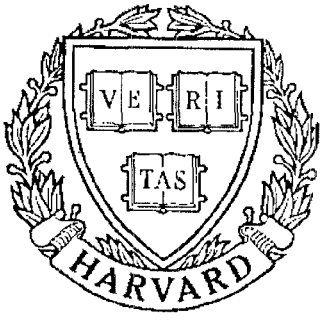
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

Building Decision Support Systems That Use Operations Research Models as Database Applications

by M.O. Ball, A. Datta, and R. Dahl

TR 92-109

Building Decision Support Systems That Use Operations Research Models as Database Applications

Michael O. Ball
College of Business and Management
and Systems Research Center
University of Maryland
College Park, MD 20742
and
Department of Operations Research
University of North Carolina
CB#3180, 210 Smith Bldg.
Chapel Hill, NC 27516

Anindya Datta
College of Business and Management
and Systems Research Center
University of Maryland
College Park, MD 20742

Roy Dahl
DISTINCT Management Consultants
10705 Charter Dr., Ste. 440
Columbia, MD 21044

October 1, 1992

Abstract

In this paper we address the problem of building decision support systems that make use of multiple operations research models as database applications. The motivation for developing applications in a database environment is that, by doing so, the development effort can be substantially reduced, while, at the same time, the application inherits valuable database features. The paper contains two main contributions. First, we present a set of modeling constructs that should aid developers in structuring such applications and in carrying out the development process. Included in this material is a fairly comprehensive model for handling versions. Second, we discuss certain design alternatives and evaluate performance tradeoffs associated with them. In addition, to evaluating the differences among competing database designs, we provide evidence that properly designed database applications, show little performance degradation over file based applications.

1 Introduction

The aim of this paper is to propose and analyze a database environment for the development of decision support systems that make use of multiple operations research (OR) models. In many environments supported by OR models, the solution approaches used involve multiple algorithmic steps. That is, an algorithm generates an intermediate result which serves as the input to a second algorithm the result of which is fed into a third and so on. For example, in the vehicle routing problem, the generalized assignment approach [10] involves three algorithmic steps: the first chooses a set of seed points, one for each vehicle; the second assigns customers to vehicles; and the third generates delivery sequences for each vehicle. Typical decision making in such environments involves assessing the quality of intermediate solutions, making changes, rerunning algorithms with altered parameters, etc. In the vehicle routing problem, once the customer to vehicle assignment step is over, the user might desire to change a few assignments manually or, alternatively, rerun the the assignment algorithm with different parameter settings. The ability to carry out such operations easily and to keep track of the alternate solutions generated is an important system requirement.

In many systems the user has the option of choosing between one of two algorithms to generate a particular solution or of choosing between an algorithm and a manually generated solution. Such systems should provide the user with the ability to maintain alternate solutions and to compare such solutions. Another typical requirement, is the ability to break large problems into chunks and then to solve a smaller problem over each chunk. In solving a vehicle routing problem over a large geographic area it is quite conceivable that the user would break the entire region into small subregions. This, of course mandates operations that can isolate chunks of data, perform consistency checks and prepare it for use by the algorithm.

A careful perusal of the discussion above helps one identify certain basic requirements of any system designed to support the environment outlined at the outset. A common theme of these requirements is the manipulation of data in several forms, namely modifications, insertions, consistency checking and so on. A Database Management System (DBMS) offers an interface where such operations are performed naturally and provides a platform where system development can proceed unencumbered by data manipulation technicalities. It is also worth mentioning at this point that a major factor which limits the widespread use and acceptance of operations research models is the large overhead associated with both the development of optimization algorithms and the development of complete systems that provides users with flexible access to algorithms. Clearly, an approach that has the potential for reducing this overhead is the coupling of operations research models with DBMSs. This research aims at studying the implications and issues concerning the problem of embedding optimization algorithms in DBMSs.

Since their inception, DBMSs have been concerned with information storage and retrieval and consequently one observes the growth and development of Data Definition Languages (DDL) and Data Manipulation Languages (DML) focussed around these issues. Most current research in database systems either deal with better knowledge representation techniques, e.g. encapsulated data objects, abstract data objects, etc., or with the confluence of knowledge and databases, e.g. intelligent databases, etc. Our view is that relatively little research has addressed programming techniques, i.e. the areas of more efficient use of existing operators and adding additional functionality through new operators.

