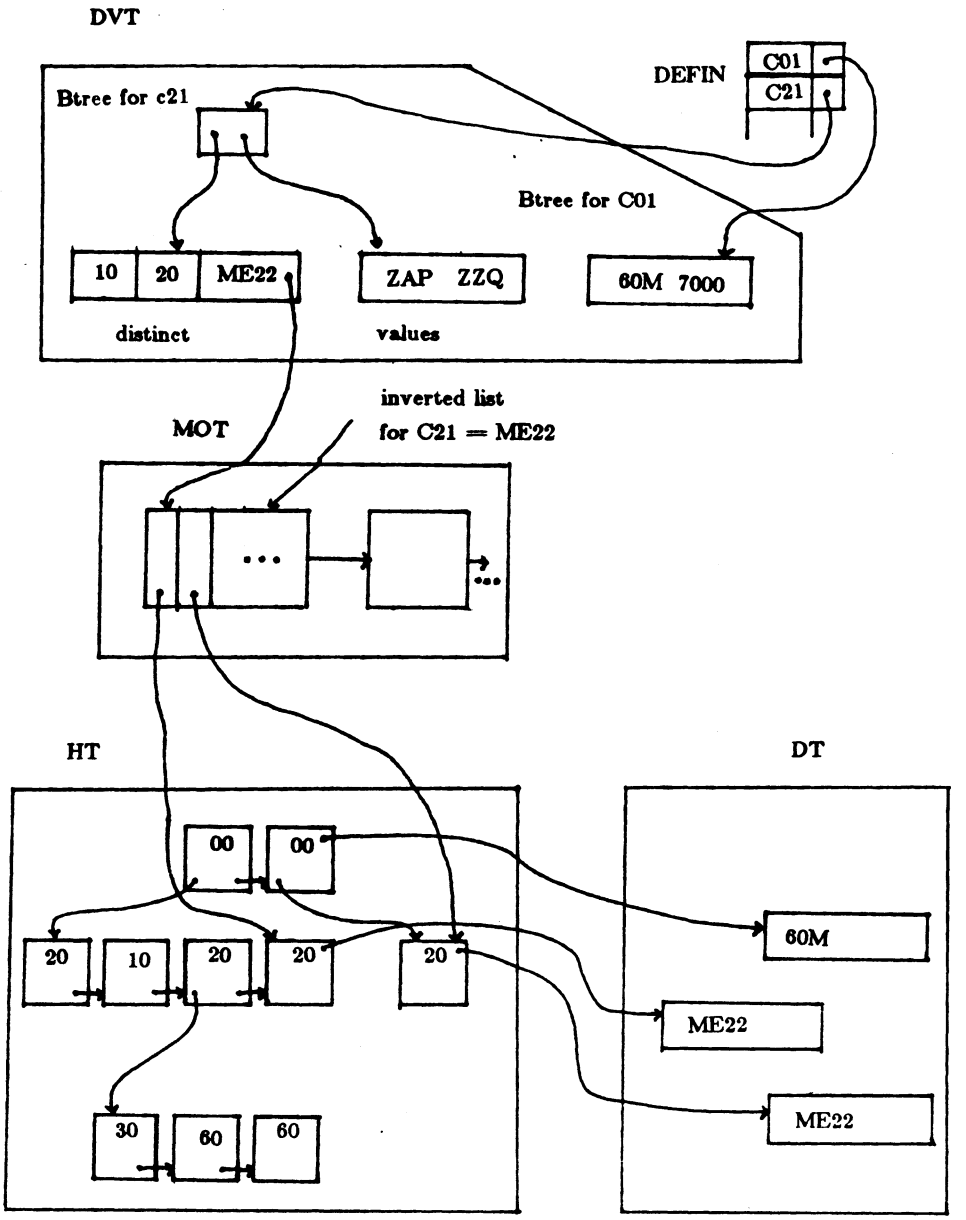


Figure 5.7: System 2000 files

```
INSERT   rec70data...   WHERE c01=x and c21=y and c61=z
LIST     rec70          WHERE c01=x and c21=y and c61=z and c74=w
```

The WHERE clause on the INSERT identifies the parent of the record to be inserted; the *rec70* records listed have the same parent, and in addition the same value of *c74* as the record inserted. There were an average of 152 *insert1* transactions per day.

System 2000 strategy

System 2000 does not use a query optimizer, but rather it executes commands following a fixed strategy. For *insert1*, this strategy is as follows: [58][sect. A.7.]

- The DVT file and the MOT are accessed to obtain three inverted lists of pointers to the HT file. We will call these lists Lc01, Lc21 and Lc61. For the example above, Lc61 contains pointers to the HT entries corresponding to records with value "z" for the field c61; similarly for the other lists.
- Lc01 is used to obtain the HT entry for the record with c01=x.
- All the children of the preceding entry with record id equal to "20" are obtained. Pointers to these entries are stored in another list that we will call Lch01. This step is called *downward normalization* in System 2000 terminology.
- Lch01 and Lc21 are sorted and intersected. This produces an HT pointer to the record with c01=x and c21=y
- Similarly, the *rec60* children of the preceding entry are obtained and intersected with Lc61, getting a pointer to the *rec60* parent of the record to be inserted.
- The record is inserted and indexes updated. The HT entry for the new record is located at the physical end of the HT file and the pointers adjusted accordingly. The actual record goes to the end of the DT file.
- For the LIST command, the preceding steps are repeated from the beginning, except that the new condition $c74 = w$ causes additional retrievals from the DVT and the MOT and another normalization step. Although System 2000 provides a facility for "remembering" the results of previous qualifications, Magalhaes reports that this facility was not used for this particular transaction; hence the repetition.

Other strategies

Our model computes the cost of this strategy and three other strategies that System 2000 could employ for this transaction. These new strategies enter the hierarchy of records at different points, and use the actual data in the DT file to check for values of attributes. Many other strategies can be imagined. The description of the new strategies follows; see also figure 5.8.

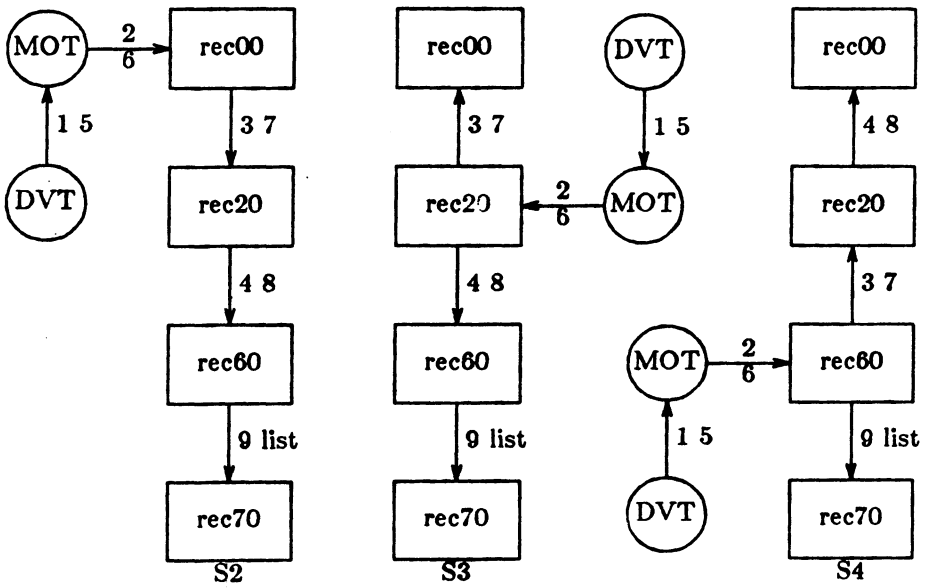


Figure 5.8: Pictorial representation of *insert1* strategies

- Strategy 2 uses the DVT to localize the *rec*00 record with $c01 = x$. The children of this entry are then scanned for *rec*20 entries that point to a DT record with $c21 = y$. The scanning stops when the entry is found. In a similar way, the children of the *rec*20 entry found are scanned looking for an entry pointing to a DT record with $c61 = z$. The new record is inserted and then the process is repeated, going this time down to the level of *rec*70 records looking for *all rec*70 records with $c74 = w$.

- Strategy 3 uses the DVT and the MOT to find all *rec*20 records with $c21 = y$. The parent of each is checked in the DT for $c01 = x$. Then the process continues as in strategy 2.

- Similarly, strategy 4 enters the hierarchy at the *rec*60 level using the DVT and the MOT for $c61 = z$. Parents are checked for $c21 = y$ and grandparents for $c01 = x$. The rest as in strategy 2.

The model

In order to compute the precise costs of these other strategies, we use analytic and simulation models. The B-tree and MOT costs are computed using an analytic model of System 2000 developed by Casas-Raposo [19] as part of his doctoral dissertation. This model utilizes average selectivities to compute costs; we modified the model slightly so that it uses *exact* selectivities measured from the actual data in the database. Thus for instance, to compute the DVT and MOT costs to get the inverted lists for $c61 = z$, the exact number of distinct values of $c61$ is used for the B-tree cost, and the exact number of records with $c61 = z$ is used for the MOT cost. Since the length of the inverted lists of the attributes is small compared with the length of a MOT block and we are considering a reorganized database, the MOT cost turned out to be one in all cases. The B-tree cost model is described in Appendix C.

The HT and DT costs were obtained with a simulation model implemented in the form of an Icon[48] program (see Appendix D). The reason for using a simulation model here, rather than the analytic models provided by Casas-Raposo, is that precise costs for these files *for each transaction* are very difficult to predict analytically; the resulting costs are heavily dependent on factors such as record distribution in blocks, the number and distribution of "foreign" children (say, *rec*10 children when looking for *rec*20's) and so on.

The database dump on Magalhaes' tape is a preorder traversal of the hierarchy; its first page is in Appendix E. Using this database dump, the program loads a copy of the database to memory and forms the data structures corresponding to files HT and DT, plus inverted lists for the indexed attributes. An extra field is added to HT entries, indicating in which page of the disk the entry would reside in the actual system. This is possible because it is known that when the database is reorganized, entries that appear contiguous in the preorder traversal are also assigned contiguous locations on the disk. Likewise, DT simulated records also indicate a disk page number within the DT file. Only the relevant attributes are

loaded into these records to save memory space; however, the actual record lengths (see Table 5.6) are used when allocating simulated records to disk.

Next, the program reads the relevant fields from each *insert1* transaction and computes the cost of executing each strategy. The transactions are read by the program in exactly the same order as they occur in the query load. The memory data structures are traversed according to the strategy, and block accesses accumulated each time a block reference differs from the previous one. This assumes a buffer pool size of 1, which is precisely the one used by Magalhaes for his experiments. The program also computes DVT and MOT accesses using the analytic models; the selectivities are obtained from the inverted lists. For simplicity, the program computes only the qualification cost; the cost of the insertion is the same for all the strategies.

Results of applying the models

The above models were applied to the query load of the second observation day. All *insert1* transactions were extracted from the query load in order of submission, and fed to the icon program described before. We thus end up with a five-column table that indicates the attribute values in the transaction and the costs for the strategy followed by System 2000, plus costs for another 3 strategies. See Tables 5.8 and 5.9 for the first 60 transactions of day 2 and their costs per strategy. The rest of the transaction load and the tables for the other three days of observation are similar.

A visual inspection of the table reveals that the identification of the best strategy is a function of the attribute values in the transaction. Strategy 2 turns out to be the best for 7 out of the 166 transactions (4.2% of the time), strategy3 is best 83.7% of the time, and strategy 4 is best 12% of the time. Strategy 1 was never the best one.

An explanation of the success of strategy 3 versus 1 and 2 can be offered by peeking at the actual selectivities and record distributions. Records *rec*00 usually have a high number of *rec*10 and *rec*20 children, and the preorder assignment together with a high number of *rec*70's causes each of the *rec*20's to be stored in a different disk block, both on the HT file and the DT file. Thus going down from *rec*00 to *rec*20 is expensive. On the other hand, attribute *c*21 has a high selectivity; thus fetching the parent *rec*00 from a few *rec*20 in strategy 3 is cheap. Checking both parent and grandparent as in strategy 4 from several *rec*60's is again expensive.

Curiously, all the transactions where strategy 3 is *not* the best involve a value for *c*01 equal to "60M" (see table 5.10). This attribute identifies a particular building. Peculiarities of the block distributions are again the cause: "60M" records happen to have very few reservations (*rec*70's), and thus all their *rec*20 children are usually stored together, making going down cheaper than going up.

Figures 5.9 and 5.10 shows cost histograms for the four strategies. The distributions of costs were examined for several subgroups of the table; they remain approximately constant all the time.

c01	c21	c61	c74	s1	s2	s3	s4
700U	15	C	800206	63	75	26	98
60M	1	103	800204	26	18	18	16
700U	05	D	800204	63	36	26	38
700U	05	D	800206	63	36	26	38
700U	04	C	800207	63	31	25	31
700U	04	C	800208	63	31	25	31
700U	05	C	800205	63	35	25	37
700U	06	D	800211	63	41	25	45
700U	06	D	800212	63	41	26	46
700U	06	D	800213	63	42	26	46
700U	06	D	800214	63	42	26	46
700U	06	D	800215	63	42	26	46
700U	02	A	800204	64	20	20	26
700U	13	A	800227	63	70	26	78
700U	16	C	800205	63	80	24	102
700U	02	D	800206	63	23	23	21
700U	06	B	800219	63	40	24	48
700U	17	C	800205	61	83	21	109
700U	09	B	800204	61	52	22	66
700U	05	C	800204	63	36	26	38
700U	11	C	800206	65	60	24	72
700U	05	C	800207	63	36	26	38
700U	03	D	800204	63	30	26	26
700U	MEZZ	A	800207	65	17	17	19
700U	07	C	800204	61	43	23	49
700U	18	B	800225	61	87	21	115
700U	09	A	800208	61	49	19	57
700U	19	B	800204	61	92	22	122
700U	08	C	800204	61	46	22	54
700U	07	D	800205	61	44	24	50

Table 5.8: Transaction costs for each strategy —part I

(continuation)

c01	c21	c61	c74	s1	s2	s3	s4
700U	10	B	800208	67	62	30	78
700U	06	A	800212	64	39	23	46
700U	18	D	800204	61	88	22	110
700U	07	C	800206	61	44	24	50
700U	MEZZ	AUD	800205	35	25	25	25
700U	MEZZ	AUD	800205	34	25	25	25
700U	07	C	800206	61	44	24	50
700U	10	A	800211	66	54	22	62
700U	03	A	800204	64	24	20	30
700U	10	C	800212	65	61	29	70
700U	10	C	800214	65	62	30	70
700U	14	C	800208	63	72	24	90
70U	09	D	800204	4	1	4	28
700U	09	D	800204	61	52	22	62
60M	1	103	800206	26	19	19	18
700U	15	D	800206	63	76	26	92
700U	04	B	800205	63	29	23	35
700U	08	B	800212	62	45	21	59
700U	08	B	800212	61	45	22	60
700U	09	B	800226	61	54	24	68
700U	16	B	800207	63	81	25	105
700U	09	B	800214	61	54	24	68
700U	09	B	800214	61	54	24	68
700U	17	C	800204	61	85	23	111
700U	10	C	800211	65	64	32	72
700U	07	B	800222	61	43	23	55
700U	06	A	800228	64	42	27	49
700U	09	A	800214	61	51	21	59
700U	07	B	800212	61	44	24	56
700U	13	A	800212	63	72	28	80

