

Optimization Techniques for Data-intensive Decision Flows

Richard Hull*, Francois Llibat[◇], Bharat Kumar*,
Gang Zhou*, Guozhu Dong[†], Jianwen Su[‡]

*Bell Laboratories, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974
{hull, kumar, gzhou}@lucent.com

[†] Dept. of Computer Science and Engineering
Wright State University
Dayton, OH 45435
gdong@cs.wright.edu

[◇] I.N.R.I.A.
Domaine de Voluceau-ROCQUENCOURT
78153 Le Chesnay CEDEX, FRANCE
francois.llibat@inria.fr

[‡] Computer Science Department
University of California
Santa Barbara, CA 93106
su@cs.ucsb.edu

Abstract

For an enterprise to take advantage of the opportunities afforded by electronic commerce it must be able to make decisions about business transactions in near-realtime. In the coming era of segment-of-one marketing, these decisions will be quite intricate, so that customer treatments can be highly personalized, reflecting customer preferences, the customer's history with the enterprise, and targeted business objectives. This paper describes a paradigm called "decision flows" for specifying a form of incremental decision-making that can combine diverse business factors in near-realtime.

This paper introduces and empirically analyzes a variety of optimization strategies for decision flows that are "data-intensive", i.e., that involve many database queries. A primary focus is on the use of parallelism and eagerness (a.k.a. speculative execution) to minimize work and/or reduce response time. A family of optimization techniques is developed, including algorithms and heuristics for scheduling tasks of the decision flow. Using a prototype execution engine the techniques are compared and analyzed in connection with decision-making applications having differing characteristics.

1. Introduction

A variety of technologies will be needed to support the explosive growth of electronic commerce. One family of research challenges concerns the development of new frameworks, infrastructures, and protocols that permit enterprises maximize their effectiveness when using e-commerce. This paper describes a paradigm called "decision flows" for

specifying and executing in near-realtime highly differentiated decisions in (e-commerce) workflows. For example, decision flows can be used to personalize the experience of web storefront customers or to help manage resources (e.g., deciding what machines or human agents should perform tasks) in the workflows that support e-commerce applications. Decision flows support a form of incremental decision-making, that can easily incorporate a myriad of business factors and specify the relative weights they should be given. This paper presents algorithms and heuristics for executing database-intensive decision flows, and describes an empirical analysis focused on minimizing workload and response time.

A decision flow consists in a family of attributes which may be evaluated during execution. Some of the attributes will be "target" and embody the output of a decision flow, e.g., what priority of service to give this customer, or what promotional image to display on the next web page. Other attributes correspond to intermediate results of the decision flow. For example, a "promo hit list" attribute might hold a listing of potential promo messages to display, along with scores combining the likelihood that a customer will buy the promo and the potential profit that might be derived. Some intermediate attributes might gather data from external sources, such as databases. Since attribute evaluation can have a real cost, enabling conditions are used to decide which attributes should be evaluated. (If an attribute A is disabled, it returns the null value \perp . Attributes that use A as input must be able to execute even if \perp is produced by A .) The set of data flow and enabling flow dependencies in a decision flow must form an acyclic graph. The "attribute-centric" perspective of decision flows permits a systematic approach for specifying what factors should be incorporated as a decision is being made.

Decision flows were first introduced in [HLS⁺99a] as part of the Vortex workflow model, that permits the spec-

ification of workflow schemas supporting highly differentiated treatments. In the decision flow model a variety of mechanisms are provided for specifying how attributes should be evaluated. This includes user-defined functions, database dips, and a generalized form of “business rules” (see [HLS⁺99a] for details). Decision flows are especially useful in customer care applications (e.g., e-commerce, call centers, insurance claims processing). Increasingly, these applications call for “segment-of-one marketing”, i.e., providing very personalized treatment to different customers [PR93]. In many cases, relevant data is widely distributed across an enterprise, and multiple database queries are needed to process each customer contact. Since current e-commerce and customer care applications must support thousands or even millions of contacts per day, there is a tremendous need for optimization of this kind of decision making, in terms of both throughput and response time.

The primary focus of the current paper is to present an empirical study comparing a variety of optimizations for data-intensive decision flows. The optimizations focus primarily on the judicious use of parallelism and speculative evaluation to reduce the *work performed* and the *response time* of processing instances of a decision flow. The technique of speculative execution has been applied in various areas, such as pipe-lined execution of machine level instructions in the field of computer architecture [PHG96]. Similar to the prior work, data flow plays an important role in the current application. In contrast with prior work, however, is the presence of enabling conditions on tasks. This permits forward and backward propagation of information about queries eligible for execution and queries unneeded for successful completion of the decision flow instance.

In §2, an example decision flow is presented, along with a formal description of the decision flow model. In §3, we present a traditional architecture for parallel processing of decision flows based on a prequalifier and a task scheduler. In §4 a family of optimization techniques is proposed, including algorithms for the prequalifier and heuristics for the task scheduler. We implemented a prototype execution engine based on these techniques. In a simulated environment, the techniques are compared and analyzed in connection with decision flow applications having differing characteristics. The results of our experiments are detailed in §5, along with tuning guidelines.

Due to space limitations the presentation here is quite terse (see [HLS⁺99b]). Also, to simplify the discussion we assume that all queries are made against a single database.

Additional related work. Decision flows can be used to support near-realtime decision making, and are richer than decision trees and traditional business rules frameworks. Decision flows are more structured than expert systems, and thereby reduce the potential for a “ripple” effect when individual rules are modified. The use of enabling conditions in decision flows is reminiscent of their use in the ThinkSheet model [PYLS96]. Decision flows are complimentary to decision support and data mining systems. Those systems pro-

vide tools to analyze large volumes of data that chronicle previous business transactions, to help develop appropriate policies for future transactions. Decision flows can be used to implement those policies during subsequent transactions.

Workflow systems such as Flowmark [LR94], Meteor [KS95], and others specify work activities (for human agents or computers) using graphs whose nodes are tasks and edges corresponding to enabling conditions. Although decision flows can serve as the basis for a workflow model (see [HLS⁺99a]), and workflow systems can use a decision flow engine as an adjunct, the current paper focuses primarily on the application of decision flows for near-realtime automated decision-making, where no human agents are involved.

An area related to data-intensive decision flows is that of “expert database systems” which focus on the use of one or more database systems to execute rule sets against large data sets in the spirit of expert systems. For example, [BKK87] focuses on cases where each rule might be instantiated by a large number of tuples, and uses a horizontal partitioning of the underlying data set to achieve effective parallelism. In contrast, decision flows are useful in applications such as e-commerce, where each execution of the decision flow involves relatively small data sets obtained from multiple data bases.