Classical problem solving techniques such as computing shortest paths, spanning trees and travelling salesmen's routes still deal with data in terms of file structures or at best, arrays and pointer structures. In contrast, DBMSs provide a smooth and relatively transparent interface to the data enabling development effort to proceed unencumbered by the complexities of structure manipulation. We and others have developed systems with algorithms implemented in a programming language such as C or FORTRAN interfacing with information stored in DBMSs as opposed to data recorded in files. Our experience indicates that DBMSs usually provide slower data manipulation because of the prohibitive maintenance overhead involved with DDLs and DMLs. However, the use the DBMSs provide some very distinct advantages:

1. Algorithm development time is reduced substantially due to the ability to use high level languages such as SQL to present data to the algorithm in a streamlined form.

2. The resultant overall system has all the features provided by DBMSs, including integrity preservation, security checking, ad hoc access to data, etc.

3. The time required to develop an interactive system surrounding the algorithms is reduced substantially, through the use of DBMS application development tools.

In spite of these advantages it appears that few people are willing to implement optimization based systems (or for that matter any update intensive operation) on a database platform. It seems clear that the primary reason for this is the supposed performance degradation. In the subsequent sections we discuss certain issues associated with the process of building optimization based decision making systems around DBMSs and approaches to solving them. One result we present is that a properly designed database application can be highly competitive, in terms of performance, with a file based application. The basic concepts we discuss are general in that they could apply to any data model, e.g. relational, object oriented, etc. However, most of our specific proposals, models and examples apply to relational DBMSs.

1.1 Literature Review

Surprisingly, there is very little reported in the literature about interfacing databases with OR models. The only paper we found that directly deals with the issue is the paper by Dahl et. al. [7]. However the authors deal with a specific application, transit vehicle and crew scheduling, and suggest a scheme to represent problem information in a DBMS, whereas our paper is concerned with the much more general problem of embedding algorithms generically in a database environment and issues concerning that. We note that the potential advantage of using DBMSs in conjunction with OR models has been envisioned by Geoffrion [12] in his paper on structured modelling where he states, "Database systems are natural adjuncts to data hungry MS/OR software. Data management and flexible retrieval capacity are just as important for most MS/OR applications as the functions performed by the solvers toward which the models are oriented".

In spite of the paucity of reported research on the commonality of OR and databases, the issues that we identify are not unique to OR problems and have been dealt with, some more extensively than others, by researchers. In particular, the two issues of retrieval/insert and version management have received widespread recognition as critical concerns in database design and have been looked at by a several researchers. The groups that we most closely identify with, in terms of analogous themes of research, are the CAD/CAM and engineering database groups, who deal with issues very similar to ours.

Retrieval/insert problems are explored in the classic Aho and Ullman paper [2]. Other works that deal with similar problems are Ghosh [13], Batory [5], March [16]. However, these papers deal strictly from a database perspective, the objective being "good database representation". In our case however, the primary purpose is to manipulate a DBMS to effectively integrate optimization models and in the process we sometimes violate classical database design rules in the interest of pragmatism and efficiency.

Techniques for handling version management receive excellent treatment in Kim and Batory [15] which deals with model and storage techniques for versions for VLSI CAD objects. An excellent survey of existing versioning schemes appears in [20]. The first papers to address database support for versions appeared in the early 1980s. Since then numerous version management models have been proposed [6, 15], "*but comprehensive frameworks for understanding version semantics are still absent*" [20]. Other research reports of versions may be found in Plouffe et al [18], Bjornerstedt and Hulten [6].

Our primary orientation is to determine how to best make use of existing database capabilities. Another more prevalent research point of view is to enhance RDBMSs' support of innovative applications. Such database systems are called extensible or extended databases. One of the first papers in this area was by Abiteboul et. al. [1]. Other examples of currently ongoing research in extensible databases are the STARBURST system being built at IBM and GENESIS. A report on STARBURST is in [14]. The introductory GENESIS

paper is [13].