Tabie 5.9: Transaction costs for each strategy —part II

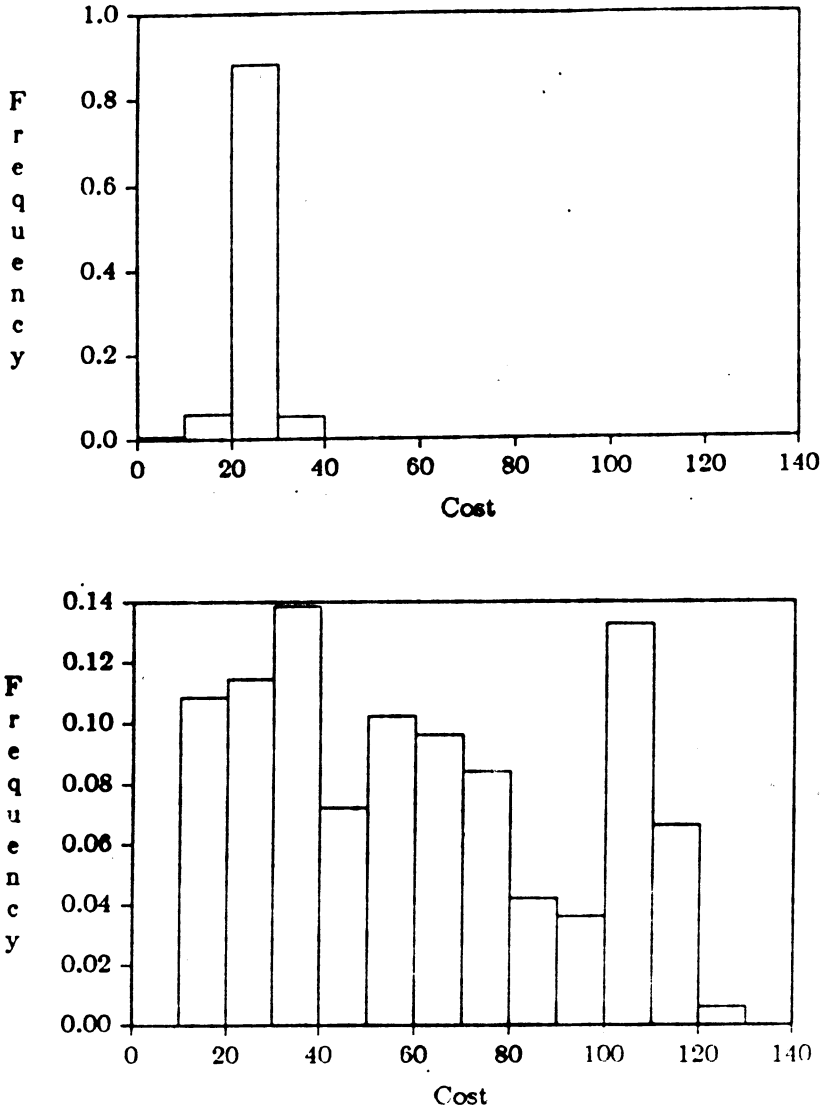


Figure 5.9: Cost histograms for strategies 1 and 2

c01	c21	c61	c74	s1	s2	s3	s4
60M	1	103	800204	26	18	18	16
60M	1	103	800206	26	19	19	18
60M	1	103	800205	26	21	21	19
60M	4	405	800205	26	13	19	11
60M	5	505	800207	23	16	24	18
60M	5	505	800205	23	18	26	20
60M	1	103	800207	26	23	23	21
60M	4	405	800218	26	15	21	13
60M	4	405	800218	27	15	21	13
60M	4	405	800211	27	16	22	14
60M	4	405	800211	27	16	22	14
60M	4	405	800211	27	16	22	14

Table 5.10: Building 60M costs

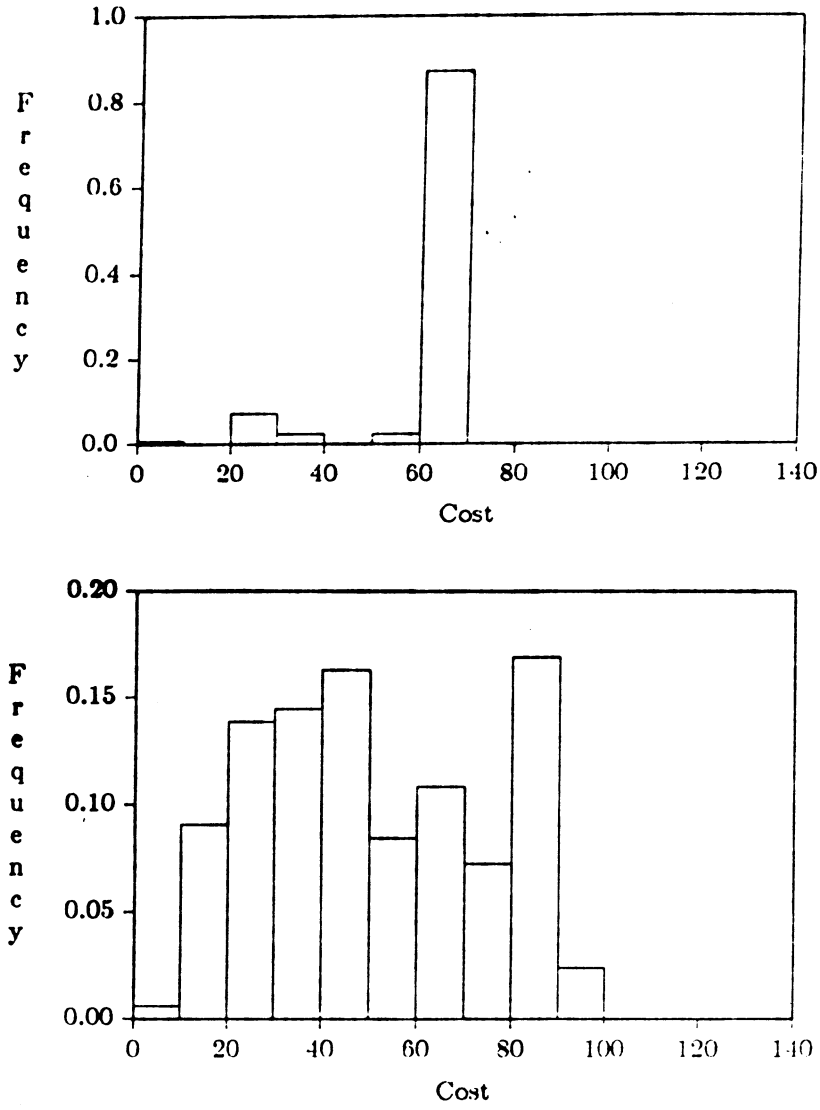


Figure 5.10: Cost histograms for strategies 3 and 4

5.2.4 The experiments

The cost table was conceptually enclosed in a black box, as in the experiment with the Unix commands, and run against our adaptive selector of strategies. Figure 5.11 shows the resulting probabilities (1000 experiments); convergence to strategy 3 is extremely fast despite the high variances and overlaps in the cost distributions. The occasions where strategy 3 is not best are few and spread widely apart; thus the adaptive selector is not able to detect and adapt to these quick changes.

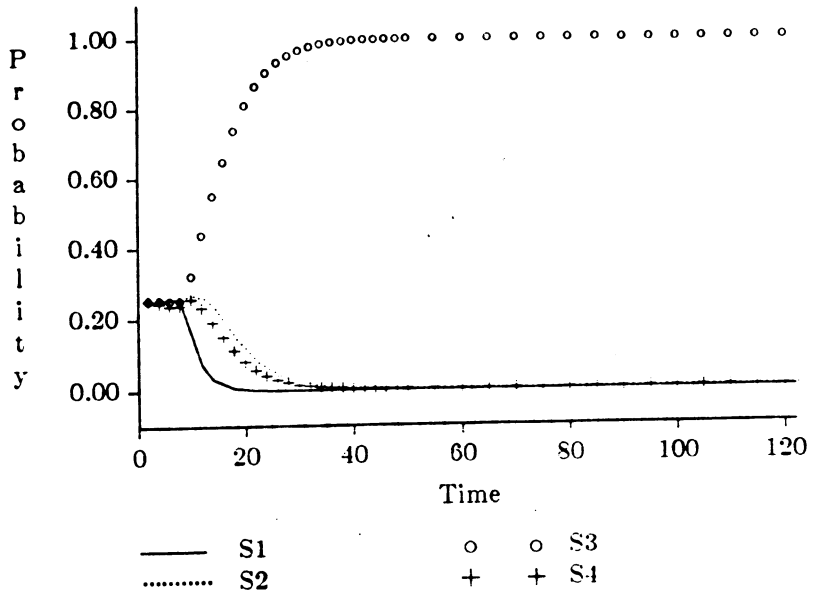


Figure 5.11: Probabilities, static conditions

To illustrate adaptability to change, let us assume that somebody decides to give a weekly seminar for the rest of the year in a room of the "60M" building. This is not at all unreasonable; in fact, the query load for the third day contains an example of a monthly reservation in another building. Thus, 48 consecutive *insert1* transactions would be issued with $c01 = 60M$. We spliced these imaginary transactions in the middle of the query load for day 2, starting at transaction number 30, and we ran the simulation/analytic models again to obtain costs. After that, Algorithm II for the adaptive selector was run with parameter α (sensitivity to change) equal to 0.8 and parameter *pmin* (minimum allowed probability) equal to 0.05. The results are shown on figure 5.12. We observe that the adaptive selector begins choosing strategies 2 and 4 together more frequently than strategy 3, then adapts back to strategy 3 when the 48 special transactions finish.

5.3 Summary

We have presented several experiments using two very different query loads. The costs used in the experiments were artificially, but realistically assigned; the costs themselves however, are irrelevant and we make no attempt to justify the cost models used. In fact, that is the point of the thesis: the adaptive selector chooses low-cost strategies regardless of the method by which the "cost" is determined. These experiments show that, under realistic conditions, the adaptive selectors are able to converge to the best strategy and to adapt to possible changes that can occur.

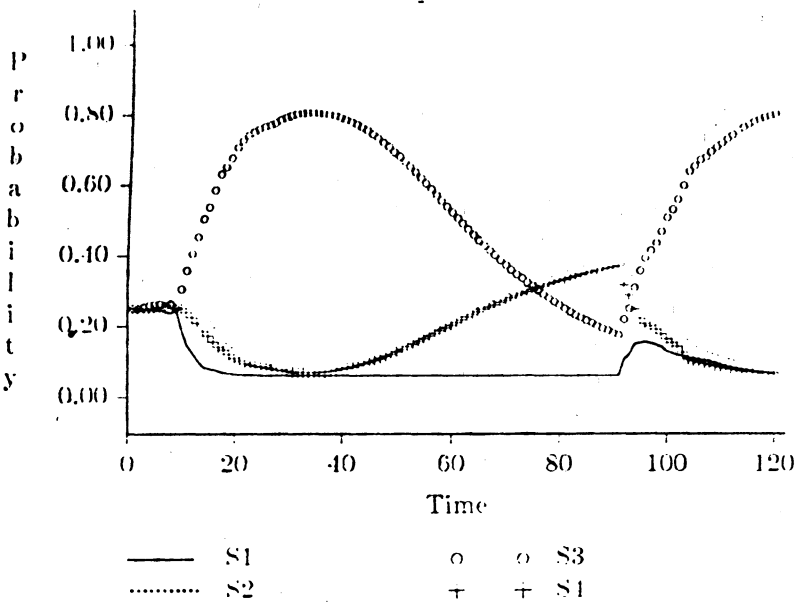


Figure 5.12: Probabilities, changing environment

Chapter 6

Query class partitioning

In chapter 2, we presented a general adaptive query processor consisting of a query classifier and several strategy selectors, one for each class of queries (Figure 2.3). The basis for classification is equivalence of the set of strategies applicable to each class.

Once this initial classification of queries has been made, it is possible to refine it by partitioning a class further into subclasses. This subdivision can take into account the costs fed back from the database as a result of query execution, and it can be modified adaptively.

The optimal partitioning for a given class would be the following: subclass number i consists of all the queries whose best strategy is s_i. In this ideal case, an adaptive strategy selector would not be needed for any subclass: determining the subclass identifies the corresponding best strategy. Obviously, the optimal partitioning cannot be known from the beginning, since initially we know nothing about execution costs of individual queries. But for some important cases, we can approximate this optimal partitioning adaptively using several automata; this chapter describes and analyzes the technique.

We stress that the algorithms that appear in this chapter are only an initial step towards solving this problem. The next chapter sketches further research directions.

6.1 Strategy selector with partitioning

Figure 2.3 of Chapter 2 shows the operation of an adaptive strategy selection system for several query classes, as a data flow diagram. The figure is repeated here (figure 6.1) for convenience. Suppose that each of these selectors utilizes a learning automaton of the type described in Chapter 4. Figure 6.2 shows the basic operation cycle of the selector. The automaton repeatedly chooses a strategy according to the probability vector p, sends the query for execution with this strategy, and updates its state M based on the cost of execution. The state of the automaton includes the vectors p, σ, n and the current averages $\bar{x}$ (Chapter 4, page 37).