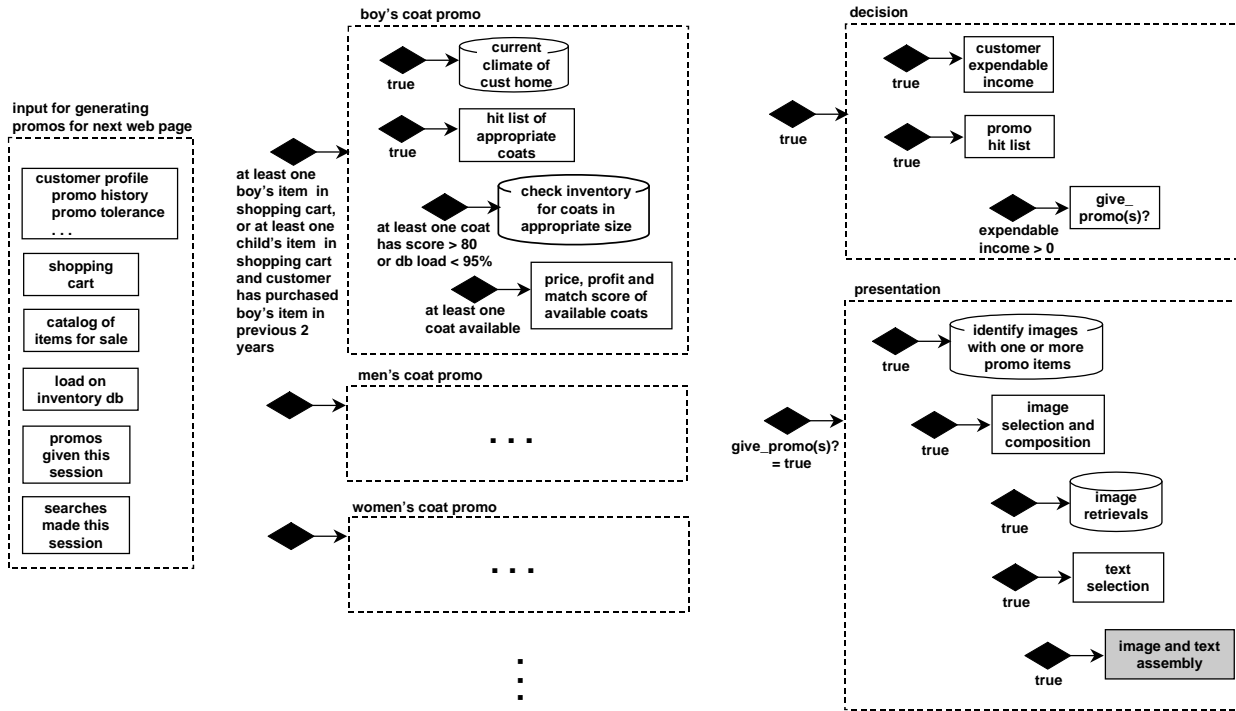
2. Data-intensive Decision Flows

This section presents an example application that illustrates decision flows. The section also presents a formal definition of decision flows, that is used to describe the execution model and optimization algorithms developed later.

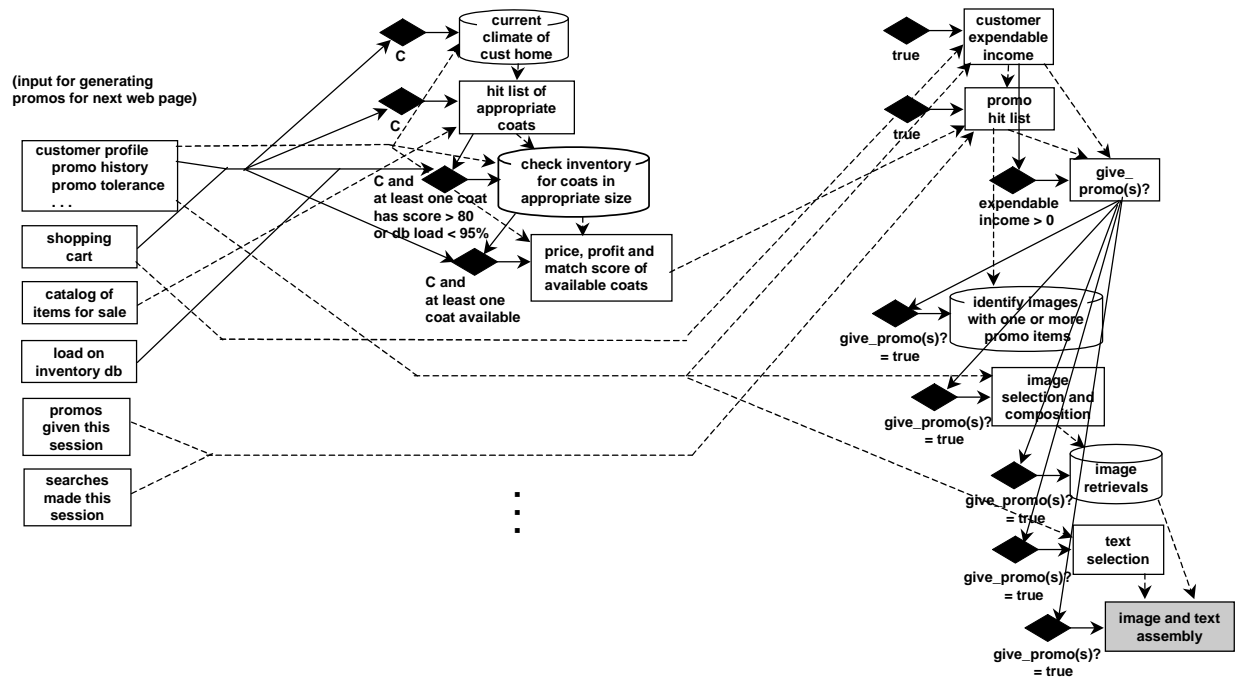
Decision flow for selecting promos when generating web pages. Figure 1(a) shows part of a (simplified) decision flow that could be used to respond to customers interacting with the web-based store front of a clothing retailer. The decision flow focuses on selecting items that can be promoted, and might be executed each time a page is generated for a customer. Other decision flows might be used to decide on the kind or level of service.

In Figure 1(a), each database icon and (solid boundary) rectangle corresponds to a *task* which might be performed for a given decision flow instance. Each task produces a value for one or more attributes whose values may be used by other tasks of the instance (“intermediate” attributes) or returned as an output value of the instance (“target” attributes). The dashed rectangles (except for the far left one) indicate groupings of tasks into *modules*; this helps support scalability in the specification of decision flows.

The input attributes for this decision flow include the profile of the customer, the current value of the shopping cart, information about promos that the business is especially interested in moving, etc. Based on different enabling conditions (shown as diamond nodes) different categories of promotions will be considered by the decision flow. For



(a) Module-based version of schema intended for users



(b) "Flattened" version of schema, with data flow (dashed) and control flow (solid) dependencies shown

Figure 1. Decision flow for selecting and generating promo images in web-based storefront

example, if there is already one boy’s item in the shopping cart, or if there is a child’s item in the shopping cart and the customer has bought something for a boy in the past two years, then a promo for a boy’s coat is considered. This involves doing a database dip to get information about the climate at the customer home, deriving a “hit list” of coats that might be appropriate to the customer, checking with inventory for coats in the appropriate size, and then creating a listing of possible coats to promo, along with info on the price, potential profit and degree of confidence that the promo matches customer interest.

The decision module will estimate the customer expendable income (based on customer profile, shopping cart, and perhaps other factors), and create a listing of promos obtained so far. Based on the business value of the promos and the likelihood of success, a decision is then made about whether to give promos.

Finally, if a promo will be given, the presentation module identifies images and text that can be used to display the promo(s), and assembles these for inclusion in the generated web page.