1.2 Sample Applications

In this section we describe three sample applications:

i. Vehicle Routing

ii. Transit Crew and Vehicle Scheduling

iii. Telecommunications Network Design

We have had experience with the development of optimization based systems for all of these application areas. For i.) and iii.) we have developed systems based on relational databases and for ii.) we have carried out a detailed design for a system based on a relational database [7]. The detailed analysis presented in the following sections applies specifically to the vehicle routing application. Part of our objective in presenting this subsection is to illustrate the generality of the this analysis to other application settings as well.

For the database environment we envision that all relevant problem data is stored in the database. Specifically, both the inputs to algorithms as well as their outputs are stored in the database. We also are oriented to systems that employ multiple algorithmic steps as well as user interaction. Fisher [9] gives a good discussion of the structure and overall design philosophy of such systems. A typical problem solving scenario might be:

1. user invokes algorithm 1;
2. user modifies solution generated by algorithm 1;
3. user invokes algorithm 2;
4. user modifies algorithm 2 parameters and invokes algorithm 2, overwriting previous solution;
5. user invokes algorithm 3.

Some important characteristics of the above scenario are that the solution generated by an algorithm might be modified by the user, the input to one algorithm might involve the output from another and an algorithm might be invoked multiple times.

1.2.1 Vehicle Routing

The databases we are concerned with contain a mixture of static and volatile information. In vehicle routing schema, given in Figure 1, the database contains static information about customers, such as customer location, which is stored in the table *customers* and static information about vehicles, such as vehicle

customers (**cust_id**, cust_name, c_node_id, billing_add)

orders (**order_id**, cust_id, amount, veh_id, seq_no)

vehicles (**veh_id**, type, capacity, s_order_id)

nodes (**node_id**, coordinate)

arcs (**arc_id**, node_id1, node_id2, tr_time)

Figure 1: Vehicle Routing Schema

capacity, which is stored in the *vehicles* table. A static transportation network upon which all travel time calculations are based, is stored in the tables *nodes* and *arcs*. An order is a specification of an amount that must be delivered to a customer. Order information, which is volatile, e.g. it might change daily, is stored in the *orders* table. The function of the optimization algorithms is to generate vehicle routes that satisfy all orders. Since the routes depend on the orders, route information is also volatile. The route information is stored in the attributes orders.veh_id, orders.seq_no and vehicles.seed. We should note that a primary issue that we will analyze is how the output from algorithms should be stored. Thus, the schema presented in Figure 1 is one of several alternatives that we will analyze (primary keys in bold). The generalized assignment approach as applied in this setting has the following steps:

a. *Seed Point Selection:* An order, called the seed, is associated with each vehicle. The clustering step will use the seed as a geographic point of attraction for the route. The result of this step is to insert values into the vehicles.s_order_id attribute.

b. *Order Clustering:* This step assigns orders to vehicles. The assignment is carried out considering vehicle capacity restrictions with the objective of minimizing travel time deviation between each order and the seed associated with the vehicle to which it is assigned. The result of this step is to insert values into the orders.veh_id attribute.

c. *Sequencing:* This step determines a delivery sequence for each route. The result of this step is to insert values into the orders.seq_no attribute.

Some typical intermediate user interactions might be to rerun the seed point selection step with altered parameters to force the use of fewer vehicles or to alter certain order-to-vehicle assignments after the order clustering step.

trips (**trip_id**, start_time, start_loc, end_time, end_loc, block_id, seq_no)

blocks (**block_id**, tot_time)

pieces (**piece_id**, block_id, st_bk_seq_no, st_tr_seq_no, end_bk_seq_no, end_tr_seq_no)

runs (**piece_id**, seq_no, run_id)