Consider now a certain class. Suppose that a *partitioning* of the queries that belong to the class has been determined. Let Q be the set of all queries that belong

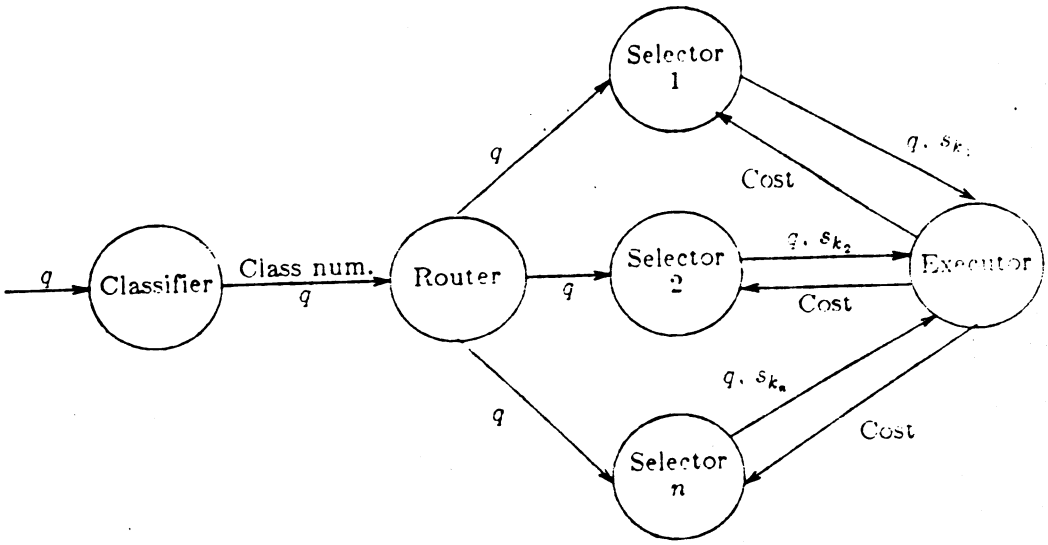


Figure 6.1: Query classification

initialize(M) (including probability vector p)
repeat forever:
  Get next query q
  $k := P(M)$ —apply policy P to choose a strategy s_k according to probability vector p
  $x := execute(q, k)$ —execute query with strategy s_k. Get feedback cost x
  $M := \tau(M, k, x)$ —update state of the automaton

Figure 6.2: Selector using learning automaton

to the class, and let m be the number of partitions (or subclasses). We now propose to have m automata, one for each subclass. Each automaton l is *guarded* by a boolean function f_l, where $f_l : Q \rightarrow \{true, false\}$. These functions test whether a query belongs to the corresponding subclass. Since the subclasses partition the class, for any $q \in Q$, exactly one of the functions is true. The operation of this set of m automata is now as follows; see also figure 6.3:

initialize(M) for all the m automata;
repeat forever:
  Get next query q;
  if $f_1(q) \rightarrow k := P(M(1)); x := execute(q,k); M(1) := \tau(M(1),k,x)$
  ▯ $f_2(q) \rightarrow k := P(M(2)); x := execute(q,k); M(2) := \tau(M(2),k,x)$
  ...
  ▯ $f_m(q) \rightarrow k := P(M(m)); x := execute(q,k); M(m) := \tau(M(m),k,x)$
  fi

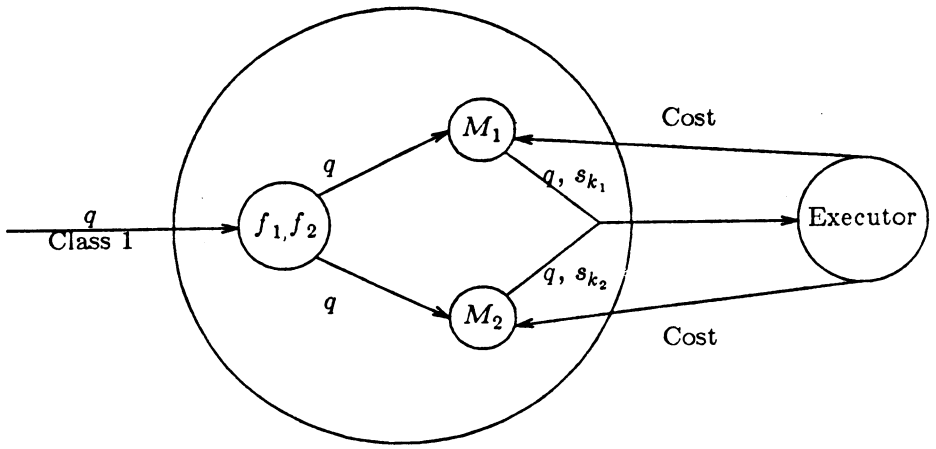


Figure 6.3: Selector with two partitions

Thus the query dictates which one of the m automata determines the strategy k to be executed.

A partitioning can be defined without explicitly listing all the queries in each subclass. As an example, suppose that all the queries in the class include a test for the value of an attribute α, and let $q.\alpha$ be the value being tested in query q. A partitioning of the queries into two subclasses could be specified as follows:

$$partitions = \{\{q \mid q.\alpha \leq a\}, \{q \mid q.\alpha > a\}\} \tag{6.1}$$

where a is some constant. The functions f_1 and f_2 would then be simple tests for the value of $q.\alpha$. Another simple way of defining the partitioning is via a hash function H from the set of all queries Q to the set $\{1 \ldots m\}$. Then $f_l(q) = (H(q) == l)$. Of course, the partitioning that consists of one subclass only is trivially specified: $f_1(q) = true$.

An *optimal partitioning* for a set of queries $\hat{Q} \subset Q$ is a set of K subclasses and the corresponding functions $f_1 \ldots f_K$ such that, for all $1 \leq l \leq K$ and all $q \in \hat{Q}$, $f_l(q)$ is true if and only if s_l is the strategy of minimum cost for q. Note that some subclasses may be empty, when the corresponding strategies are never the best for any of the queries. In general, the optimal partitioning can change over time, as the costs of query execution change.

We would like to find the optimal partitioning for the set of queries seen so far and to update it incrementally as new queries come in. We are also concerned with efficient ways of describing each partition —more on that later.

Once we have an optimal partitioning for the whole set Q, any query would likely be executed using its best strategy, as we show next:

Theorem 1 *Let $f_1 \ldots f_K$ be the classifying functions corresponding to the optimal partitioning for $\hat{Q} \subset Q$. Assume that each query $q \in \hat{Q}$ is posed with a constant frequency > 0 and assume that the learning parameter λ_l for each automaton is sufficiently small. Further assume that the environment does not change (constant database state). Then, as $t \rightarrow \infty$, each automaton l in the algorithm above that corresponds to a non-empty subclass will have converged to strategy s_l.*[1]

Proof. Let $\hat{Q}_l \subset \hat{Q}$ be the set of queries belonging to subclass l. By construction, $f_l(q')$ is true for all $q' \in \hat{Q}_l$ Since all these queries are posed with frequencies > 0, automaton l will tend to receive an infinite number of queries as $t \rightarrow \infty$. Now let $C : \hat{Q}_l \times S \rightarrow \Re$ be the underlying cost function that gives the cost of execution of queries in $\hat{Q}_l$, and consider the restriction of C to strategy s_l, C_l. Since $C(q', l)$ is the smallest cost in each K-tuple $< C(q', 1), C(q', 2), \ldots C(q', K) >$ for all $q' \in \hat{Q}_l$, then the mean of the cost distribution defined by C_l will be the smallest of the means defined by the other restrictions C_i for $i \neq l$. Therefore, given a small enough λ_l, automaton l will converge to s_l[85].

Corollary 1 *If all queries posed belong to the set $\hat{Q}$, then as $t \rightarrow \infty$, query processing cost will asymptotically approach the minimum possible.*

The converse of the theorem does not hold: If automaton l converges to s_l, for $l = 1 \ldots K$ the corresponding partitioning need not be optimal. As an example, consider the following four queries assumed to be posed with equal frequencies and their costs on two strategies:

query	Cost	
	s_1	s_2
q1	10	20
q2	15	10
q3	20	10
q4	10	15

[1] By this we mean that the probability associated with s_l is $1 - \epsilon$ for some suitable choice of ϵ.

If the queries are partitioned as $\{ \{q1, q2\}, \{q3, q4\} \}$, the first partition will converge to strategy 1 with an average cost of 12.5, and the second partition will converge to strategy 2 with the same cost average. Since all queries are equally frequent, the overall cost will be 12.5. The optimal partitioning, however, is $\{ \{q1, q4\}, \{q2, q3\} \}$ with an average cost of 10. Nevertheless, the first partitioning above is still better than no partitioning at all, which would achieve an average cost of 13.7. We will later prove that this is true in general: any partitioning in which at least one partition converges to a strategy different from the converging strategy of the full class, improves upon the class as a whole.

6.2 Gradual improvement of the partitioning

To determine and maintain the optimal partitioning adaptively is a very complex problem: recall that when a query is executed, all we get back is the cost of execution on the strategy tried; we don't even know which would have been the best strategy for that query. From a practical point of view, however, we might be satisfied with a sub-optimal partitioning that works well for the kinds of queries that users are asking, which is typically a small subset of all possible queries.

In this section, we present a general approach to derive good partitionings adaptively. We will also show under which conditions the approach obtains the optimal partitioning for all queries.

We first describe the process informally and illustrate it with an example from the System 2000 database of chapter 5. Then we will generalize and analyze the procedure.

6.2.1 Description of the approach

The approach considers several candidate partitionings in parallel. One of them, the *executive partitioning*, includes several automata that are in charge of deciding the strategy to be executed. The executive partitioning is the best one known so far. The other partitionings are called *trainees* and also include several automata each. They have no voice on taking decisions, but their automata are nevertheless updated each time step. Whenever one of the trainees improves on the current executive, it replaces the executive. The process starts with the simplest executive partitioning (one subclass only) and no trainees. Trainees are then generated in such a way that eventually, all possible partitionings are evaluated. A special module (the *president*) is responsible for hiring new trainees, examining periodically the performance of all involved and making the promotions. Thus, assuming a large enough memory and long enough usage, the executive gradually improves until it becomes optimal. The *order* in which the trainee partitionings are generated is critical for the efficiency of this process; we will say more about this later.

A crucial requirement of this approach is that no strategy is ever starved, that is, its probability of execution should not be reduced to 0. Therefore the executive automata use Algorithm II of chapter 4 (page 44), in which a minimum allowed probability value is specified.

6.2.2 An example

We will illustrate the process with the transaction *insert1* of the System 2000 database described in chapter 5. We use a particular procedure to generate the trainees that happens to work well in this case; however we will later argue that the principles involved do have some merit for common database situations. The procedure assumes that queries are classified by the attribute names that appear in the query. We further assume a constant environment. See Table 5.8 of chapter 5 for a typical transaction load.

Initially the executive consists of one subclass; therefore its only automaton is guarded by the condition *true*:

```
procedure executive()
repeat forever
  Get query q
  if true → k := P(M(1)); x := execute(q,k); M(1) := τ(M(1),k,x)
  fi
  train(q, k, x)
end executive
```

The president initially hires three trainees. One discriminates queries on the basis of the value of attribute $c01$, the other on $c21$, and the last one on $c61$. Since all these attributes are character strings, a first approach discriminates on the middle value, let us say "M":

```
procedure train(q, k, x)
local M...
 —first trainee:
 if q.c01 ≤ "M" → M(1,1) := τ(M(1,1),k,x)
 ▯ q.c01 > "M" → M(1,2) := τ(M(1,2),k,x)
 fi
 —second trainee:
 if q.c21 ≤ "M" → M(2,1) := τ(M(2,1),k,x)
 ▯ q.c21 > "M" → M(2,2) := τ(M(2,2),k,x)
 fi
 —third trainee:
 if q.c61 ≤ "M" → M(3,1) := τ(M(3,1),k,x)
 ▯ q.c61 > "M" → M(3,2) := τ(M(3,2),k,x)
 fi
 updtActualLimits(q)
end train;
```

Note that the executive passes the execution cost to the three trainees simultaneously; each trainee's only job is to update its state.

After a while, the president stops everybody to evaluate convergences. They will be as follows:

- The executive has converged to strategy 3, the best average strategy for the class.
- Trainee (1,1) has also converged to 3: for all the queries posed, $q.c01 \leq$ "M" (see Table 5.8) and thus it has received all the queries. Trainee (1,2) has not received any.
- Likewise, (2,2) has converged to 3: Most of the values of *c21* are numeric, and therefore $>$ "M". (2,1) has only received a few queries and has not had time to converge to any strategy. The situation with Trainee 3 is similar.

The president decides that these trainees are no good since they *either have not converged or have converged to the same strategies as the current executive.*

Next, the president attempts a change of discriminator. The module *updtActualLimits* is in charge of maintaining the maximum and minimum values of attributes requested so far in queries. For our example, let us assume that these values are $q.c01 \in \{$"60M" ... "700U" $\}$, $q.c21 \in \{$"01" ... "20" $\}$ and $q.c61 \in \{$"A" ... "$MEZZ$" $\}$ Middle points in these ranges are a reasonable next guess. The result is:

```
procedure train(q, k, x)
local M...
 —first trainee:
 if q.c01 ≤ "65  " → M(1,1) := τ(M(1,1), k, x)
 ▯ q.c01 > "65  " → M(1,2) := τ(M(1,2), k, x)
 fi
 —second trainee:
 if q.c21 ≤ "10  " → M(2,1) := τ(M(2,1), k, x)
 ▯ q.c21 > "10  " → M(2,2) := τ(M(2,2), k, x)
 fi
 —third trainee:
 if q.c61 ≤ "F" → M(3,1) := τ(M(3,1), k, x)
 ▯ q.c61 > "F" → M(3,2) := τ(M(3,2), k, x)
 updActualLimits(q)
 fi
end train;
```

This time, by the discussion in section 5.2.3 of the previous chapter, (1,1) will converge to strategy 2, and (1,2) to strategy 3. The current executive continues to prefer strategy 3. We can see that trainee1 is better that the executive. If trainee1 is promoted, all queries where $q.c01 \leq$ "65 " would execute strategy 2, which is better for them than strategy 3. On the other hand trainees 2 and 3 provide no improvement: both automata in each of trainee2 and trainee3 converge to strategy 3. Thus the president decides to fire the executive and promote trainee1, *because trainee1 is able to converge to* more *different strategies than the current executive*:

```
procedure executive() —new
repeat forever
  Get query
  if q.c01 ≤ "65   " : k := P(M(1)); x := execute(q,k); M(1) := τ(M(1),k,x)
  [] q.c01 > "65   " : k := P(M(2)); x := execute(q,k); M(2) := τ(M(2),k,x)
  fi
train(q,k,x)
end executive
```

In this particular example, this partitioning happens to be very close to the optimal one *for the specific queries that occur.*

6.2.3 Prescription for the president module

The high-level pseudo-code in Figure 6.4 specifies more precisely the behavior of the president module. Basically, the president goes through a cycle of hiring trainees, letting them work, evaluating their performance and making promotions. At all times, the president arranges that each automata in the current executive converges to a different strategy. If m is the number of partitions in the current executive, a trainee can be promoted only if its automata converge to m or more different strategies. Ties in the number of converging strategies with the executive or other trainees are resolved by comparing the performances of the partitionings involved; the performance measure of a partitioning is specified in the next section.

6.2.4 The performance of a partitioning

The performance measure is the expected query processing cost of the partitioning if it were an executive.