Attributes and tasks. A decision flow is *attribute-centric*: the main objective of the execution is to determine the values of certain attributes, based on other given or derived attribute values. Decisions made by a decision flow are represented in the attribute values.

Attributes are computed in decision flows by two kinds of tasks. A *foreign task* is external to the decision flow execution engine (e.g., database queries, web server routines, questions to a human). In general these can produce one or more attribute value, but for brevity in this paper we assume that each produces a single attribute. A *synthesis task* produces a single attribute value, specified by a user-defined function or using a specialized framework involving “business rules” (see [HLS⁺99a]),

Data flow and enabling flow. The decision flow model presented to users is modular, to support scalability and levels of abstraction. However, for execution we focus on a “flattened” version of the decision flow model, which permits more freedom with regards to the order of task execution. To flatten a module M , we combine (with the “and” connective) the enabling condition for M with the enabling condition of each task and submodule within M . The “flattened” version of the decision flow of Figure 1(a) is shown in part (b). (Ignore the lines and arrows for now.) For example, the enabling condition for the boy’s coat promo module (abbreviated as ‘C’) has been “anded” into each of the enabling conditions for the four tasks inside.

More formally, a (*flattened*) *decision flow schema* is a 4-tuple ($Att, Cond, Source, Target$) where

1. Att is a set of *attributes*. For each non-source attribute A there is a unique foreign or synthesis task which computes the value of A .
2. $Source$ and $Target$ are disjoint subsets of Att , corresponding to the *source* and *target* attributes, respec-

tively. The target attributes are used outside of the decision flow. In an execution of the decision flow, a value should be produced for each target attribute that is enabled (see below).

3. $Cond = \{C_A \mid A \text{ is a non-source attribute}\}$ is the set of enabling conditions, one for each non-source attribute.

The flow of data and enabling conditions in a decision flow is largely implicit. Associated to a (flattened) decision flow schema is its *dependency graph*, that highlights these two kinds of dependencies between attributes. Figure 1(b) shows the data flow (using dashed lines and arrows) and the enabling flow (using solid lines and arrows) for the example decision flow. A data flow edge is included from attribute A to attribute B if A is used as input for B (e.g., `promo_hit_list` to the module identifying images that show the promo items). An enabling flow edge is included from attribute A to attribute B if A is used in the enabling condition for B (e.g., `customer_expendable_income` to `give_promo(s)`).

A decision flow schema S is *well-formed* if the dependency graph of S is acyclic. We consider only well-formed decision flow schemas.

Execution of decision flows. Before presenting the declarative semantics for decision flows we describe intuitively how they can be implemented. During execution, an attribute becomes *stable* if its enabling condition becomes true and the task specifying the attribute has executed and returned a value, or if its enabling condition becomes false, in which the attribute is assigned the value \perp , i.e., null value. (In [HLS⁺99a] we distinguish exception values from other values.) A task can be executed after all of its input attributes have become stable. This and the acyclicity condition imply that attribute assignment is *monotonic*: if an attribute value is assigned, then it will never be overwritten.

Tasks in a decision flow must be capable of executing once their input attributes are stable, even if some of them have value \perp . This requirement is appropriate in many e-commerce applications, where a decision may have to be made with incomplete information, e.g., if a database is down.

A straightforward approach to implementing a decision flow is to proceed in an order given by a topological sort of the attributes according to the dependency graph. When an attribute A is considered, all of the inputs to the enabling condition of A , and all the data inputs for A , will be stable. Thus, the enabling condition of A can be evaluated, and if true, the task defining A can be evaluated. This paper develops optimizations of that approach, using parallelism, speculative evaluation and runtime algorithms that analyze the structure of decision flow schemas.

Intuitively, a target attribute is one that must be stable in order for execution of a decision flow instance to successfully complete. In the example the only target attribute is the one for image and text assembly (shown in gray). If this attribute is enabled, then execution will not complete until

a value is obtained. If the attribute becomes disabled, then execution can halt immediately. (This attribute will be disabled if attribute `give_promo(s)?` is false, which can occur if `customer_expendable_income = 0`.)

Declarative semantics of decision flows. In the abstract, during execution an attribute will have one of four states: UNINITIALIZED, ENABLED, VALUE or DISABLED. (Additional states are possible and described when specific execution details are involved; see §3 below.) Source attributes start with state VALUE. An attribute will become ENABLED if its enabling condition becomes true, and it will become DISABLED if its enabling condition becomes false. If ENABLED, an attribute will take a value and will then reach the VALUE state. If DISABLED, the attribute will take the null value \perp .

The semantics of decision flows is declarative, and defined using the notion of “complete snapshot”. A *complete snapshot* is a pair $s = (\sigma, \mu)$, where

- (a) the state function σ maps each non-source attribute into $\{\text{VALUE}, \text{DISABLED}\}$,
- (b) the value function μ maps each non-source attribute A with state VALUE into the value returned by the task producing A and maps each non-source attribute with state DISABLED into the null value \perp , and
- (c) non-source attributes A is in state VALUE if the enabling condition C_A evaluates to true (using the values given for attributes occurring in C_A), and is in state DISABLED otherwise.

The acyclicity assumption guarantees that there is a unique complete snapshot for given source attribute values. An execution of a decision flow instance is *correct* if it produces states and values for the set of target attributes, and is compatible with the unique complete snapshot. (The states and values produced or not produced for other attributes are viewed as irrelevant.)

In this paper we assume that for each given instance of the decision flow, the data needed by the database queries to compute the attribute values remains fixed during the processing of this decision flow instance. This assumption is reasonable for near-realtime decisions in e-commerce applications. This assumption permits flexibility in the timing of launching queries and the use of speculative execution.

Snapshots can provide a basis for reporting on the behavior of a decision flow. In particular, a (possibly nested) relation can be formed, where each tuple is the snapshot of one execution of the decision flow. Attributes concerning the success or failure of the decision can be incorporated. Manual and automated data mining techniques can be performed on this relation, to discover possible refinements to the decision flow.

3. An Execution Model for Decision Flows

This section presents the execution model for decision flows and the architecture of the execution module. The key

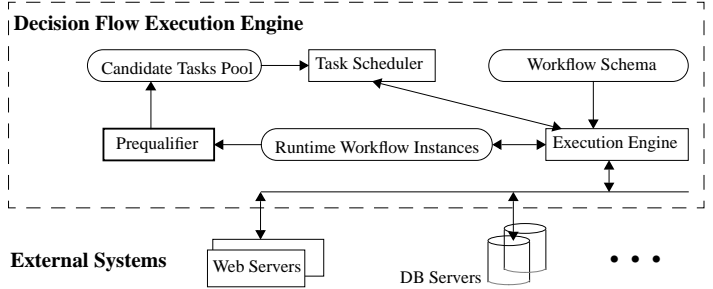


Figure 2. Architecture of execution module

feature of this execution model is its flexible scheduling of parallel executions of the tasks in the decision flow.