Figure 2: Transit Crew and Vehicle Scheduling Schema

1.2.2 Transit Crew and Vehicle Scheduling

The fundamental entities associated with transit scheduling systems include i. *routes* which specify the sequence of stops for each transit line, ii. *timetables* which specify the times at which vehicles visit the stops on each line, iii. *vehicle* blocks which describe the complete itinerary for vehicles over the course of a day and iv. *crew runs*, which specify the complete itinerary for crews over the course of a day. We now present a schema for a simplified version of the transit crew and vehicle scheduling problem. This schema addresses the decision making process that arises after routes and timetables have been specified, i.e. the determination of vehicle blocks and crew runs. In order to address these problems the route and timetable data is usually first summarized in terms of a set of *trips* . Each trip represents a one-way traversal of a transit line. Each trip is characterized by a start location and start time and an end location and end time. A vehicle block is a set trips between one exit from and entry to the vehicle depot. Since the vehicle blocks may be longer in time than a single driver can work, for the purposes of crew scheduling, they are broken in pieces which represent a portion of a crew workday (most typically a piece is 1/2 of a crew workday). Thus, a single crew workday (run) consists of one or more pieces.

The partial schema given in Figure 2 uses trips as a starting point and includes the data necessary to define the vehicle and crew schedules defined based on the trips. A typical three step approach to generating crew and vehicle schedules is (see Ball and Benoit [3]):

a. *Block formation:* Sequences of trips are combined into blocks so as to minimize a combination of excess travel time and waiting time. The result of this step is to insert rows into the blocks table and to assign values to the attributes trips.block_id and trips.seq_no.

b. *Piece Formation:* Based on various work rules and crew pay considerations, the blocks are broken into pieces. Pieces typically represent an entire crew workday or 1/2 a crew workday. Each piece must start and

end at a feasible relief stop. The result of this step is to insert rows into the pieces table.

c. *Run Generation:* Pieces are combined into crew runs. The objective function of this step is to minimize total crew paid hours. The runs must adhere to various crew work rule restrictions. The result of this step is to insert rows into the runs table.

Many computer systems that support transit scheduling include graphics based interactive modules that allow block modification after step a. Also, it is typical for step b. to be rerun after viewing the results of step c. In fact, [3] describes a formal feedback loop between steps b. and c.

1.2.3 Packet Network Design

The packet network design problem is to produce a telecommunications network design which can transport projected traffic loads between a set of terminals and host computers. The output must specify locations and configurations for switching equipment as well as determine which switches should be directly connected. Switches are connected by dedicated links that have a complex tariff structure. There are many types of switching equipment, but for design purposes they can be categorized into local and backbone switch types. Backbone switches are large expensive devices that are used to form the core of a large network. Local switches, including concentrators, which technically are not switches, form the basis of the local access network, which carries traffic between the source terminal or host and the backbone network. The design problem considers cost, delay, and reliability as the primary objectives. The locations for switches are usually chosen from a limited number of possible locations.

An accurate schema for the complete packet network design problem is very complex. A partial schema that captures important aspects of the problem is given in Figure 3. Most practical approaches to this problem break the analysis into two separate steps . These steps are local access and backbone network design. However, these problems are not independent in that the local access design does effect the volume of traffic on the backbone network as well as the selection and number of backbone switch locations. Cheaper and more reliable local access networks are possible with more backbone switch locations, but more backbone switch locations involve more costly backbone network designs. The important point is that trade-offs between the designs need to be analyzed. See Monma and Sheng [17] and Gavish and Altinkemer [11] for descriptions of optimization algorithms and related complete systems for the solutions of these problems.

A typical multi-step approach for generating a network design, together with related performance information is:

terms (**term_id**, loc)

term_traff (**from_term, to_term**, traffic)

concentrators (**concen_id**, num_switches, loc, cost, traff_load, delay)

access_links (**from_loc, to_loc**, size, traff_load, cost)

back_traff (**from_node, to_node**, traffic, delay)

backbones (**backbone_id**, num_switches, loc, size, cost)

back_links (**from_loc, to_loc**, size, traff_load, cost)

Figure 3: Packet Network Design Schema

a. *Backbone switch location:* Based on either earlier local access designs or estimations derived from terminal traffic, a set of backbone switch locations are selected. The result of this step is to insert rows into the backbones table. Only the attributes backbones.backbone_id and backbones.loc are given values at this time.