Let N be the total number of queries received by the partitioning, L be the number of subclasses in the partitioning[2], $\bar{x}_{l,k}$ be the average cost for automaton l and strategy k, $p_{l,k}$ the corresponding probability, and $n_{l,k}$ the number of queries that have been received by l *and* executed on strategy k, for all $k = 1 \ldots K$ and all $l = 1 \ldots L$. The performance is then:

$$\mu = \left(\sum_{l} \sum_{k} n_{l,k} \times p_{l,k} \times \bar{x}_{l,k} \right) / N$$

The summation over k of $p_{l,k} \times \bar{x}_{l,k}$ is the familiar expected reward of the l'th automaton. These rewards are weighted by the proportion of queries received, $n_{l,k}/N$. We are using the running averages $\bar{x}_{l,k}$ as the best estimates that we have of the real mean of the marginal distribution of costs for strategy k, partition l. If the corresponding $n_{l,k}$ is small, these estimates may not be very accurate, but the weight $n_{l,k}/N$ attached to them will also be small.

We state without proof the following

[2]note that $L \leq K$ if the partitioning being evaluated is the executive, but may be greater than K if it is a trainee

procedure president()
Initialize executive X with one automaton; $f_1 = true$
Repeat until a BREAK-LOOP is executed:

Call the *recruiter* several times to generate T, the next set of trainees. Each call to the recruiter produces a new trainee. The number of trainees generated depends on the amount of memory available for this query class, and the size of the current trainees generated.

If $T = \phi$ then BREAK-LOOP (current executive cannot be further improved)

Let system run for a time proportional to the largest number of partitions in any trainee (This ensures that trainees are given a good chance of convergence)

Let c (for *candidates*) be the set of trainees that converge to m or more different strategies.

If $c \neq \phi$,

Compare performance (expected query processing cost) of the executive and the candidates. Choose w (for *winner*), the partitioning of best performance. In case of ties among the trainees, pick randomly. In case of ties with the executive, pick the executive.

if $w \neq X$ then —promote:

"Or together" the guards of automata within w that converge to the same strategy and simplify the resulting expressions, thus obtaining a new set of guards.

Replace executive by the winner trainee, having this new set of guards.

end loop
end president.

Figure 6.4: The president module

Theorem 2 *After a sufficient number of queries have been received, the performance of the optimal partitioning for the set of all queries is the minimum among the performances of all possible partitionings.*

This follows from the fact that, by definition of optimal partitioning, every query will be executed with its best strategy.

6.2.5 Correctness argument

Assuming that the recruiter module eventually generates all partitionings, and that there is enough memory available, the optimal partitioning will eventually be generated. By theorem 2, its performance will be smaller than the one of the current executive, and it will be promoted. Since no other partitioning generated later will have smaller performance, the optimal executive will not get fired.

The president attempts to speed up the search of the optimal by using several heuristics. First, the number of partitions on the executive increases monotonically, since trainees become candidates for promotion only if they converge to the same or a greater number of strategies than the executive. This will eventually produce executives with K partitions. Second, the "Or together" step of the algorithm ensures that each automaton in the promoted trainee converges to a different strategy; in theorem 1 we proved that this is a characteristic of the optimal partitioning. Third, the performance evaluation step ensures that μ for the executive decreases monotonically also.

Note that the correctness of the president is independent of the order of generation of trainees. The *efficiency* of the process is affected by this order, as we will show next.

6.2.6 The recruiter module

Generating the trainees in the best possible way is an interesting problem for further research. Here we will only list some desirable and often conflicting features of the generating procedure and then suggest a candidate procedure that satisfies some of these requirements:

- Trainees that have a high probability of improving on the current executive should be tried out. Thus the recruiter might look at the current executive, borrow some of its partitions and attempt to improve on others. However, doing *only* this may prevent giving a chance to very different trainees: the familiar explore-exploit conflict.

- Trainees should be generated in increasing order of complexity. This ensures that if simple partitionings turn out to be good, they are tried as soon as possible. There are two aspects of complexity: execution time and size. Execution time of a trainee is affected by the time of evaluation of the guards. The size of a trainee depends on the number of partitions as well as the size of the guards.

As usual, sometimes there is a tradeoff between the two aspects of complexity. A partition that can be described by a simple boolean function (see equation 6.1) is both compact and quick to evaluate. A partition that can only be described by an exhaustive listing of its queries occupies a lot of storage, but may also be evaluated quickly, via hashing for instance.

Trainee size also affects the time required to find good executives. Trainees are not tried one by one, but in batches that are updated in parallel. A large number of small trainees can be evaluated at the same time.

Balancing out all these aspects, *increasing size* is probably the best criterion for trainee generation.

- The partitionings generated should be good for the kind of queries being posed, not necessarily for the set of all possible queries. This implies keeping track of some aspects of the queries, such as ranges of attribute values.
- To guarantee absolute optimality given enough time and memory, the generating procedure must be able to produce all possible partitionings. Nevertheless, in common situations with a relatively small amount of memory, it may not be practical to continue the search until the optimum is found.

A good recruiter

We suspect that the simple generation procedure that was sketched for *insert1* above is likely to work well for most databases. There are several cases where the identification of the best strategy depends on the values of one or a few of the attributes in the query. Here are some other of these cases:

1. Piatetsky-Shapiro and Connel [70] report an example of a query to the New York stock exchange database. Each record contains, among other things, the name of a company, the year and amount of sales. The query is:

   LIST *company* WHERE *sales* $> s$ AND *year* $= y$

   There is an index for each attribute. The two strategies consist of choosing one of the indexes to access all the records that satisfy one condition and checking on those records for the other condition. Assuming that cost is proportional to the number of records that have to be retrieved, the *year* index turns out to be better for small values of s, and the *sales* index for large values of s. Thus a discriminator based on the value of s forms a good partitioning for selecting strategies.

2. Consider searching on an ordered linear list, and let three strategies be: search from the left, search from the right, or do a binary search. The first is better for small key values, the second for large keys, the third for the rest. Note that searches close to the extremes of lists might be fairly common.

Figure 6.5 shows the pseudocode of a recruiting procedure based on these ideas. The recruiter is written as a coroutine of the president module. At each presidential

coroutine Recruiter()
BEGIN:
For $n = 2$ to ∞ (number of subdivisions of executive partition)
For every partition p of the executive. Let g be its guard.
Let A be the set of attributes *not* being tested in g
If $A = \phi$ continue for loop above.
For every $a \in A$

Let *amin to amax* be the range of values of a that have been observed so far in queries received *by partition p*. Generate R, a set of n range tests on the value of a, equally spaced between *amin* and *amax*.

Generate trainee T'. Partitions of T' are the same as those in the executive except that partition g is replaced by n partitions $p'_1 \ldots p'_n$, where the guard of p'_i is "*g and* $R[i]$"
Coroutine return (T')
If executive has changed, go to BEGIN.
End 3 loops.
end coroutine

Figure 6.5: The recruiter module

cycle, the recruiter is called enough times to fill up the memory allocated to the class. The recruiter attempts a systematic refinement of the current executive partitions. The guards produced are boolean functions over ranges of attribute values. Trainees with a small number of partitions and simple boolean functions are generated first.

Explanation

Initially, the executive has only one partition with guard *true*; therefore *g and* $R[i]$ above yields $R[i]$, and the recruiter starts producing trainees that discriminate on two intervals of each attribute, then three intervals and so on. Suppose that the automata of a trainee that tests for attribute $a5$ with the following guards, converges to strategies 1, 3 and 1 respectively:

$$\begin{aligned} a5 &< 5 \\ 5 \leq a5 &< 10 \\ 10 \leq a5 & \end{aligned}$$

We will prove below that this trainee must have a better performance than the 1-partition executive. Thus it is promoted by the president, with the following guards:

$$\begin{aligned} (a5 < 5 \textit{ or } a5 &\geq 10) \\ 5 \leq a5 &< 10 \end{aligned}$$

At this moment the recruiter detects a change in the executive, and starts generating trainees that are refinements of the executive partitions. First the first guard is combined with two intervals of the first attribute, yielding:

$$\begin{aligned} (a5 < 5 \textit{ or } a5 \geq 10) \textit{ and } a1 &< 20(\text{say}) \\ (a5 < 5 \textit{ or } a5 \geq 10) \textit{ and } a1 &\geq 20 \\ 5 \leq a5 &< 10 \end{aligned}$$

If this results in different strategies for the first two partitions, again this trainee will be promoted. Else the first executive guard will be combined with two subdivisions of other attributes, then the second executive guard will get combined and so on. Trainees will continue to be generated until they are so big that not even one will fit in memory; then the president will conclude that no further improvement is possible —(with this recruiter).

This process of executive refinement works because of the following theorem:

Theorem 3 *Let* $P = p_1 \ldots p_m$ *be the set of executive partitions. By construction, each corresponding automaton converges to a different strategy; let these strategies be* $s_1 \ldots s_m$. *Let* p_j *be any of the partitions, and consider the trainee* T *with partitions identical to* P *except that* p_j *is replaced by* $p_{j,1} \ldots p_{j,l}$, *where the* $p_{j,*}$ *are themselves partitions of* p_j. *Let* $s_{j,1} \ldots s_{j,l}$ *be the strategies to which each of these subdivisions converge. Then, if at least one of the* $s_{j,*}$ *is different from* s_j, *the performance measure of* T *will be smaller than the one of* P

Proof. We only have to consider whether the performance of the subdivided p_j improves upon p_j, since all the other partitions are the same. Suppose we group the $p_{j,i}$'s that converge to the same strategy together, arriving at a new set of partitions $p'_1 \ldots p'_n$, and let $x_{p,q}$ be the average cost of queries in p'_p on strategy s_q,

and f_p be the combined frequency of queries in p'_p. Further let r be any of the partitions, converging say to s_x, and t be any other partition. We can show that the sub-partitioning that consists of r and t improves upon having both combined together. Indeed, the performance of r and t would be $f_r \times x_{r,x} + f_t \times x_{t,u}$, where s_u is the strategy to which t converges. The performance of their combination would be $f_r \times x_{r,x} + f_t \times x_{t,j}$. But since s_u is best for t, then $x_{t,u} \leq x_{t,x}$, therefore the partitioning of r and t improve upon their combination. A similar argument can be made considering r and each of the other partitions, not only t.

6.3 Review of related work

The area of pattern recognition analyzes problems with some similarities to the one we have here, yet sufficiently different to justify a new approach such as the one we have presented.

A crucial assumption of the work in pattern recognition is that *all the information relevant for classification is available or can be obtained from either the object to be classified, or a human operator.* In contrast, in our problem, the relevant information is clearly *all* K strategy costs; but only one of these costs is available —the cost of execution in the strategy chosen by the selector.

In the *statistical* school of pattern recognition[38], objects are usually represented by means of a *measurement vector* which may or may not be subject to measurement error. The vector is then processed by a *feature extractor* which reduces it to a *feature vector* with the features relevant for processing by a *classifier.* In structural pattern recognition[43], an object is represented as a collection of elements and their relationships; the object may then be classified by *grammars.* In both approaches, we can see that the object representation itself includes information relevant for classification. In a related approach, a *trainable classifier*[82] is presented with a series of patterns, and the class identification given by a human operator. The classifier must then learn the features of the feature vector that are relevant for classification.

Closer to our problem are *clustering approaches*, especially *adaptive clustering* [92,39]. Here, objects are clustered according to some measure of distance; class boundaries are evolved adaptively. Again, the object representation contains all the information needed by the clustering procedure.

Any of these methods can probably be used for query partitioning given a complete query-cost table such as Table 5.8. But for the on-line strategy selector that has available only one of the K costs at a time, there appears to be no alternative other than evolving good classifiers with a guided trial-and-error procedure as we have suggested.

6.4 Summary

We have defined the query partitioning problem and presented and analyzed a general procedure for administering the work (the president module), and a particular

module to generate good partitionings.

More sophisticated approaches are possible in the search for the best classifier. Our intent here has been to show that a simple procedure has good chances to work well in practice. Best of all, while the search for good trainee partitionings takes place, the executive automata take care of using the best average strategy for the query class.

Chapter 7

Conclusions and further research

This chapter summarizes the thesis and points out future research directions. In the following section, we present a list of the main contributions of this work. The principles of adaptive selection that we have developed are extremely general, and indeed their application need not be restricted to database systems. Section 7.2 mentions other potential areas of application. Finally in section 7.3 we discuss future directions of this research.

7.1 Summary of contributions

1. This is a novel application of adaptive systems to the selection of query processing strategies. Previously, adaptive approaches to databases have been used mainly for the incremental modification of storage structures [83,49,92]. Dynamic modification of strategies while a query is executed has been tried before with some success for particular cases [27,76]. Yu et. all [94] fed back the last execution cost in order to adapt a cost formula. No system known to us uses the *actual cost history* of query execution as a means of improving future choices of strategies.

   (a) We presented in chapter 2 a general framework for adaptive selectors, based on previous work on general adaptive systems[53]. This framework permits a clear visualization of the possible kinds of adaptive strategy selectors and will be very useful for further work in this area.

   (b) We identified two main situations where the traditional approach to strategy selection based on analytic cost models is inadequate.

   - The case where cost models are totally unavailable has not been considered before; for this case, the adaptive approach is the only solution presented so far. Several realistic situations in which no cost models are available were presented in chapter 1.
   - The case where analytic cost models are unreliable because of the complexity of the environment has been addressed before and several solutions have been proposed [25,24,36,55,70]. As we mentioned

in chapter 1, all these solutions have limitations and still require a number of assumptions about the database and query load environment. These approaches may have better adaptability than our approach to certain sudden changes in the database structure, for instance, the creation of a new index. Other changes, however, may go undetected, including cost changes caused by variations in the query load characteristics, changes in database concurrent usage at different hours during the day and so on. Adaptive selection, on the other hand, is almost assumptions–free and does adapt to all changes in the environment.

(c) The adaptive approaches are extremely portable across database management systems, precisely because they do not need to make assumptions about database structures and query loads. In contrast, other methods of strategy selection are closely tied to particular environments.

2. In chapter 3, we mapped the optimal policy design problem to similar problems in statistics (Bandit problems) and decision theory, and presented examples of solutions to our problem based on dynamic programming. We also presented a new analysis of the running time of the resulting algorithms.

3. We identified a number of possibilities for the design of heuristic adaptive plans of strategy selection, and chose one based on learning automata. This is the first time that these automata have been used for query optimization. We extended an effective automata algorithm [85] to handle arbitrarily evolving environments, and presented comprehensive simulations to illustrate and explain its behavior. We showed that this new approach to strategy selection works well and adapts to change under real query loads of very diverse natures.

4. We formally stated the query class partitioning problem and proposed a general algorithm for its solution. Part of this algorithm is a procedure to generate candidate partitions; we proposed a simple form to do this and showed that it works well in several cases.