Architecture of the execution module. Figure 2 shows the architecture of the execution module. The three round boxes represent data repositories. One contains *decision flow schemas*, and another contains *runtime flow instances* of the decision flows. Whenever a new case, e.g. a new set of promos for a web page needs to be generated, a new flow instance is created. The rectangles represent software modules. The *execution engine* works on the decision flow instances to execute the tasks in the decision flow and propagate the effects of the executions until the goal is reached. The engine works in a multi-thread fashion, so that parallel processing of multiple flow instances, and multiple tasks within one instance is possible. To execute the tasks, the engine consults the *task scheduler* that dynamically chooses one or more tasks from a pool of candidate tasks, i.e., the round box *candidate tasks pool*. More precisely, there is one pool of tasks per flow instance and the scheduler chooses tasks for each flow instance independently from the other flow instances. The candidate pool is maintained by the *prequalifier*. Recall that we are assuming in this paper that each task computes a single attribute. This means that we can identify each task by the attribute that it produces. Further, we interchangeably refer to execution of a task or evaluation of the corresponding attribute.

The execution algorithm. We now give a sketch of the *execution algorithm*, which summarizes the three important phases of executing decision flows. This algorithm is based on a generalized notion of snapshot, which is described shortly. The execution program is invoked each time a new decision flow instance is initiated, and each time new values of attributes are obtained for a running flow instance.

- (1) Evaluation phase:
 - (a) Construct a new snapshot that incorporates the new attribute value(s).
 - (b) If a terminal snapshot (i.e., all the target attributes are stable) is reached, then exit.
- (2) Prequalifying phase (*prequalifier*): Identify a set of *candidate attributes* in the decision flow that are ready to be evaluated.

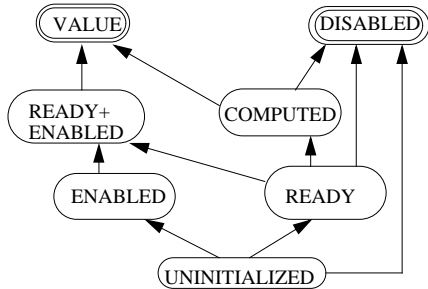


Figure 3. Finite state automata for states of attributes

- (3) Scheduling phase (*scheduler*): Select one or more attributes out of the candidate attribute set based on scheduling heuristics, and send their corresponding queries to the external server(s).

The primary focus here is on optimization techniques used in the prequalifying and scheduling phases.

The execution algorithm constructs a series of snapshots, each one incorporating newly acquired information obtained through the evaluation of attributes. We now describe the extended form of snapshots used. As in Section 2, the extended snapshots will be ordered pairs of form (σ, μ) . However, the set of possible states for attributes is expanded, as indicated in Figure 3. The intuitive meaning of an attribute A being in a state of the fsa is given now.

States UNINITIALIZED, ENABLED, VALUE and DISABLED retain the meaning from Section 2. The states VALUE and DISABLED are shown with double circles because they are terminal states for attributes; when an attribute moves to one of these states then it is stable. An attribute A can move into state ENABLED (DISABLED) if, based on information accumulated so far, the enabling condition for A is determined to have value true (false). An attribute moves from the ENABLED state to the VALUE state as soon its value has been computed. The state READY indicates that all of the input attributes for an attribute have stabilized (i.e., their states are DISABLED or VALUE). If an attribute is in state READY, then it can be evaluated speculatively. State READY+ENABLED indicates both that the input attributes are stable and the enabling condition for an attribute has been determined to be true. State COMPUTED (and not enabled), indicates that the value for A has been computed speculatively but the truth value of the enabling condition is not yet determined. An attribute moves from the COMPUTED state to the VALUE (DISABLED) state as soon its enabling condition is evaluated to true (false). There is a natural partial ordering on the states of the fsa. For example, we write $READY < COMPUTED$.

An execution permitted by the execution algorithm can be described by a sequence of snapshots $(s_0, \dots, s_i, \dots, s_n)$ where s_0 is the initial snapshot (having values only for

the source attributes), $s_i \ i \in [1, n]$, are snapshots computed by the execution algorithm and s_n is the terminal snapshot where target attributes are stable. In [HLS⁺99b], we define sufficient conditions for execution sequences to yield terminal snapshots that are consistent with the declarative semantics defined in §2. We have used these sufficient conditions to prove the correctness of the optimization algorithms presented in the following section.

4. Optimization Strategies

We first state the optimization goals. Then we present our optimization strategies for the prequalifying and scheduling phases of the execution algorithms. In this and the following section we focus exclusively on decision flows where all tasks are database queries. However, the optimizations presented here generalize to other types of tasks, including synthesis tasks and web-queries.

Optimization Goals. Motivated by e-commerce and similar applications, our optimization goal is to be able to guarantee a quality of service in terms of response time whatever the workload conditions are. Thus we need to provide optimization techniques that both minimize (1) the response time and (2) the work performed for the execution of the decision flow instances. The first goal is motivated by the desire to serve web customers as quickly as possible. The second goal is motivated by the fact that e-commerce sites can have bursty load, and can easily become overloaded. It is important to understand trade-offs between time and work, and to be able to gracefully move along those trade-offs depending on current load. For example, given a fixed amount of work that can be performed, what is the best response time possible and how can we obtain it? In §5 we provide answers to this question in two contexts: where the database resource is essentially unlimited, and where it is limited and dedicated to supporting the decision flow.

Optimizations in the Prequalifying Phase. We expand the prequalifying phase of the execution algorithm in §4 with the following two steps: (i) Identify maximal number of eligible attributes; (ii) Eliminate “unnneeded” attributes from the eligible attribute set to get a set of *candidate attributes*.

To obtain a maximal set of candidate attributes with minimal number of unneeded attributes, we use the technique of eager evaluation of enabling conditions. In particular, we perform partial computation of enabling conditions based on the attribute values that are available. As a simple example, in Figure 1 the enabling condition of the node to check coat inventory might be evaluated to false using just the `db_load` attribute. Such reasoning can be used to determine that an attribute is disabled, and hence takes value \perp before the attribute is READY and before all attributes in the enabling condition are stable. Analogous reasoning with disjunctions can determine that an attribute is enabled. This can help to quickly move an attribute to state ENABLED or READY+ENABLED, or from COMPUTED to VALUE.

Another useful activity in decision flow execution is the identification of attributes whose values are *unnneeded* for successful completion of a decision flow instance. This may arise from *forward propagation* of information, i.e., inferring that an attribute is DISABLED by propagating forward the fact that attributes involved in its enabling condition are also DISABLED attributes. Inference of unneeded attributes also arises from *backward propagation*, which involves inferring that although an attribute is or may become enabled, its value is not needed for successful completion of the decision flow instance. As one example, suppose that `expendable_income` is determined to be 0. Then `give_promo(s)?` will be DISABLED and take value \perp . The condition “`give_promo(s)? = true`” is false, and so the five attributes having that as enabling condition will also be DISABLED. As a result, `promo_hit_list` is not needed as input for any enabled attributes. Forward and backward propagation can be combined.

In [HLS+99b] an algorithm, called here *Propagation Algorithm*, is described that performs eager evaluation of enabling conditions and detects unneeded attributes at runtime. The algorithm executes in an incremental fashion, incorporating new information as it becomes available from the execution of the decision flow. Importantly, the cost of executing the algorithm is linear in the size of the decision flow, regardless of what order the tasks are executed in.

Optimizations in the Scheduling Phase. Given a candidate attribute set, we typically need to select a subset of attributes and execute their corresponding database queries, because the underlying database server can support a finite multi-programming level. We focus on two heuristics for scheduling candidate queries.