b. *Local access link and concentrator switch selection:* This step locates concentrators and connects terminals to concentrators and concentrators to backbone switch locations. The result of this step is to insert rows into the concentrators and access_links tables. Only the attributes concentrators.concentrator_id, concentrators.loc, access_links.from_loc and access_links.to_loc are given values at this time.

c. *Local access analysis and Configuration:* This step determines the resulting traffic for the backbone network as well as calculating delay and other statistics. The result of this step is to insert rows into the back_traff table and to assign values backbones.traff_load attribute and to all attributes in the access_link and concentrators tables that were not given values in step b.

d. *Backbone design:* Based on the backbone locations selected in local access design along with the resulting traffic this routine determines which switches should be directly connected by dedicated links and the size of each of the links. The major considerations include cost, connectivity, reliability, and delay. The result of this step is to insert rows into the back_links table.

e. *Backbone Analysis and Configuration:* Based on traffic through switches, the size, type and number of switches is determined for each backbone

node. To accomplish this a backbone traffic routing algorithm must be invoked. The result of this step is to update the values in the backbone.size, backbone.num_switches and backbone.cost attributes.

One aspect of the problem solving setting just described that is different from the previous two is the use of analysis routines. Here, by analysis routine we mean a procedure that does not solve a decision problem, but rather generates performance information. This information is typically taken into account by the user in deciding whether to accept a proposed solution or to modify it by rerunning algorithms or by making manual changes.

2 Problems and Solutions

In this section we describe some of the fundamental problems that motivated our work and summarize the main contributions of this paper. The final section of this paper describes additional problems that we feel are worthy of future study.

2.1 Versioning and Algorithmic Flow Control

Conventional database systems work in terms of a single evolving logical state. Transactions are used to alter the database from one consistent state to the next. As soon as the transaction commits, the previous state is discarded for all practical purposes. The system has no memory with respect to prior states of the database. However, many application areas now, or in the future, supported by DBMSs cannot be modelled adequately by representing only the present state of the world. Instead it must be possible to model past and future states as well as states existing in parallel with each other.

In designing optimization applications, the notion of versions takes on special significance. Versions may be used for various purposes, e.g. to illustrate the life cycle of a design object, to document the process of development or to represent variants in the process of development. Some of the most obvious reasons motivating version control in a database supported optimization application design environment are:

1. Often, there exist multiple algorithms or heuristics to solve a problem. It is reasonable to assume therefore, that users would want to compare the results of the application of different algorithms to the same data set by comparing their outputs.

2. Often, users want to modify an automatically generated solution. The user might achieve this by employing the system's interactive features to alter the solution. Alternatively, the user might rerun an algorithm using a different parameter setting. Afterwards the user would typically examine

and compare all solutions generated prior to making a final decision on one of them.

3. In a multi-user environment, it is quite possible that two users may generate alternate solutions to the same problem. These solutions could be stored as versions and later analysis could be used to choose between the two solutions.

Controlling the algorithmic flow is especially important in systems where multiple steps are required to solve a problem, and where the various steps require the application of different algorithms. In such environments, while providing the user enough flexibility for problem solving, it is crucial to ensure that the user does not perform operations that lead to erroneous results. The complexity in achieving these goals arises from the need to provide the user with the ability to interactively alter solutions and to rerun algorithms after changing parameters. Also, it should be noted that the required input data for one algorithm may have been put into place through the running of a different algorithm, or through the application of a procedure that may have nothing to do with optimization. It is quite conceivable that the user may wish to modify data or parameters that can potentially render the information for the next algorithm run in an inconsistent state. The system features required would:

1. generally keep the user apprised with respect to progress toward an overall solution,

2. inform the user as to which algorithmic steps are currently feasible,

3. warn the user when certain actions will render the database in an infeasible state.

Thus, proper handling of versions and flow control in algorithms are closely tied. One of the major causes of incorrect flow is inadequate versioning methodologies. When a number of algorithms using the same set of input relations return different versions of the same object (e.g. a number of different routes given identical networks and customer requirements) each of those objects, typically would be meant to be input into other algorithms. If the user, desiring to modify object Oj, mistakenly updates Oi, a flow control problem has occurred because all subsequent algorithm runs will be out of sync. However, it is precisely the version control scheme that is supposed to oversee the prevention of such mistakes, and in the process, maintain feasibility of database states, insuring correct flow.