7.2 The application of adaptive selection to other areas

The adaptive approach that we have presented can be conveniently used for any class of problems that has the following characteristics:

1. Problem instances have to be solved repeatedly varying the input.

2. The solution can be obtained using any of several strategies.

3. The cost of solving a problem instance using each strategy depends on (partially) unknown environmental factors.

For problem classes with these characteristics, the adaptive approach selects over time the best average solution, and is able to adapt its selection to changes in the environment. Consider the following examples:

- Several data structures can be searched in more than one way. Consider for instance an ordered list of size N. Both worst case and average case analysis show that binary search is better than linear search under a number of assumptions. In particular, the average case analysis usually assumes that keys searched are uniformly distributed. If, however, keys searched have some unpredictable locality of reference, one of the following K strategies may prove superior: Suppose that the list is circular, and start a linear search at position $(k-1) \times N/K$, for $k = 1 \ldots K$. An adaptive selector can choose over time among all these strategies and the binary search strategy as well.

  Locality of reference is extremely common. On almost any file ordered by date, queries that refer to recent dates are likely to be more frequent. On a file of student grades ordered by grade, a researcher may become interested in students with very low grades and ask several queries about them. A few days later another researcher decides to query the best students.

- Char et al. [21], report that in numerical algebra, there are many cases where several algorithms are available to solve a given problem. One example is the greatest common divisor of polynomials, where at least three algorithms are available; the authors list several other cases that occur in the Maple[22] symbolic algebra package. On many occasions, which algorithm is better for a given problem instance is not clear-cut. The Maple system uses several heuristics and then loads to memory the required algorithm. Some of the heuristics depend on hardware characteristics of the machine, such as the relative speed of certain instructions. An adaptive selector could be made to choose over time the best algorithm, *for the kinds of problems that users are currently requesting*; a measure of cost that combines memory usage and execution time would likely be adequate in this case.

7.3 Further research

There are several directions in which this research can be extended. In the thesis, we have analyzed the case where the only *signal* available from the environment is the cost of query execution. We first list a number of issues remaining to be solved in this case. Then we mention the possibility and implications of receiving more signals. Finally we discuss potential interactions of this approach with the traditional approach, with adaptive modification of data structures, and with the data base administrator.

- The optimal policy design problem can be extended to include the possibility of change of the underlying distributions while the policy is being applied.

- The algorithms that we have studied require a number of parameters that have to be set up by the database designer. These include the learning parameter λ of Algorithm I (page 37), and the sensitivity to change α of Algorithm II (page 44). In principle, these parameters could be set and changed adaptively by the system itself; more research is needed however into this problem.

- Analytic results are needed on convergence time of our algorithms under reasonable assumptions.

- We have considered heuristic policies based on a particular learning automata algorithm. As we mentioned in chapter 4, other automata algorithms might also be used [20], as well as algorithms not based on automata [74,8,28,52,41].

  Some of these approaches, though, might require additional assumptions about the environment and thus reduce portability. Further, they may be difficult or impossible to modify for changing environments.

- In chapter 6 we gave one simple way of generating the trainee partitions that works well for several cases. More research is needed to investigate the applicability of this generating procedure and to derive analytical measures of its efficiency. A promising approach in the search of good candidate partitions is genetic algorithms [1] which have been shown to be particularly effective in searching multi-modal spaces.

- The costs of different strategies are not independent of each other. All strategies are constrained by the fact that they must produce the same answer from the database, accessing similar data to arrive at the answer. This creates complex dependencies between costs; for instance, consider two strategies that roughly go "down the hierarchy" when answering a query in a hierarchical database. Typically, the cost of one of the strategies is high when the cost of the other is also high, though the dependency might not be linear. Nevertheless, a clever system might try to detect the form of the dependency, and therefore predict the cost of one strategy when *the other* strategy is executed; this may decrease convergence time. This approach will likely need further assumptions about the nature of the possible dependencies or the type of strategies.

- Depending on the definition of the query classes, the costs for one class may be related to those of another class. As an example, consider that one class comprises the queries that join relations A and B, and another the queries that join relations A, B and C. Clearly the costs would be related, and it might be possible to affect the probability vector for the second class when a query of the first class is executed. To implement this, feedback signals from intermediate steps of query executions would have to be fed back.

- When the total number of *possible* classes is too large, it may be wasteful to keep an automaton for every class (and many more if the partitioning approach of chapter 6 is used); most of these automata would stay idle forever. What we need is a scheme to create and delete automata according to which are the currently *most important classes*, the ones that are consuming most resources

from the database. The number of the most important classes is typically small [58,75]. Queries that do not belong to the most important classes would be answered using arbitrary strategies.

The general framework of chapter 2 specifies a set of *signals* coming from the environment when a query is executed. We have considered the case where the only signal is a single measure of *cost* of execution; but clearly other signals are possible, such as intermediate or final selectivities of attributes, record counts and so on. Making suitable assumptions, these other signals may help to predict the cost of several strategies when only one of them was executed, decreasing convergence time.

It might be possible to combine the traditional approach to strategy selection with the adaptive approach. The adaptive approaches become slow to adapt to change when the number of strategies is large. A rough analytic model might help to eliminate the strategies that are evidently very expensive, leaving to the adaptive selectors to decide over time among the remaining strategies.

Even more interesting is the possibility of allowing analytic models to consider feedback signals as input to improve the models adaptively. How should an analytic model change when the cost that it predicts turns out to be wrong? See Yu et al. [94] for some initial steps in this direction.

There is a limit to the amount and kind of information that a given computer system may learn by itself. Beyond this limit, it becomes practical for the system to be able to accept advice from humans. The database administrator may sometimes know which are the most important classes (using perhaps a measure of *importance* that goes beyond mere cost), or which strategies are obviously very bad for certain query classes and which should be ignored after a database reorganization and so on. Practical learning systems, ours included, must therefore incorporate ways for *optionally* accepting advise from people at appropriate points.

There are very interesting tradeoffs to study between adaptive selection of execution strategies and the adaptive self-modification of data structures, such as *move to front* lists[73], self-modifying trees[84] and so on. Both approaches have as primary objective the adaptive adjustment of *something* as a result of actual queries posed by users. Adaptive selection is by its nature more general, since it is independent of underlying data structures. But when adaptive selection is applied to, say, a move to front list, it is not clear where it is most profitable to apply resources: to the modification of the list, or to the adaptive selection of the best average strategies that search the same list, or to the adaptive subdivision of queries into subclasses, or a combination of everything. Perhaps new algorithms can be devised that can optimally combine all the approaches.

Appendix A

Program to find the optimal policy

The following Maple program was described in section 3.2.4. The program is fully commented and should be easy to follow. Here are a few notes about the language Maple for readers unfamiliar with this powerful language. Interested readers should refer to the manual [22] for more details.

1. Maple is an untyped expression language. A double quote symbol (") refers to the value of the last expression:
   5+3:
   a := a + ": #adds 8 to variable a.

2. p := *proc*($a1, a2, a3$) : *statements ... end* : assigns to p a procedure with formal arguments $a1, a2, a3$. The procedure can then be called as in $p(5, 4, 3)$. Procedures can return objects of any type. By default, a procedure returns the value of the last expression computed.

3. $l := [2, 3, 4]$ assigns to l a list of three elements. Individual list elements can be selected by indexing: l[1] equals 2. Lists can be nested.

4. All arrays are associative. Index types are arbitrary: d [[a,b,c]]:=[5] assigns a 1-element list as the value of the d-array element indexed by a three-element list.

5. *map*(p, l) where p is a procedure and l is a list, returns the list that results when applying p to each element of l in turn.

6. The *option remember* declaration within a procedure body establishes an invisible associative array inside the procedure that relates argument values used in procedure calls with the value returned. If the procedure is called again with the same argument values, the procedure accesses the table and returns the value, rather than repeating the computation. This feature alone makes possible a clear recursive expression of dynamic programming problems that runs in the same amount of time as iterative versions.

```
#                    FIND OPTIMAL POLICY
#note: time == "stage"
#      "at time t" == after t-1 decisions have already been taken, and we
#                     are considering the (t)st decision.
#      "state": whatever is kept of the history of observations
#      h       : time horizon, or the number of decisions that will
#                be taken
#      K       : number of strategies
#

#
# the basic optimality equation for all dynamic programming decision problems
# with a fixed number (K) of decisions at each stage t.
# S is the current state, just before taking the decision.
#
V:= proc(t,S) #value of optimal policy at time t, state S
              #side effect: stores in dstar[t,S] the optimal decision.
    local mmin, which, k;
    options remember;

    mmin:=99999; which:=1;
    for k to K do
    VA(k,t,S);   #value of decision k
        if " <= mmin then which:=k; mmin:="" fi od;
    dstar[t,S] := which;
    mmin
end:

# 1. Now assume values in strategy k are integers between xmin[k]and xmax[k]

VA:= proc(k,t,S) #value of deciding strategy k at time t, followed by
                 #optimal decisions thereafter.
     local zum, x, p;

     zum := 0;
     for x from xmin[k] to xmax[k] do
         p := pr(x,k,S);                  #probability of x on k given S
         nextS(x, k, S);                  #S after getting an x from strat. k
 if t=h then 0 else V(t+1,") fi;  #"...optimaly thereafter..."
         zum := zum + p*(x + ") od
end:

# 2. Assume probability is computed from Posterior Bayesian Distribution

pr:= proc(x,k,S) #probability of x on k given S

     Bayes(prior[k],S[k],k); #posterior distr on arm k
     p(x,k,")
end:

# 3. Assume the distribution family on arm k is given by a discrete set of
# sizF[k] discrete distributions. Each distribution consists of N[k] values, where
# value number l is the probability of obtaining xmin[k]+l-1, l=1..N[k]. So the
# distribution family is a list of sizF[k] N[k]-tuples, called F[k].
# A prior or posterior on this distribution family is given by a vector of
# sizF[k] probabilities, where the Lth element represents the current subjective
# probability that alternative number L is the real one.
```

```
p:=proc(x, k, d) #probability of x given distribution of alternatives d for arm k
   local j, Fk, i, zum;

   j:=x-xmin[k]+1; Fk:= F[k];
   zum:=0; for i to sizF[k] do zum:=zum+d[i]*Fk[i][j] od;
end:

# 4. Assume that the state for arm k is kept as a list of N[k] counters,
# one each for each possible value on the arm

Bayes:=proc(prior,nobs,k) #posterior on arm k given its prior and number of
                          #observations nobs.
   local i, j, post, wlklhd, Fk, Fki, scalefac, prod;
   options remember;

   Fk:=F[k];
   scalefac:=0.0;
   wlklhd:=array(1..sizF[k]); post:= array(1..sizF[k]);
   for i to sizF[k] do;
       prod:=prior[i];
       Fki:=Fk[i];
       for j to N[k] do;
           if nobs[j] <> 0 then
              Fki[j];
              if " = 0 then prod:=0;break else prod:=prod * (" ^nobs[j]) fi
           fi od;
       wlklhd[i]:=prod;
       scalefac:=scalefac+" od;
   for i to sizF[k] do; post[i]:=wlklhd[i]/scalefac od;
   RETURN(post)
end:

nextS:=proc(x,k,S) local j; j:=x-xmin[k]+1;
       [S[1..k-1],
                  [S[k][1..j-1],
                   S[k][j]+1,
                   S[k][ j+1..N[k] ]],
        S[k+1..K]]
end:

part:=proc(k,s,f) #list of partitions of s into k integers, repetitions
                  #allowed. Each partition is a list of k elements, were each
                  #element can only be equal to 0, or f or 2*f or ... l*f=s.
      local result, i, d, l;
      options remember;

      l:=s/f;
      if k=1 then RETURN( [ [s] ] ) fi;
      result:=[];
      for i from 0 to l do;
          d:=i*f;
          part(k-1,s-d,f);
          map(proc(x,d1) [d1,op(x)] end, ",d);
          result:=[op("), op(result)] od;
      result
end:
```

```
# "input data"
print('input');
K:=2;
h:= 11;
xmin:=[1,2]; xmax:=[3,4]; N:=[3,3];

# 5. Assume that the list of alternatives is explicitly given.
# as follows:

F[1]:= [ [0.1, 0.1, 0.8], [0.1, 0.5, 0.4] ];
F[2]:= [ [0.8, 0.1, 0.1], [0.4, 0.3, 0.3] ];
printlevel := 0;
S0:=[[0,0,0],[0,0,0]]; #initial state

for k to K do
    sizF[k] := nops(F[k]);
    prior[k]:=array(1..sizF[k]);
    for i to sizF[k] do prior[k][i]:=1/sizF[k] od
          od;
print('output');
dummy:=1;
print('expected value of any optimal strategy:',V(1,S0));
dummy:=1;

#list of possible states while following optimal policy, and the corresponding
#optimal decision:

print('optimal choices following one of the optimal strategies');

possStates:=proc(t)
   options remember;
   if t=1 then [S0]
   else
      map ( proc(aState) local d,l,x;
               l:=[];
               d:=dstar[t-1,aState];
               for x from xmin[d] to xmax[d] do l:=[nextS(x,d,aState),op(l)] od;
               op(l)
            end,
            possStates(t-1) )
   fi
end:
for t to h do possStates(t); map( proc(x) print(x,dstar[t,x]) end,") od;
print('time'); time();
quit;
```

Appendix B

Magalhaes' tape

The following pages contain a listing of the beginning of the transaction load for database B on day 2. The first column in the listing denotes the database and the day (B2). The second column is the start time of the transaction in hexadecimal.The third column is the transaction number. Then there are up to four records per transaction, as follows:

- Record "A" is a coded version of the transaction. Most of the transactions are predefined; transactions resemble procedure calls,

  transId(arg1,arg2,...)

  The transaction Id for *insert1* is "C1771". The formal arguments are:

  C1771(c01,c21,c61,c74,c75,c76,...)

- Record "B" includes messages issued by the system during the execution of the transaction.

- Record "C" shows i/o and CPU costs per database module. Modules are coded by 3-digit integers. We are interested in the module that handles qualifications(the "where") clause; this is module 303. The 3 numbers after the module id are the number of i/o block references in the database, the number of block references in scratch files, and the cpu cost.