Topologically-earliest first: Choose the nodes that are topologically-earliest in the dependency graph. This may help identify eligible and unneeded attributes initially using forward propagation, so that many other attributes can become eligible as soon as possible.

Cheapest first: Choose the nodes that have the shortest estimated execution duration. This can have two advantages: (1) The results of these queries will come back faster, so that the corresponding attribute values can be propagated through the graph sooner. (2) In case the executed queries turn out to unneeded later, the wasted time and resources are less compared to more expensive queries.

5. Performance Evaluation

In this section we evaluate the effectiveness of our optimization strategies in a variety of settings. The section begins with a description of our experiment environment. A key component here is a mechanism for generating decision flow schemas having a variety of different “patterns” or characteristics. We then present experiment results that assume the targeted databases have unbounded resources. From this, we develop guidelines that help a potential user

to choose the best optimization strategy for a given application. Last, we consider the more realistic case where resources are bounded, i.e., where database system load has an impact on the performance. We develop a simple analytical model that can be used for tuning performance of decision flows, and experimentally verify its accuracy.

Experiment Environment. We implemented an execution module with all the optimization algorithms and heuristics discussed in §4 built in. The external database server is simulated using CSIM 18, in order to support a fine level of control over various database related parameters, e.g., database load, query cost etc. We use the following three parameters to measure response time and efficiency: (i) *TimeInUnits* : Response time for processing decision flow instance, measured in *units of processing*. This is used in the abstract context of unbounded resources. In actuality, the time it takes for each unit of processing varies depending on system load. (ii) *TimeInSeconds* : At the end of this section we study the context of bounded resources, and thus use this absolute measure of time. (iii) *Work* : The total number of units of processing being performed for each decision flow instance. As a simple example, if one instance takes total ten units of processing and three of the units were processed in parallel, then *TimeInUnits* is 8 and *Work* is 10. In general, *Work* is inversely proportional to the efficiency.

In the experiments we evaluate the execution algorithms with the following four options.

Propagation of information: In §4 we described how forward and backward propagation can identify unneeded attributes. We use the letter ‘P’ (for Propagation) to denote the option where we perform such propagation and do not place unneeded tasks in the candidate pool. Otherwise, we use the letter ‘N’ (for Naive).

Speculative vs. Conservative: We use the letter ‘S’ (for Speculative) to denote the option where the prequalifier selects tasks that are READY+ENABLED or READY for the candidate pool. We use the letter ‘C’ (for Conservative) for the option where the prequalifier selects only tasks that are READY+ENABLED.

Scheduling heuristics option: The scheduling heuristics “topologically-earliest first”, and “cheapest first” are denoted by ‘E’, and ‘C’, respectively.

Parallel processing option: This determines the percentage of the candidate attributes that will be selected for execution. We denote this option as *%Permitted* whose value is between 0 and 100, with the constraint that at least one attribute must be selected for execution. Hence, 0 means that for any given instance, only one attribute can be computed at a time (no parallelism).

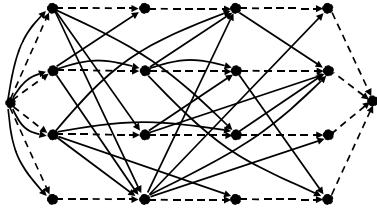
We identify a particular execution *strategy*, i.e., combination of the above options, using a sequence of characters corresponding to these four options. For example, PSE80% denotes the strategy that uses ‘P’, ‘S’, and ‘E’ options with the parallel processing option set to 80%. We use * to rep-

Parameter	Range	Description
nb_nodes	64	# of internal nodes
nb_rows	[1,16]	# of schema rows
%enabled	[10,100]	% of enabled nodes
%enabler	50	% of potential enablers
%enabling_hop	50	max enabling edge hop (as % of total # of columns)
Min_pred	1	min # of predicates per enabling conditions
Max_pred	4	max # of predicates per enabling conditions
%added_data_edges	[-25,+25]	% of data edges added to skeleton
%data_hop	50	max data edge hop (as % of total # of columns)
module_cost	[1,5]	units of cost for executing a module
num_CPUs	4	# of CPUs in the database
num_disks	10	# of disks in the database
unit_CPU_cost	1	# of units of CPU # per execution unit cost
unit_IO_cost	1	# of IO pages per unit execution
%IO_hit	50	probability of IO page hit in buffer
IO_delay	5	IO delay in msecs.

Table 1. Simulation parameters

resent a set of strategies. For example PC** denotes all the strategies that use the 'P' and 'C' options.

We conducted a broad array of experiments based on two basic dimensions: the type of decision flow and the characteristics of the supporting database. Table 1 summarizes the various dimensions that we explored.



Decision flow schema with $nb_nodes=16$, $nb_rows=4$, $\%enabled=50$ and $\%enabling_hop=50$

Figure 4. Example schema pattern

To create a decision flow schema, we first build a data flow *skeleton* based on the parameters nb_nodes and nb_rows . The nodes and dashed edges of Figure 4 shows a skeleton for $nb_nodes = 16$ and $nb_rows = 4$, i.e., 16 internal nodes and 4 rows. The skeleton contains one source attribute, one target attribute, and nb_nodes internal attributes. The number of *columns* in the skeleton is given by $\frac{nb_nodes}{nb_rows}$. In the skeleton, data dependency edges are organized as follows: The source attribute is an input attribute of the first nodes of all the rows. Each internal node is an input attribute of its successor in the same row. The last nodes of all the rows are inputs of the target attribute. By varying nb_rows for a fixed number of nodes, we vary the *diameter* $\frac{nb_nodes}{nb_rows}$ of the schema. Intuitively, the smaller the diameter is, the higher the parallelism can be, assuming everything else being equal.

We now describe how skeletons are used to form schema

patterns Enabling conditions are restricted to conjunctions or disjunctions. The number of predicates in (enabling edges into) each enabling condition is spread uniformly between Min_pred and Max_pred . There are $\%enabler$ attributes whose values are used in at least one enabling condition of another attribute. The maximum hop (as a percentage of the total number of columns in the schema) between the two nodes of an enabling edge is given by the parameter $\%hop$. At the end of the execution $\%enabled$ percent of the enabling conditions will be true. The dashed and solid edges of Figure 4 taken together show a decision flow pattern generated with $nb_node = 16$, $nb_rows = 4$, $\%enabler = 50$, $min_pred = 1$, $max_pred = 4$ and $\%hop = 50$. In an analogous manner we can add (or delete) data flow edges from the skeleton schemas. We experimented with a range of added and deleted data flow edges, but focus here on the case where no data edges are added or deleted from the skeleton. For the experiments, the evaluation of each attribute involves exactly one database query. The cost of the query is set randomly in the range of $module_cost$.

To simulate a database we use a physical model similar to [ACL87] where disks and CPUs are simulated using service queues. The cost of a query is specified in terms of number of units of processing. The cost of a unit of processing on the database is represented by the average number of pages the query accesses and the average CPU time consumed by the query. The last six rows in Table 1 describe the parameters used in our database simulation.

Experiment Results with Infinite Resources. We now address the question of how the optimization strategies perform for decision flows with different patterns. We assume the supporting database has infinite resources and focus on minimizing work performed or minimizing response time measured in units of processing.

Based on our experiments we have found that nb_rows and $\%enabled$ are two key characteristics of a decision flow schema that affect the performance of the optimization strategies. nb_rows controls the diameter of the schema, and has major impact on potential parallelism. $\%enabled$ is a key characteristic of decision flow applications. It indicates the average number of attributes whose values are actually needed to compute the target attributes. This clearly impacts the amount of work that can be saved.

Minimizing the amount of work To study minimizing work we focus on *C*0% because the Conservative option ('C') avoids executing attributes that will become DISABLED, and no parallelism ($\%Permitted = 0\%$) never increases work. We answer the following questions: (i) What are the benefits of using Propagation_Algorithm? (ii) For which decision flow patterns is Propagation_Algorithm is the most efficient? (iii) What scheduling option is performing the least work? (iv) Is the the best scheduling option dependent on decision flow pattern?

Figures 5(a) and 5(b) show the amount of work per-

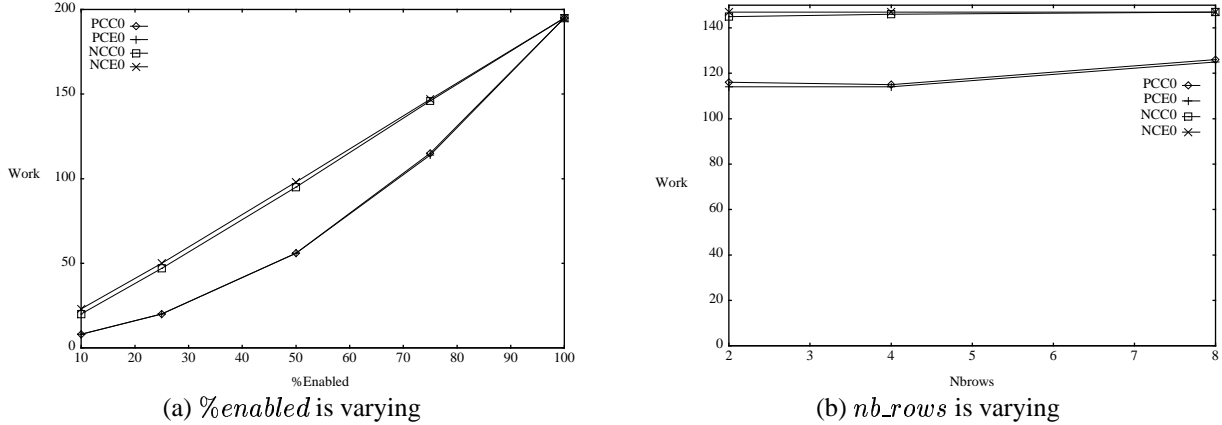


Figure 5. Comparison of strategies for minimizing work

formed by the execution strategies when we vary respectively nb_rows and $\%enabled$. In Figure 5(a) nb_rows is set to 4. (Since there is no parallelism, these figures also show the response time for the different strategies.) This figure shows two clusters of curves based on whether the Propagation_Algorithm (option 'P') is used or not. When it is not used the amount of work performed by programs NC*0% is approximately linear with the percentage of DISABLED attributes ($100 - \%enabled$). Indeed, the C option avoids executing attributes that will become DISABLED. With Propagation_Algorithm, additional unneeded attributes are discovered and the amount of work is reduced further. Figure 5(a) also indicates that decision flows with a large percentage of disabled nodes benefit the most from Propagation_Algorithm. The best benefits are about 60% and are obtained when $\%enabled = 10\%$. For Figure 5(b), $\%enabled$ is set to 75.

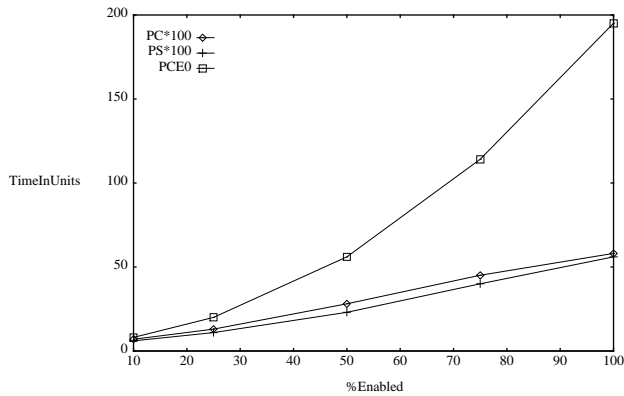
Now we examine the effects of the scheduling heuristics. The performance of the “cheapest” and “earliest” heuristics are very close, within 10% of each other. When the P option is not used, “cheapest” scheduling always gives the best performance. In contrast, “earliest” gives the best benefits when combined with the P option. Indeed, the “earliest” scheduling allows more aggressive forward propagation to detect eligible and DISABLED attributes. Consequently, more forward propagation produces more start points for backward propagation to detect unneeded attributes. This is an interesting result, since in most application areas “cheapest” is the heuristic of choice. We explore this further below.

Minimizing the response time We now turn to minimizing response time. To this end we assume 100% parallelism, and explore the following questions: (i) What benefits in response time can be obtained by combining the Speculative or Conservative strategies with high parallelism? (ii) How do those benefits depend on decision flow patterns? (iii) What is the trade-off between *Work* and *TimeInUnits*, or more precisely how much additional work is necessary to reduce *TimeInUnits*? To address these questions we

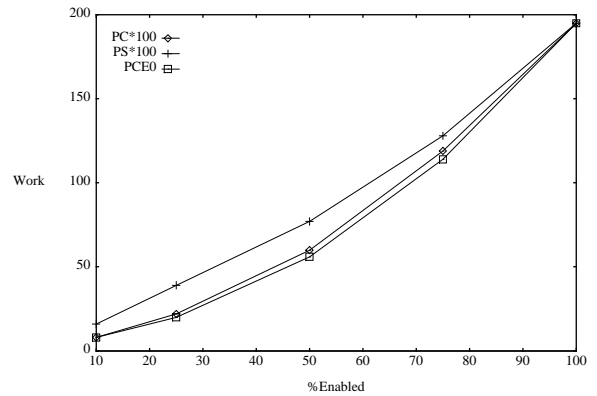
consider programs of the form $P^{**}100\%$ and compare their performance with $PCE0\%$ (which, according to Figure 5, gives the best response time when there is no parallelism). Figure 6(a) shows that using maximal parallelism can reduce significantly the response time. For example, using $PC^{*}100\%$ when $nb_rows = 4$ and $\%enabled = 75\%$ leads to a 60% reduction of response time. This result is not very surprising since parallelism takes advantage of the eager detection of eligible attributes made by Propagation_Algorithm. When using the “Conservative” strategy there is little increase in work. In contrast, Figure 6(a) shows that with the “Speculative” strategy a rather small gain over “Conservative” strategy (maximum of about 10%) can be achieved, but with a significant increase in work (see 6(b)). For example, when $\%enabled = 50\%$, 30% additional work is performed. This is due to the fact that “Speculative” strategy executes many attributes that become DISABLED afterwards. The “Speculative” strategy becomes more profitable when $\%enabled$ is high, because the increase in work is lower.