- Record "D" encodes the sequences of i/o operations to the database files, showing file number, block number and kind of operation (read or write). These records were omitted from the sample listing.

```
B2 00000000    1 A  1  TIMING ON: ECHO ON:
B2 014C4956    2 A  1  USER,SILL:DBN IS BUILDNG:
B2 014C4956    2 B  1 -556- OPENED.....BUILDNG              367       1    80/02/0
2  05:55:52
B2 014C4956    2 C  1 500  0   1  0 300  0   0  0
B2 014C5054    3 A  1 *C773(800205):
B2 014C5054    3 C  1 301  1   4  0 303 11  29  5 304 13  92  0 307  0   0  2 30
4 27 107  0
B2 014C5054    3 E  1   79 93   83 93   84 93   74 93   71 93   78 93   75 93
76 93   77 93
B2 014CC7BE    4 A  1 *C783(800205):
B2 014CC7BE    4 C  1 301  0   2  0 303 16  69  5 304  1  17  0 307  0   0  0 30
4  3  18  0
B2 014CC7BE    4 E  1   79  9   83  9   84  9   74  9   75  9   76  9   81 12
82 12
B2 014CDBAO    5 A  1 *C775(800205):
B2 014CDBAO    5 C  1 301  1   4  0 303 11  29  5 304  9  91  0 307  0   0  2 30
4 31 174  0
B2 014CDBAO    5 E  1   79 93   83 93   84 93   74 93   71 93   78 93   75 93
76 93   77 93
B2 014D551C    6 A  1 *C783(800205):
B2 014D551C    6 C  1 301  0   2  0 303 16  69  5 304  1  17  0 307  0   0  0 30
4  3  18  0
B2 014D551C    6 E  1   79  9   83  9   84  9   74  9   75  9   76  9   81 12
82 12
B2 014D69F8    7 A  1 *C773(800204):
B2 014D69F8    7 C  1 301  1   4  0 303  5  29  5 304 16 101  0 307  0   0  2 30
4 30 117  0
B2 014D69F8    7 E  1   79110   83110   84110   74110   71110   78110   75110
76110   77110
B2 022EB90E    8 A  1 CLEAR AUTO:
B2 022EC23C    9 A  1 *C1771(700U,15,C,800206,09.00,12.00,MCDONALD,SAME,6880,GEN
 SRVS,8):
B2 022EC23C    9 B  1 -342-       1 SELECTED RECORD(S) -
B2 022EC23C    9 B  2 -258- UPDATE CYCLE=     2 -
B2 022EC23C    9 C  1 301  1   4  0 303  2  28  0 306  0   3  1 307  0   1  0 30
8  2  24  1 301  1   4  0 303  3  35  0
B2 022EC23C    9 C  2 304  0   1  0 307  0   0  0 304  0   0  0
B2 022EC23C    9 E  1   79  1   83  1   84  1   74  1   71  1   78  1   75  1
76  1   77  1
B2 022F27D6   10 A  1 *C972(800205,07)
B2 022F27D6   10 C  1 301  1   2  0 303 11  31  6 304  0   5  0 307  0   0  0 30
4  1   4  0
B2 022F27D6   10 E  1   79  4   83  4   84  4   74  4   71  4   78  4   75  4
76  4   77  4
B2 022F5030   11 A  1 *C1771(60M,1,103,800204,10.00,12.00,BRODIE,SAME,3218,PRPTY
,4):
B2 022F5030   11 B  1 -342-       1 SELECTED RECORD(S) -
B2 022F5030   11 B  2 -258- UPDATE CYCLE=     3 -
B2 022F5030   11 C  1 301  1   3  0 303  1   9  0 306  0   4  1 307  0   4  0 30
8  2  23  1 301  1   4  0 303  2  17  0
B2 022F5030   11 C  2 304  0   2  0 307  0   0  0 304  0   4  0
B2 022F5030   11 E  1   79  1   83  1   84  1   74  1   71  1   78  1   75  1
76  1   77  1
B2 022FDA0A   12 A  1 *C1771(700U,05,D,800204,13.00,17.00,GUPTA,SHELLY,4152,TUD,
2):
B2 022FDA0A   12 B  1 -342-       1 SELECTED RECORD(S) -
```

```
B2 022FDAOA   12 B  2 -258- UPDATE CYCLE=     4 -
B2 022FDAOA   12 C  1 301  1   1  0 303  2  28  0 306  0   3  1 307  0   1  0 30
8  2  23  1 301  2   4  0 303  3  35  0
B2 022FDAOA   12 C  2 304  0   3  0 307  0   0  0 304  0   4  0
B2 022FDAOA   12 E  1   79  2   83  2   84  2   74  2   71  2   78  2   75  2
76  2   77  2
B2 022FE2C0   13 A  1 *C1771(700U,05,D,800206,13.00,17.00,GUPTA,SHELLY,4152,TUD,
2)
.
.
. etc. ...
.
.
. etc. ...
```

Appendix C

Analytic cost model of system 2000 DVT table

As explained in the text, we only utilize the B-tree and Multiple Occurrences Table models developed by Casas-Raposo[19], modified to allow for exact rather than average selectivities. Also, we only require the parts of the models that deal with simple one-key searches in the storage structures. The following are the relevant page numbers in Casas-Raposo's thesis:6.5.3, 6.5.4, A.1, A.2 and C.2.

C.1 B-tree access model

This model is based on a similar one by Batory [10].

Assumptions

Distribution of keys in blocks is uniform. All blocks of the B-tree contain the same number of keys on average

Input parameters

dvtN	—Number of keys in tree
ddtRlength	—Length of the keys
dvtPad	—Block padding [0–1]. Fraction of block reserved for additions
dvtBovh	—Block overhead. [0–1]. Used by block pointers etc.

Intermediate values and result

dvtR	—record capacity
dvtH	—expected number of records on a leaf block
dvtL	—height of tree
Cost	—Access cost in number of blocks

$$
\begin{aligned}
dvtR &= \left\lfloor \frac{dvtBlock \times (1 - dvtBovh)}{dvtRlength} \right\rfloor \\
dvtH &= \lfloor dvtR \times (1 - dvtPad) + 0.5 \rfloor \\
dvtL &= \lceil \log_{dvtN} dvtH \rceil \\
Cost &= dvtL
\end{aligned}
$$

Appendix D

Simulation model

This is the Icon[48] program that computes the cost of new strategies for transaction "insert1". The program should be easy to follow. We list here a few notes about Icon for readers not familiar with this powerful language:

1. Icon is a non-typed language; however, an option of the compiler forces all variables to be declared either Global or Local. Global variables are known in all procedures; Local variables only in the procedure in which they are declared. All procedures are actually functions that can return values of arbitrary data types.

2. The notation *record node(pageNum,firstChild)* declares a new data type named *node* with two fields. Later, $p := node(5,,)$ creates memory space for a new *node* and assigns a pointer to it to p. The second field is assigned &null. $c.pageNum$ references the first field.

3. *if /p ...* is an abbreviation of *if* $p = \&null$. Likewise, *if* \p ... abbreviates $ifpnot = \&null...$

4. $t01 := table()$ creates an associative table. $t01["a"] := 5$ assigns 5 to the element indexed by *"a"*. Types of indexes are arbitrary. A reference to a non-assigned index value returns *&null*. These tables are used to keep the inverted lists of pointers to the HT file.

5. $l := [5,3]$ assigns a list of two elements to l. New elements can be appended to the right of the list with $put("a",l)$ which would return the new list $[5,3,"a"]$

6. *every e* := !*l do*{*statements...*} is a loop expression that assigns to e each element of the list l in turn.

```
#This is a simulation of accesses to files 5 and 6 of the system
#2000 database. Database records in preorder are read from stdin and
#loaded to memory-resident versions of files 5 and 6. As the files are
#formed, the physical page number of each file entry is computed. This
#is then used to compute the exact number of page accesses for queries using
#several traversal strategies. Only the record fields relevant to queries
#"insert1" and "insert2" are kept in memory; however, full record lengths
#are used to estimate page numbers in file 6.

#TYPE rectype = {"00","10","20",..."80"}
#     recPoin = {@rec00,@rec10,...}
#     indxedattr = {"c01","c21","c61","c74","c75"}

record node(   #file 5 entry
        father, sibling, firstChild, #:node. pointers to family members
        dataPoin,                #:recPoin.  Pointer to file 6 entry
        recordId,                #:rectype.
        pageNum)                 #:int. Page number of this entry in file 5

record rec00(  #entry for record 00 in file 6
        c01,                     #:string. Value of field C01.
        pageNum)

record rec20(c21, pageNum)
record rec60(c61, pageNum)
record rec70(c74,c75,pageNum)
record recOther(pageNum)

#CONST

global L,               #:table[rectype] of int. Length of each record
       fat,             #:table[rectype] of rectype. Father of each record
       brot,            #:table[rectype] of rectype. Brother, if any.
        klen,           #:table[indxedatr] of integer. Lengths of indexed items
       PageLen5, PageLen6,  #page lengths
       NodeLen,         #length of a file 5 node
        dvtBlock,       #block length of DVT file
        dvtBovh,        #block overhead
        dvtPad          #padding
```

```
#VAR
global root,            #:node Pointer to root node.
       eof5, eof6,      #number of bytes used in each file
       c01, c21, c61, c74, c75   #:table[indxedatr] of (list of node).Inverted
                                 #lists of file 5 node pointers for each
                                 #distinct attribute value.

procedure main()

initial {
        fat := table(); brot := table(); L := table(); klen := table()
        c01 := table(); c21:=table(); c61:=table();c74:=table();c75:=table()
        fat["00"] := "root";
        fat["10"] := fat["20"] := "00"; brot["20"] := "10" ;brot["10"]:="20";
        fat["30"] := fat["60"] := "20"; brot["30"] := "60" ;brot["60"]:="30";
        fat["40"] := "30"; fat["50"] := "40";fat["70"] := "60";fat["80"] := "70"
        PageLen5 := 2492; PageLen6 := 2492;
        NodeLen  := 14;
        dvtBlock := 2492; dvtBovh := 0.05; dvtPad := 0.25
        L["00"] := 42; L["10"] := 5; L["20"] := 27; L["30"] := 56;L["40"] := 57
        L["50"] := 16; L["60"] := 10;L["70"] := 106;L["80"] := 23;
        klen["c01"] := klen["c21"] := klen["c61"] := klen["c74"] := 6
        klen["c75"] := 5
}

builddb ()    #build database
#traverse(root)
queries()     #read queries and compute costs
return
end
#----------------
procedure builddb()

local l,v,v2,
      currPg5,    #current page in file 5
      recId,      #:recType. Record id of current record
      recPoin,    #:recPoin. Pointer to record in file 6.
      p,          #:node. Pointer to file 5 entry of current record
      papa,       #:node. Pointer to father of current record
      last,            #:table[rectype] of node. Last record inserted for
                       # each rectype.
      h;          #:node. Pointer to older brother of current record

eof5 := 0; eof6 := 0
last := table()

last["root"] := root := node(,,,,"R",,)

while l := nextRecord() do {
        recId := l[1]; recPoin := l[2]
        currPg5 := eof5 / PageLen5
        eof5 +:= NodeLen
        papa := last[ fat[recId] ]
        (\papa) | assert(1)
        p := node(papa,,,recPoin,recId,currPg5);

        if /papa.firstChild then
```

```
         papa.firstChild := p
      else
         if \(h := last[recId]) & h.father === papa & /h.sibling then
            h.sibling := p
         else {
            (\(h := last[ brot[recId] ]) & h.father === papa  &
            /h.sibling) | assert(2)
            h.sibling := p }

      case recId of {
      "00": {v:=recPoin.c01;if /c01[v] then c01[v] := [p] else put(c01[v],p)}
      "20": {v:=recPoin.c21;if /c21[v] then c21[v] := [p] else put(c21[v],p)}
      "60": {v:=recPoin.c61;if /c61[v] then c61[v] := [p] else put(c61[v],p)}
      "70": {v:=recPoin.c74; v2:=recPoin.c75
            if /c74[v]  then c74[v] := [p]  else put(c74[v],p)
            if /c75[v2] then c75[v2] := [p] else put(c75[v2],p)}
      }
      last[recId] := p }
return
end
#---------------------
```

```
procedure nextRecord() #read next record and store it in file 6.
                       # Returns [RecordId, PointerToFile6]
                       # Fails on eod.

local recId, currPg6,
      p;               #:recPoin. Pointer to file 6 record.

if (recId := read()) ~== "EOD"  then {
        currPg6 := eof6 / PageLen6; eof6 +:= L[recId]
        if (eof6 / PageLen6) > currPg6 then currPg6 +:= 1

        case recId of {
                "00": p := rec00(read(),currPg6)
                "20": p := rec20(read(),currPg6)
                "60": p := rec60(read(),currPg6)
                "70": p := rec70(read(),read(),currPg6)
                default: p := recOther(currPg6) }

        return ([recId, p]) }
fail;
end
#--------------------
global cost,cost2,cost4,cost5,cost6,  #:int. Cost of new strategy
                                      #on each s2k file
       oldcost5,oldcost6, #:int. Old values of costs for debugging
       lastP5, lastP6,    #:int. Last page accessed in each file
       debug,
       vc01, vc21, vc61, vc74, vc75   #:strings. Values of attributes in query

procedure queries()

#reads "insert1" queries and computes the cost of several strategies.

local s2kcost                      #:int. Cost of s2k strategy

debug := "1"
while vc01 := read() do {      #for each insert1 query
        vc21:=read(); vc61:=read(); vc74:=read(); vc75:=read(); s2kcost:=read();
        write(vc01," ",vc21," ",vc61," ",vc74," ",vc75," ",s2kcost)
        writes("    ")
        strategy2(); write(); writes("         ",);
        strategy3(); write(); writes("               ");
        strategy4(); write(); writes("                   ");
        strategy5()
        write() }
return
end
#--------------------
procedure strategy2() #use Btree for c01 to access Rec00. Then walk down the
                      #hierarchy to access rec20, rec60 and rec70.

local l,                           #:list of node. Inverted list for vc01
      p                            #:node. Traveling pointer

        oldcost5 := cost5 := oldcost6 := cost6 := 0
        lastP5 := -1; lastP6 := -1;
```

```
        cost2 := Btree(*c01,klen["c01"],dvtPad)
        if(debug=="1") then writes("c2 ",cost2);
        if \(l := c01[vc01]) then {
            (*l = 1 ) |assert(13)
            p := l[1]
            (type(p) == "node") | assert(10)
            updtcost5(p);
            if (debug=="1") then princosts()
            p := findfirst(vc21, "20", p)
            if \p then {
                if (debug=="1") then princosts()
                p := findfirst(vc61, "60", p)
                if \p then {
                    if (debug=="1") then princosts()
                    cost2 *:= 2; cost5 *:= 2; cost6 *:=2
                    if (debug=="1") then princosts()
                    insert70(vc74, vc75, p)
                    findall(vc74, "70", p)
                    if (debug=="1") then princosts() } } }
        cost := cost2 + cost5 + cost6
        writes ("s2 ",cost)
return
end
#--------------------
procedure strategy3() #access rec20 via Btree of c21 and perhaps MOT file.
                      #check its father, then walk down the hierarchy for
                      #rec60 and rec70.

local l, p, f

cost2 := Btree(*c21,klen["c21"],dvtPad); cost4 := 0
if(debug=="1") then writes("c2 ",cost2);
oldcost5 := cost5 := oldcost6 := cost6 := 0
lastP5 := -1; lastP6 := -1;
l := c21[vc21]
```

```
if \l then {
        if *l > 1 then cost4 := mot(*l)
        if(debug=="1") then writes(" c4 ",cost4,"~");
        if \(f:= findfather(l,vc01) ) then {
                p := f[1]
                updtcost5(p)
                if (debug=="1") then princosts()
                p := findfirst(vc61, "60", p)
                if \p then {
                    if (debug=="1") then princosts()
                    cost2 *:= 2; cost4 *:= 2; cost5 *:= 2; cost6 *:=2
                    if (debug=="1") then princosts()
                    insert70(vc74, vc75, p)
                    findall(vc74, "70", p)
                    if (debug=="1") then princosts() } } }
writes("s3 ",cost2 + cost4 + cost5 + cost6)
return
end
#--------------------
procedure strategy4() #access rec60 via btree and perhaps mot. Check its father
                      #and grandpa, then access its children rec70
local l, p, f, g

cost2 := Btree(*c61, klen["c61"], dvtPad); cost4 := 0
if(debug=="1") then writes("c2 ",cost2);
oldcost5 := cost5 := oldcost6 := cost6 := 0
lastP5 := -1; lastP6 := -1;
l := c61[vc61]
if \l then {
        if *l > 1 then cost4 := mot(*l)
        if(debug=="1") then writes(" c4 ",cost4,"~");
        if \(f:= findfather(l,vc21) ) & \(g:= findfather([f[2]],vc01) ) then{
                    p := f[1]
                    updtcost5(p)
                    if (debug=="1") then princosts()
                    cost2 *:= 2; cost4 *:= 2; cost5 *:= 2; cost6 *:=2
                    if (debug=="1") then princosts()
                    insert70(vc74, vc75, p)
                    findall(vc74, "70", p)
                    if (debug=="1") then princosts() } }
writes("s4 ",cost2 + cost4 + cost5 + cost6)
return
end
#------------------
```

```
procedure strategy5() #use strategy 3 for the insert, then btree and
                      #mot and up up up for the list
local l, p, f, g, gg

cost2 := Btree(*c21,klen["c21"],dvtPad); cost4 := 0
if(debug=="1") then writes("c2 ",cost2);
oldcost5 := cost5 := oldcost6 := cost6 := 0
lastP5 := -1; lastP6 := -1;
l := c21[vc21]

if \l then {
        if *l > 1 then cost4 := mot(*l)
        if(debug=="1") then writes(" c4 ",cost4,"~");
        if \(f:= findfather(l,vc01) ) then {
                p := f[1]
                updtcost5(p)
                if (debug=="1") then princosts()
                p := findfirst(vc61, "60", p)
                if \p then {
                    if (debug=="1") then princosts()
                    insert70(vc74, vc75, p)
                    cost2 +:= Btree(*c74,klen["c74"],dvtPad)
                    if(debug=="1") then writes("c2 ",cost2);
                    l := c74[vc74]
                    (\l) | assert(56)
                    if *l > 1 then cost4 +:=mot(*l)
                    if(debug=="1") then writes(" c4 ",cost4,"~");
                    every p := !l do {
                        if /(f:=checkfather(p,vc61)) then next
                        if /(g:=checkfather(f,vc21)) then next
                        if /(gg:=checkfather(g,vc01)) then next
                        updtcost5(p) }
                    if (debug=="1") then princosts() } } }
writes("s5 ",cost2 + cost4 + cost5 + cost6)
return
end


#----------------------------
procedure findfather(l,v) #find the father of one of the records in l that
                          #has value v. Return [record, father], or null
                          #if not found.
local f, d, p

every p := !l do {
        updtcost5(p)
        f := p.father
        updtcost5(f)
        d := f.dataPoin
        updtcost6(d)
        if d[1] == v then return [p,f] }
return &null
end
#------------------
procedure checkfather(p,v) #check if father of p has value v. If so return
                           #father. Else return &null
local f,d
```

```
updtcost5(p)
f := p.father
updtcost5(f)
d := f.dataPoin
updtcost6(d)
if d[1] == v then return f else return &null
end
#------------------
procedure princosts()
        writes (right(cost5-oldcost5,3), right(cost6-oldcost6,3),"~")
        oldcost5 := cost5; oldcost6 := cost6
return
end
#---------------------------
procedure findfirst(v, id, p) #find first record of type "id" under p
                                #with first value == to v. Return node
                                #pointer or null. Updates globals lastP5,
                                #lastP6, and cost as it goes along.
local d         #:recpoint. ".

            p := p.firstChild;

            repeat {
                if /p then return p
                updtcost5(p)
                (p.recordId == id |
                 (\brot[id] & p.recordId == brot[id]) ) | assert(11)
                if p.recordId == id then {
                        d := p.dataPoin;
                        updtcost6(d);
                        if d[1] == v then return p
                        }
                p := p.sibling }
end
#-------------------------
```

```
procedure findall(v, id, p) #find all records under "p", al the corresponding
                            #file 6 records, and update costs. Other arguments
                            #ignored.
local d         #:recpoint. ".

            p := p.firstChild;

            while \p do {
                updtcost5(p)
                (p.recordId == id ) | assert(12)
                d := p.dataPoin;
                updtcost6(d);
                p := p.sibling }
return
end

#-----------------------------
procedure insert70(v74, v75, p) #insert c70 record under p
local new5, new6, currPg5, currPg6

        currPg6 := eof6 / PageLen6; eof6 +:= L["70"]
        if (eof6 / PageLen6) > currPg6 then currPg6 +:= 1
        currPg5 := eof5 / PageLen5
        eof5 +:= NodeLen
        new6 := rec70(v74, v75, currPg6)
        new5 := node(p,,,new6,"70", currPg5)
        new5.sibling := p.firstChild; p.firstChild := new5
        if /c74[v74] then c74[v74] := [new5] else put(c74[v74],new5)
        if /c75[v75] then c75[v75] := [new5] else put(c75[v75],new5)
return
end
#--------------------
procedure updtcost5(p)
if p.pageNum ~= lastP5 then { cost5 +:=1; lastP5 := p.pageNum}
return
end

procedure updtcost6(p)
if p.pageNum ~= lastP6 then { cost6 +:=1; lastP6 := p.pageNum}
return
end

procedure traverse(t)
        if /t then return
        if t.recordId ~== "R" then {
                writes(t.recordId)
                if type(t.dataPoin) ~== "recOther" then writes (" ",t.dataPoin[1])
                if t.recordId == "70" then writes(" ",t.dataPoin[2])
                write("    p5: ", t.pageNum, "    p6: ", t.dataPoin.pageNum)}
        traverse(t.firstChild)
        traverse(t.sibling)
end


procedure assert(n)
        write ("internal con#%fuzzion number ",n)
        write ()
```

```
        return;
end

procedure Btree (dvtN, dvtRlength, dvtPad) #access cost of a s2k Btree with N
                                           #keys of length Rlength
local dvtR,         #record capacity
      dvtH,         #expected number of records on a leaf block
      dvtL          #height of tree

dvtR := floor(dvtBlock * (1 - dvtBovh) / dvtRlength)
dvtH := floor(dvtR * (1 - dvtPad) + 0.5)
dvtL := ceilLog(dvtH, dvtN)
return (dvtL)
end

procedure floor(x); return(integer(x)); end

procedure ceil(x)
if x-floor(x) > 0.5 then return(floor(x)+1)
                    else return(floor(x))
end

procedure ceilLog(base,x) #ceil of log of x
local i
i:= 0;
repeat {
        if real(base)^i > x then return i
        i +:= 1
}
end

procedure mot(motKeyKVOccurrences); return(1); end
```