Effect of “Earliest” and “Cheapest” Scheduling heuristics. We now consider the impact of varying the degree of parallelism on the performance of the optimizations. To do such comparison we consider programs of the form PCE^{*} , PSE^{*} , PCC^{*} PSC^{*} and vary the degree of parallelism.

Figures 7(a) and (b) show respectively the values of *TimeInUnits* and *work* for various degrees of parallelism when the programs are executed on a decision flow schema with $nb_rows = 4$ and $\%enabled = 75\%$. Figure 7(a) shows that “Earliest” scheduling always gives a shorter response time than the “Cheapest” scheduling. The best gains are obtained when the degree of parallelism is between 40 and 80. When “Conservative” (C) strategy is used, the gain of using Earliest scheduling compared to Cheapest is about 15% when the parallelism is equal to 40. It is 30% when “Speculative” (S) strategy is used. Moreover Figure 7(b) shows that Earliest and Cheapest heuristics consume approximately the same amount of work. There are

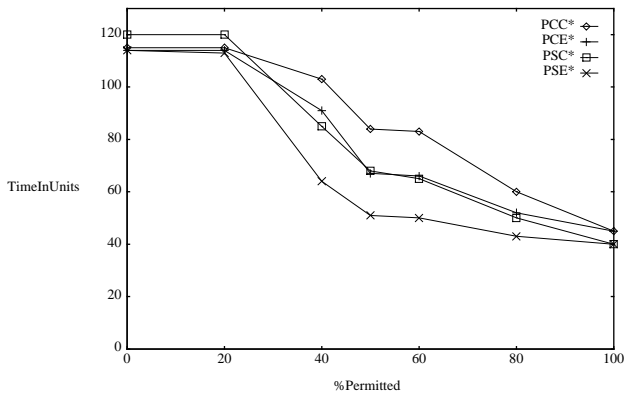


(a) %enabled vs. response time

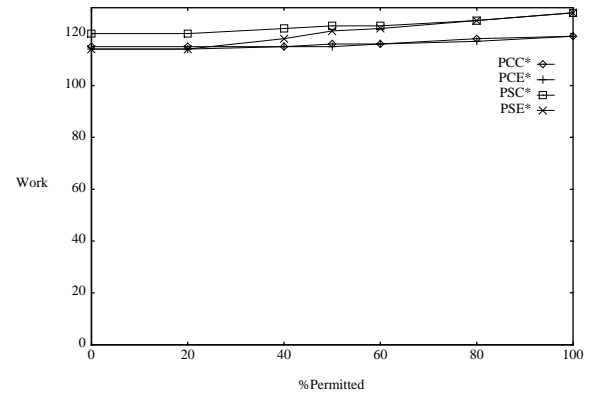


(b) %enabled vs. work

Figure 6. Comparison of strategies for minimizing response time



(a) Parallelism vs. response time



(b) Parallelism vs. work

Figure 7. Effect of different levels of parallelism

two reasons why “Earliest” heuristics is more profitable. First by choosing attributes closer to the source the earliest strategy enhances the effects of forward propagation, and consequently the effects of backward propagation (new start points for backward propagation are discovered). As a consequence, eligible and unneeded attributes are discovered earlier. The second reason is the fact that an attribute that is close to the source will be discovered DISABLED (or ENABLED) sooner than an attribute close to the target. By choosing attributes that are close to the source when the “Speculative” strategy is used, the Earliest scheduling limits the risk of choosing an attribute that will become DISABLED afterwards. This explains the additional gain of using “Earliest” compared to “Cheapest” (30% instead of 15%) when “Speculative” (S) strategy is used.

Lessons learned and guideline maps. Our experiments suggest the following guidelines.

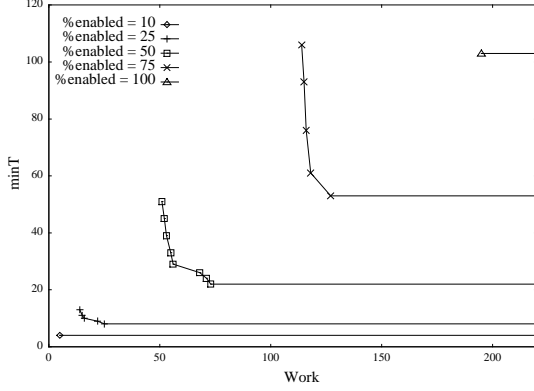
Lesson 1: Using option ‘P’ (Propagation_Algorithm) reduces both response time and amount of performed work. The benefits obtained using option ‘P’ are most significant when the proportion of potential DISABLED nodes per in-

stance is large (more than 20%).

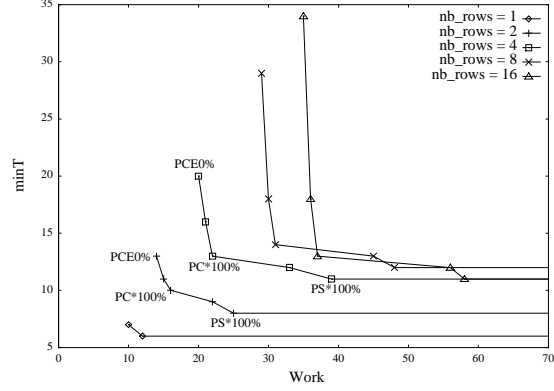
Lesson 2: When option ‘P’ is used, Conservative strategy (option ‘C’) is usually more profitable than the Speculative strategy (option ‘S’). The Speculative strategy becomes more profitable when the proportion of DISABLED nodes per instance is low (less than 25%).

Lesson 3: When option ‘P’ is used, the “Earliest” scheduling heuristic is the most profitable. The fact that “Earliest” out-performs “Cheapest” is significant for two reasons: (a) “Earliest” is simpler and easier to implement than “Cheapest”, and (b) for most applications studied in the literature, the “Cheapest” heuristics wins over “Earliest”.

Our experiment platform can be used to predict the performance of a particular decision flow schema during the design phase of the decision flow. Figure 8 shows the kind of guideline maps our tool can produce. The curves show the minimal *TimeInUnits* that can be obtained for a given bound on *Work*. It also suggests a execution strategy for that bound. For example, in Figure 8(b) for a work limit of 40 units, the minimal response time can be obtained with PS*100% when the schema pattern has 2 or 4 rows. The expected response times are 8 and 12 units, respectively. Fi-



(a) %enabled is varying



(b) nb_rows is varying

Figure 8. Guideline maps: $minT$ vs. corresponding $Work$ for various implementations

nally, no implementation can guarantee a work limit of 25 units with schemas of 8 rows. Figure 8(b) shows similar guidelines when the proportion of enabled nodes is varying.

An Analytical Model for Finite Database Resources. In the previous paragraph work performed and response time are measured in units of processing. Now we consider decision flows (typically E-commerce applications) that use dedicated databases, where the load imposed by the decision flow processing on the databases is the dominant factor impacting the response time for processing queries. In this section we provide an analytical model that can be used to determine (i) given a targeted throughput what is the maximal amount of work the decision flow can afford, and (ii) given this limit what is the strategy that gives the best response time (in seconds).

We first present our analytical model, then tuning prescriptions, and finally experimental verification.