Appendix E

Database B

The next page shows a few records of database B. The first two digits on each line are the record Id. Then there is a list of (attribute id, value) pairs. The original data on the tape was a continuous stream of characters, with an asterisk after every record id and after every attribute value. A simple awk[2] filter produced the listing. Another filter preserved only the attributes used in the *insert1* transactions, for input to the simulation model.

```
00   1 700U 2 700 UNIVERSITY AVE 4 800201
10   11 P1 12  14.417
10   11 P2 12  9.500
10   11 P3 12  8.967
10   11 T1 12  10.000
10   11 T2 12  20.000
10   11 T3 12  50.000
10   11 T4 12  65.000
10   11 T5 12  15.000
10   11 T6 12  40.000
10   11 T7 12  20.000
20    21 LC 22  9114 23  2136 24  0 25  0 26  46 27  7041 28  14
60   61 053 62  15 63 2
70    71 TROUPE 72 GLENNA 73 2732 74 800205 75 13.00 76 16.30 77  2 78 BOF 79 70
OU 83 LC 84 053
70    71 TROUPE 72 GLENNA 73 2732 74 800207 75 08.30 76 16.30 77  2 78 BOF 79 70
OU 83 LC 84 053
70    71 TROUPE 72 GLENNA 73 2732 74 800204 75 08.30 76 16.30 77  2 78 BOF 79 70
OU 83 LC 84 053
70    71 NODWELL 72 CAROLANN 73 6452 74 800206 75 09.00 76 16.00 77  12 78 COMP
PLN 79 700U 83 LC 84 053
80   81 OVHD PROJ 82 1
70    71 FLETCHER 72 SAME 73 3631 74 801202 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 801104 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 801007 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800902 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800805 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800701 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800603 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800506 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800401 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800304 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
70    71 FLETCHER 72 SAME 73 3631 74 800205 75 09.00 76 12.00 77  10 78 PRNTNG 7
9 700U 83 LC 84 053
30   31 SDPD 32 PRINTING SERVICES 33 J. IVY 34 HO E21
40    41 P1 42 733139 43 56933000000081248001 44  33362.71 45  .00 46  33362.71
50   51  2233 52 790501 53 790930
50   51  2254 52 791001
40    41 P3 42 733139 43 56933000000081248001 44  .00 45  .00 46  .00
40    41 T2 42 733139 43 56933000000081483801 44  120.00 46  120.00
50   51  6 52 790101
40    41 T3 42 733139 43 56933000000081483801 44  200.00 46  200.00
50   51  4 52 780301
40    41 T6 42 733139 43 56933000000081483801 44  800.00 46  800.00
50   51  20 52 790801
30   31 FISD 32 FINANCIAL INFO SYSTEMS 33 S.A. FRASER 34 H3A1
40    41 P2 42 510131000000521002 43 56936000000081248001 44  304.00 45  .00 46
```

```
 304.00
50   51  32 52 760101
30   31 SPDO 32 SYSTEM PLANNING 33 MRS. E. BARNES 34 H8 F18
40    41 P2 42 7201306 43 56936000000081248001 44  126.00 46  .00
40    41 P2 42 72013060000072600940 43 5693600000081248001 44  133.00 46  133.0
0
50   51  14 52 790201
30   31 BSMO 32 ADMIN SYS & CONTROL-BRANCH SERVICES 33 J.W. MC MARTIN 34 H7J21
40    41 P2 42 58313200000021209800 43 5693500000081248001 44  342.00 46  342.0
0
50   51  36 52 780201
30   31 HEPC 32 HEPCOE 33 0000 34 0000
40   41 00 42 56932000000081248001 43 5693200000081248001 44  480.00 46  480.00
50   51  1328 52 780101
30   31 DCAS 32 DESIGN&CONST-ADMIN SYS. 33 G. BOX 34 H14A4
40    41 P2 42 790138 43 5693500000081248001 44  674.00 45  .00 46  674.00
50   51  71 52 790607
30   31 BOF1 32 BLDG & OFF FAC 33 D.I. SILLARS 34 H2G3
40    41 P1 42 56913000000081210200 43 5693200000081248001 44  14330.00 45  .00
 46  14330.00
50   51  994 52 780913
40    41 T2 42 56913000000081210300 43 5693200000081483801 44  40.00 46  40.00
50   51  2 52 771201
40    41 T3 42 56913000000081210300 43 5693200000081483801 44  600.00 46  600.0
0
50   51  12 52 780901
40    41 T4 42 56913000000081210300 43 5693200000081483801 44  65.00 46  65.00
50   51  1 52 760601
40    41 P2 42 56913000000081210200 43 5693200000081248001 44  2156.00 45  .00
46  2156.00
50   51  227 52 790607790607
40    41 T6 42 56913000000081210300 43 5693200000081483801 44  400.00 46  400.0
0
50   51  10 52 790501
30   31 AMEU 32 AMEU 33 J.W. MC MARTIN 34 H7J21
```

```
40    41 P2 42 57513100000026209800 43 56935000000081248001 44  85.00 45  .00 46
  85.00
50    51  9 52 760801
30    31 DPDO 32 DATA PROCESSING 33 MRS.R.K. ESDON 34 M4D10
40    41 P2 42 54213010 43 56934000000081248001 44  893.00 45 .00 46  893.00
50    51  94 52 760101
30    31 STDS 32 STA T&D-SURVEY 33 G. BOX 34 H14A4
40    41 P1 42 78513090 43 56935000000081248001 44  .00 46  .00
50    51  48 52 780601 53 790930
40    41 T6 42 78513090 43 56935000000081483801 44  40.00 46  40.00
50    51  1 52 780601
30    31 CBLR 32 CABLE ROOM 33 F. RIMMER 34 CENTRAL REGION
40    41 P2 42 60303102310235608 43 56935000000081248001 44  180.00 45  .00 46
180.00
50    51  19 52 760101
30    31 TDOO 32 TREASURY DIV 33 D. PEPER 34 H2 AO1
40    41 P1 42 519131000000741001 43 56936000000081248001 44  .00 45  .00 46  .0
0
40    41 T2 42 519131000000741001 43 56936000000081483801 44  .00 46  .00
40    41 T3 42 519131000000741001 43 56936000000081483801 44  .00 46  .00
40    41 T6 42 519131000000741001 43 56936000000081483801 44  .00 46  .00
40    41 P1 42 51913100000054100100 43 56936000000081248001 44  20255.00 46  202
55.00
50    51  1405 52 790101
40    41 T2 42 51913100000054100100 43 56936000000081483801 44  20.00 46  20.00
50    51  1 52 790101
40    41 T3 42 51913100000054100100 43 56936000000081483801 44  950.00 46  950.0
0
50    51  19 52 790101
40    41 T6 42 51913100000054100100 43 56936000000081483801 44 120.00 46  120.00
50    51  3 52 800101
30    31 FIRE 32 FIRE 33 D.I.I SILLARS 34 H2G3
40    41 T1 42 56913000000081210300 43 56932000000081483801 44  60.00 46  60.00
50    51  6 52 770301
30    31 SLDO 32 SEC-EXECUTIVE 33 N. CATCHPOLE 34 H19E26
40    41 P1 42 5021311 43 56936000000081248001 44  1268.00 46  1268.00
50    51  88 52 780101
30    31 GPDE 32 GEN PROJ-DARLINGTON 33 G. BOX 34 H17B1
40    41 T6 42 78713 43 56935000000081483801 44  210.00 46  .00
40    41 T2 42 78713 43 56935000000081483801 44  45.00 46  .00
40    41 P1 42 78713 43 56935000000081248001 44  .00 45  .00 46  .00
50    51  67 52 781006 53 -1864.00
30    31 DDIS 32 DESIGN & DEVELOP-INST&CONTROL 33 G. BOX 34 H17B1
40    41 P2 42 77513 43 56935000000081248001 44  522.00 46  522.00
50    51  55 52 790101
30    31 VAGS 32 VISUAL & GRAPHIC SERVICES 33 D.I. SILLARS 34 H10A4
40    41 P1 42 56913000000081210204 43 56932000000081248001 44  2681.00 46  2681
.00
50    51  186 52 790101
40    41 T2 42 56913000000081210204 43 56932000000081483801 44  20.00 46  20.00
50    51  1 52 790101
40    41 T6 42 56913000000081210204 43 56932000000081483801 44  160.00 46  160.0
0
50    51  4 52 790101
30    31 PRDO 32 PUBLICRELATIONS 33 MISS. H. JANETAKES 34 H10D4
40    41 P2 42 52414000000013209005 43 56936000000081248001 44  .00 45  .00 46
.00
```

```
50   51  42 52 790606 53 791231
40    41 P2 42 52413000000013209005 43 56936000000081248001 44  399.00 46  399.0
0
50   51  42 52 800101
30   31 DCDO 32 DISTRIBUTED COMPUTING DEPT. 33 MR. F. COGEN 34 M4D2
40    41 T6 42 52013040 43 56934000000081483801 44  40.00 46  40.00
50   51  1 52 791201
30   31 PTBO 32 PRODUCTION & TRANSMISSION BR. 33 MRS. S. RUDDERHAM 34 H4H18
40    41 P2 42 58843091 43 56935000000081248001 44  1235.00 46  1235.00
50   51  130 52 800101
20   21 GF
20    21 MEZZ 22  3025 23  643 24  5200 25  0 26  0 27  1411 28  0
60   61 AUD 62  143
70    71 HYD CLB 72 SUSIE 73 5662 74 800402 75 12.00 76 13.00 77  30 78 PRY&BIBL
E 79 700U 83 MEZZ 84 AUD
.
.
.
```

Bibliography

[1] Carnegie-Mellon University. *Proceedings of an International Conference on Genetic Algorithms and their Applications*, 1985.

[2] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk —a pattern scanning and processing language. In *UNIX™ Programmer's Manual*, University of California, Berkeley, 1983.

[3] P. Armitage. The search for optimality in clinical trials. *International Statistical Review*, 53:15–24, 1985.

[4] N. Baba. On the learning behavior of variable-structure stochastic automata in the general n-teacher environment. *IEEE Transactions on Systems, Man and Cybernetics*, smc-13:224–231, 1983.

[5] T.A. Bancroft and C.P. Han. *Statistical Theory and Inference in Research.* Marcel Dekker Inc., New York, 1981.

[6] A. G. Barto and P. Anadan. Pattern-recognizing stochastic learning automata. *IEEE Transactions on Systems, Man and Cybernetics*, smc-15:360–375, 1985.

[7] A. G. Barto, P. Anadan, and C. W. Anderson. Cooperativity in networks of pattern-recognizing stochastic learning automata. In K. S. Narendra, editor, *Adaptive and Learning Systems*, pages 235–246, Plenum Press, New York, 1986.

[8] J. A. Bather. Randomized allocation of treatments in sequential experiments. *Journal of the Royal Statistics Society B*, 43:265–292, 1981.

[9] J. A. Bather. A simple bandit problem. In H. J. Tijms and J. Wessels, editors, *Markov Decision Theory*, pages 213–220, Mathematisch Centrum, Amsterdam, 1977.

[10] D. S. Batory. *An Analytic Model of Physical Databases.* PhD thesis, University of Toronto, Computer Systems Research Group, 1980. Tech. report CSRG-124.

[11] R. Bellman. *Adaptive Control Processes: a Guided Tour.* Princeton University Press, Princeton, New Jersey, 1961.

[12] R. Bellman. A problem in the sequential design of experiments. *Sankhya A*, 16:221–229, 1956.

[13] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, and J.B. Rothnie. Query processing in SDD-1. *Transactions on Database Systems*, 6:602–625, 1981.

[14] D. A. Berry and B. Firstedt. *Bandit Problems.* Chapman and Hall, London, 1985.

[15] A. D. Bethke. *Genetic Algorithms as Function Optimizers.* PhD thesis, University of Michigan, Ann Arbor, 1981. Dept. of Computer Science.

[16] R.N. Bradt, S.M. Johnson, and S. Karlin. On sequential designs for maximizing the sums of n observations. *Ann. Math. Statist*, 27:1060–1074, 1956.

[17] R. G. Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series.* Prentice Hall Inc., New Jersey, 1962.

[18] R. R. Bush and F. Mosteller. *Stochastic Models for Learning.* John Wiley and sons, New York, 1958.

[19] I. R. Casas-Raposo. *Prophet: a Layered Analytical Model for Performance Prediction of Database Systems.* PhD thesis, University of Toronto, Computer Systems Research Institute, 1986. Technical report CSRI-180.

[20] B. Chandrasekaran and D. W. C. Shen. On expediency and convergence in variable structure automata. *IEEE Transactions on System Science and Cybernetics*, ssc-4:52–60, 1968.

[21] B.W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *On the Design and Performance of the Maple System.* Technical Report 1, University of Waterloo, 1985. Computer Science Department.

[22] B.W. Char, K.O.Geddes, G.H.Gonnet, and S.M. Watt. *Maple User's Guide.* WATCOM Publications Ltd., Waterloo,Ont. Canada, 1985.

[23] H. Chernoff and A. J. Petkau. Sequential medical trials involving paired data. *Biometrika*, 68:119–132, 1981.

[24] S. Christodoulakis. Estimating block selectivities. *Information Systems*, 9:69–79, 1984.

[25] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8:105–115, 1983.

[26] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *Transactions on Database Systems*, 9:163–186, 1984.

[27] S. E. Clausen. Optimizing the evaluation of calculus expressions in a relational database system. *Information Systems*, 5:41–54, 1980.

[28] T. Colton. A model for selecting one of two medical treatments. *Journal of the American Statistical Association*, 58:388–400, 1963.

[29] MRI Systems Corporation. *System 2000 Publications: General Information Manual; Basic Reference Manual.* MRI, Austin, Texas, 1976.

[30] N.E. Day. A comparison of some sequential designs. *Biometrika*, 56:301–311, 1969.

[31] U. Dayal and N. Goodman. Query optimization for codasyl database systems. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, 138–150, 1982.

[32] K. de Jong. Adaptive system design: a genetic approach. *IEEE Transactions on Systems, Man and Cybernetics*, 10:566–574, 1980.

[33] M. H. DeGroot. *Optimal Statistical Decisions.* McGraw Hill, 1970.

[34] M.H. DeGroot. *Probability and Statistics.* Addison-Wesley, Philippines, 1975.

[35] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, New Jersey, 1979.

[36] R. Demolombe. Estimation of the number of tuples satisfying a query expressed in predicate calculus language. In *Proceedings of the Conference on Very Large Data Bases*, pages 55–63, 1980.

[37] T.G. Dietterich and R.S. Michalski. Learning and generalization of characteristic descriptions:evaluation criteria and comparative review of selected methods. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 223–231, 1979.

[38] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[39] B. Everitt. *Cluster Analysis.* Halsted Press, New York, 1980.

[40] J. Fabius and W. R. van Zwet. Some remarks on the two-armed bandit. *Ann. Math. Statist.*, 41:1906–1916, 1970.

[41] D. J. Finney. Improvement by planned multistage selection. *Journal of the American Statistical Association*, 79:501–509, 1984.

[42] S. French. *Decision Theory.* Ellis Horwood, West Sussex, 1986.

[43] K.S. Fu. *Syntactic Pattern Recognition and Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[44] J.C. Gittings. Comments to bather's[8] paper. *Journal of the Royal Statistical Society B*, 43:283, 1981.

[45] J. C. Gittins and D. M. Jones. A dynamic allocation index for the discounted multiarmed bandit problem. *Biometrika*, 66:561–565, 1979.

[46] D. E. Goldberg. The genetic algorithm approach: why, how, and what next? In K. S. Narendra, editor, *Adaptive and Learning Systems*, pages 247–253, Plenum Press, New York, 1986.

[47] J. Grefenstette, R. Gopal, B. Rosmaita, and D.V. Gucht. Genetic algorithms for the traveling salesman problem. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, The Robotics Institute, Carnegie-Mellon Uneversity, 1985.

[48] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice-Hall, New Jersey, 1983.

[49] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 1–8, 1976.

[50] C.J. Harris and S.A. Billings, editors. *Self-tunning and Adaptive Control.* Peter Peregrinus Ltd., London, 1985.

[51] D. V. Hinkley. Inference about the change point from cumulative sum tests. *Biometrika*, 58:509–523, 1971.

[52] D.G. Hoel, M. Sobel, and G.H. Weiss. A survey of adaptive sampling for clinical trials. In R. Elashoff, editor, *Perspectives in Biometrics*, pages 29–61, Academic Press, 1975. (vol. 1).

[53] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[54] A.D. Horowitz. *Experimental Study of the Two-Armed Bandit Problem.* PhD thesis, Univ. of North Carolina at Chapel Hill, USA, 1973.

[55] N. Kamel and R. King. A model of data distribution based on texture analysis. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 319–325, 1985.

[56] D. E. Knuth. *The Art of Computer Programming.* Addison Wesley, Massachusetts, 1973.

[57] V. Y. Krylov and M. L. Tsetlin. Games between automata. *Automation and Remote Control*, 24:975–987, 1963.

[58] G. C. Magalhaes. *Improving the Performance of Database Systems.* PhD thesis, University of Toronto, Computer Systems Research Group, 1981.

[59] L.G. Mason and XueDuo Gu. Learning automata models for adaptive flow control in packet-switching networks. In K. S. Narendra, editor, *Adaptive and Learning Systems*, pages 213–227, Plenum Press, New York, 1986.

[60] R. W. McLaren. A stochastic automata model for a class of learning controllers. In *Proceedings of the Joint Automatic Control Conference*, pages 267–273, 1967. University of Pennsylvania, Phyladelphia.

[61] K. S. Narendra and S. Lakshmivarahan. Learning automata - a critique. *Journal of Cybernetics and Information Science*, 1:53–65, 1977.

[62] K. S. Narendra and Jr. R. M. Wheeler. Recent developments in learning automata. In K. S. Narendra, editor, *Adaptive and Learning Systems*, pages 197–211, Plenum Press, New York, 1986.

[63] K. S. Narendra and M. A. L. Thathachar. Learning automata - a survey. *IEEE Transactions on Systems, Man and Cybernetics*, smc-4:323–334, 1974.

[64] K. S. Narendra and M. A. L. Thathachar. On the behavior of a learning automaton in a changing environment with application to telephone traffic routing. *IEEE Transactions on Systems, Man and Cybernetics*, smc-10:262–269, 1980.

[65] K. S. Narendra and R. Viswanathan. A two-level system of stochastic automata for periodic random environments. *IEEE Transactions on Systems, Man and Cybernetics*, smc-2:285–289, 1972.

[66] M. F. Norman. *Markov Processes and Learning Models.* Academic Press, New York, 1972.

[67] M. F. Norman. Some convergence theorems for stochastic learning models with distance diminishing operators. *Journal of Mathematical Psychology*, 5:61–101, 1968.

[68] Jr. O. V. Nedzelnitsky. *The Application of Learning Methodology to Message Routing in Data Communication Networks.* PhD thesis, Yale University, 1983.

[69] E. S. Page and L. B. Wilson. *An Introduction to Computational Combinatorics.* Cox and Wyman Ltd., London, 1979.

[70] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 256–276, 1984.

[71] H. Raiffa and R. Schlaiffer. *Applied Statistical Decision Theory.* Harvard University Press, Boston, 1961.

[72] J.R. Rice. A metalgorithm for adaptive quadrature. *Journal of the Association for Computing Machinery*, 22:61–82, 1975.

[73] R. Rivest. On self-organizing sequential search heuristics. *Communications of the Association for Computing Machinery*, 19:63–67, 1976.

[74] H. Robins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, 1952.

[75] J. Rodriguez-Rosell and D. Hildebrand. *A Framework for Evaluation of Data Base Systems.* Technical Report RJ1587, IBM San Jose Researh Laboratory, 1975.

[76] L. A. Rowe and M. Stonebraker. The commercial ingres epilogue. In M. Stonebraker, editor, *The INGRES Papers*, pages 63–82, Addison-Wesley, Reading, Mass., 1985.

[77] D. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, 1959.

[78] G.N. Saridis. *Self-Organizing Control of Stochastic Systems.* Marcel Dekker, Inc., New York, 1977.

[79] S. A. Schmitt. *Measuring Uncertainity: an Elementary Introduction to Bayesian Statistics.* Addison-Wesley, Reading, Mass., 1969.

[80] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in relational database management systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 23–34, 1979.

[81] D. Siegmund. *Sequential Analysis.* Springer-Verlag, New York, 1985.

[82] J. Sklansky and G. N. Wassel. *Pattern Classifiers and Trainable Machines.* Springer-Verlag, New York, 1981.

[83] P. M. Stocker and P. A. Dearnley. Self-organizing data management systems. *The Computer Journal*, 16:100–105, 1973.

[84] R.E. Tarjan. Algorithm design. *ACM Communications*, 30:205–212, 1987.

[85] M. A. L. Thathachar and P. S. Sastry. A new approach to the design of reinforcement schemes for learning automata. *IEEE Transactions on Systems, Man and Cybernetics*, smc-15:168–175, 1985.

[86] W.R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285–294, 1933.

[87] V. I. Varshavskii and I. P. Vorontosova. On the behavior of stochastic automata with variable structure. *Avtomat. Telemekh.*, 24:353–360, 1963.

[88] R. Viswanathan and K. S. Narendra. Stochastic automata models with applications to learning systems. *IEEE Transactions on Systems, Man and Cybernetics*, smc-3:107–111, 1973.

[89] W. Vogel. An asymptotic minimax theorem for the two-armed bandit problem. *Ann. Math. Statist*, 31:444–451, 1960.

[90] S. B. Yao. Approximating block accesses in database organizations. *Communications of the Association for Computing Machinery*, 20:260–261, 1977.

[91] S.B. Yao. Optimization of query evaluation algorithms. *Transactions on Database Systems*, 4:133–155, June 1979.

[92] C. T. Yu and C. H. Chen. Adaptive information system design: one query at a time. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 280–290, 1985.

[93] C.T. Yu and C.C. Chang. On the design of a query processing strategy in a distributed database environment. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, page 30, 1983.

[94] C.T. Yu, L. Lillien, K. Guh, M. Templeton, D. Brill, and A. Chen. Adaptive techniques for distributed query processing. In *IEEE International Conference on Data Engineering*, pages 86–93, 1986.

[95] C.T. Yu, C.M. Suen, K. Lam, and M.K. Siu. Adaptive record clustering. *Transactions on Database Systems*, 10:180–204, 1985.