Analytical model We develop an equation that characterizes the relationship of the response time for performing a unit of processing vs. the number of decision flow instances being processed and the amount of work being done for each one. In the development, we simplify by assuming that the decision flow accesses a single database; we expect that the results will generalize in a natural fashion.

We begin by introducing some key variables. In addition to $Work$ and $TimeInUnits$ defined above we use the following variables: (i) Th : (Throughput) the number of decision flow instances that are being processed per second. (ii) $Lmpl$: The average multi-programming level (or number of queries being executed in parallel) for each decision flow instance. (iii) $Impl$: Number of decision flow instances processing in parallel. (iv) $Gmpl$: The multi-programming level of the database (or number of units of processing being executed in parallel on the database). (v) $TimeInSeconds$: Response time (on average) for processing a decision flow instance, measured in seconds. (vi) $UnitTime$: The response time of the database to perform a unit of processing. (vii)

Db : The function mapping the multi-programming level of the database to the response time of the database per unit of processing. This is empirically determined for each database; the graph of Db for our experimental database is shown in Figure 9(a).

The following (straightforward) equations describe the relationships between these variables when the execution is in a stable state. Here Equations (2), (3) and (4) can be combined to obtain (5), and (1) and (5) yield (6).

$$UnitTime = Db(Gmpl) \quad (1)$$

$$Impl = Th \times TimeInSeconds \quad (2)$$

$$TimeInSeconds = TimeInUnits \times UnitTime \quad (3)$$

$$TimeInUnits = Work / Lmpl \quad (4)$$

$$Gmpl = Impl \times Lmpl \quad (5)$$

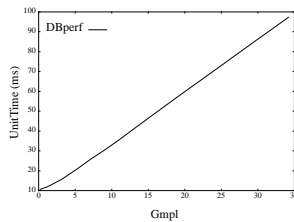
$$= (Th \times TimeInSeconds) \times Lmpl$$

$$= (Th \times Work \times UnitTime)$$

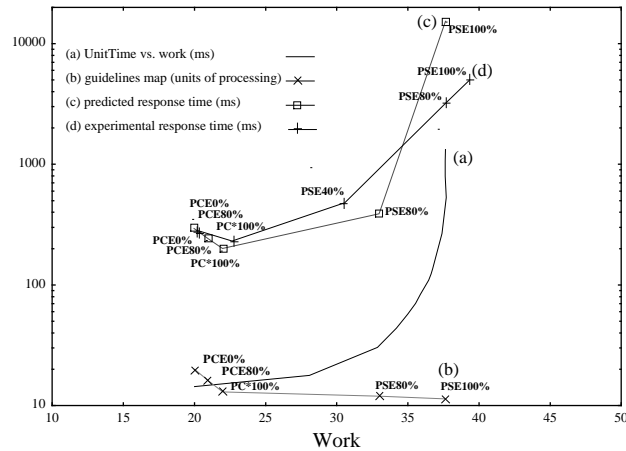
$$UnitTime = Db(Th \times Work \times UnitTime) \quad (6)$$

Prescriptions for Tuning Equation (6) can be applied in two ways. First, this equation indicates, for a given throughput Th , an upper bound on the amount of work, measured in units of processing, that can be performed for each decision flow instance. In particular, this upper bound is the maximum value for $Work$ such that Equation (6) has a solution. For example, using the function Db of Figure 9(a) and a throughput of 20 instances per second, this upper bound is 19 units. The upper bound on $Work$ can be used in conjunction with the guidelines maps given in Figure 8 to determine whether a given throughput can be supported at all. For example, using the guidelines in Figure 8(b) we can see that only schemas with nb_rows equal to 2 and 4 can support a throughput of 20 instances per second without delays.

Suppose now that the database can support a given throughput. The second application of Equation (6) is to choose the execution strategy that will minimize overall response time. We illustrate this application now, assuming the schema pattern in Figure 8(b) with nb_rows equal to 4. Assume that a given throughput level is given, and can



(a) *UnitTime* vs. *Gmpl* for our experimental DB



(b) Finding most effective execution program

Figure 9. Graphs illustrating accuracy of analytic model

be supported by the database. Then graphs such as those shown in Figure 9(b) can be used to choose the best execution program. The graph (a) in Figure 9(b) shows, for a fixed throughput, the relationship between different values for *Work* and the value of *UnitTime*, based on Equation (6). The graph (b) in Figure 9(b) shows the guidelines map from Figure 8(b) that gives for a value of *Work* the minimal response time that can be obtained and the program that should be used to obtain it. Finally, the graph (c) in Figure 9(b) combines the other two graphs using multiplication, to show the mapping from *Work* to *TimeInSeconds*, i.e., the predicted response time per instance. For this particular throughput, we conclude that the optimal response time is obtained by the programs *PC*100%* and its value is 220 milliseconds (We are using a log scale for the *y* axis.)

We verified this analysis experimentally. The graph (d) in Figure 9(b) shows the (average) response time measured when instances of the corresponding schema are executed against the experimental database with an arrival rate of 10 instances per second. We can see that *PC*100%* gives the best response time. Moreover the prediction on the response time was quite accurate (less than 10% error).

6. Conclusions

This paper provides an initial exploration of optimizing data-intensive decision flows, which can be used in a variety of e-commerce and other business applications. Specific properties of decision flows permit a variety of optimization strategies. This paper shows the value of detection of eligible and unneeded attributes using forward and backward propagation, and studies trade-offs between response time and work performed in various contexts. The results of the paper can be used to tune an execution engine to minimize response time and work performed as the overall workload of the system changes through time.

A variety of questions are raised by this study, e.g., how to optimize when several decision flows will be executed based on overlapping data, whether queries from one or several decision flows should be clustered to reduce overall database access time, and how best to incorporate this technology into existing workflow systems.

References

- [ACL87] R. Agrawal, M.J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. on Database Systems*, 12(4):609–654, 1987.
- [BKK87] J. Bein, R. King, and N. Kamel. Moby: An architecture for distributed expert database systems. *Proc. of Intl. Conf. on Very Large Data Bases*, pp. 13–20, 1987.
- [HLS⁺99a] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proc. of Intl. Joint Conf. on Work Activities Coordination and Collaboration (WACC)*, pages 69–78, February 1999.
- [HLS⁺99b] R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar, and G. Zhou. Efficient support for decision flows in e-commerce applications (long version). Technical report, Bell Laboratories, Lucent Technologies, Murray Hill, NJ, 1999. <http://www-db.research.bell-labs.com/projects/vortex/ictec-full.ps>.
- [KS95] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-systems tasks to support enterprise-wide operations. *Dist. and Parallel Databases*, 3(2), 1995.
- [LR94] F. Leymann and D. Roller. Business process management with FlowMark. In *Proc. of IEEE Computer Conf.*, 1994.
- [PHG96] D. Patterson, J. Hennessy, and D. Goldberg. *Computer Architecture : A Quantitative Approach*. Morgan-Kaufmann Publishers, Inc., 1996.
- [PR93] D. Peppers and M. Rogers. *The One to One Future*. Doubleday, New York, 1993.
- [PYLS96] P. Piatko, R. Yangarber, D. Lin, and D. Shasha. Thinksheet: A tool for tailoring complex documents. *Proc. ACM SIGMOD Symp. on the Management of Data*, 1996.