2.2 Retrieval and Insert Problems

Viewed in its most elementary form, the relationship between the database and an optimization or analysis algorithm is that the algorithm retrieves its input

nodes (**node_id**, coordinate)

arcs (**arc_id**, node_id1, node_id2, tr_time)

customers (**cust_id**, cust_name, c_node_id, billing_add)

orders (**order_id**, cust_id, amount)

vehicles (**veh_id**, type, capacity)

seeds (**veh_id**, s_order_id)

clusters (**order_id**, veh_id)

routes (**order_id**, seq_no)

Figure 4: Alternate Vehicle Routing Schema

data from the database, performs its required function and then inserts its output into the database. It is our view that the principal impediment toward more widespread use of databases for the delivery of OR models is the relative inefficiency or perceived inefficiency of these retrieval and insert operations. We have found that the efficiency of these operations has a heavy dependence on both the logical and physical database design and the specific database operators used.

For most commercial database systems alternatives exist for inserting records into a database and for updating records in a database. Striking performance differences exist between some of these alternatives. We have found that transaction logging can substantially slow down many operations. On the other hand, in many situations the benefits of logging can be achieved by simple application level controls. One general problem faced by applications designers is: which operations should be logged and which should be non-logged and, directly related to this is which combination of database operators should be used for certain operations.

A second issue relates to the logical database design. Figure 4, provides an alternative to the VRP design given in Figure 1.

This alternative design effectively dedicates a table to the output of each algorithmic step. Such designs have certain performance advantages which we will explore later. However, they are less natural and may be more cumbersome for the user to interact with.

2.3 Contributions of this Paper

The principal contributions of this paper can be viewed as solutions to the problems described above. In addition, we describe certain techniques that are of use in the overall design process. Our intent is that the methods proposed in this paper should be used by developers of decision support systems of the type we have described. Such systems would make use of a commercial database product such as ORACLE or Sybase, high level languages such as C or FORTRAN (primarily for the optimization algorithms, but possibly also to develop parts of the interactive interface) and application development tools, e.g. forms packages, that usually are provided with database products. The specific contributions we provide are:

Logical Design Constructs In Section 3.1 we propose logical design constructs to be used when designing decision support systems. Whereas a classical database design would be embodied in a schema definition, we propose augmenting the schema with Aggregated Objects, Algorithm Input/Output Definitions and an Integrity Constraint Graph.

Specific Data Model for Versions Part of the design constructs include a versioning scheme. In Section 3.2, we provide a specific data model for implementing versions. Section 3.4 outlines certain alternatives for physical implementation.

Interactive System Controls The interactive interface must insure that all operations carried out by the user leave the database in a logically consistent state. Section 3.3 provides controls that must be embedded into the interface to accomplish this.

Logical and Physical Design Alternatives As was mentioned above the logical and physical database design and the specific database operators used to carry out certain functions can significantly impact overall system efficiency. In Section 4, the results of computational experiments comparing certain alternatives are presented. Included are results indicating that a properly designed database system can provide performance competitive with file based systems.

3 Logical System Design Constructs for Version Control and Control of Algorithmic Flow

3.1 The Conceptual Level

In this section we introduce a general model for version control and control of algorithmic flow. The model employs three constructs: *aggregated objects* (AOs), *algorithm input/output definitions* and *integrity constraint graphs*. We

AGGREGATED OBJECT	CONSTITUENT TABLES
NETWORK	nodes(**node_id**, coordinate)
	arcs(**arc_id**, node_id1, node_id2, tr_time)
CUSTOMERS	customers(**cust_id**, cust_name, c_node_id, billing_add)
ORDERS	orders(**order_id**, cust_id, amount)
VEHICLES	vehicles(**veh_id**, type, capacity)
SEEDS	seed(**veh_id**, s_order_id)
CLUSTERS	clusters(**order_id**, veh_id)
ROUTES	route(**order_id**, seq_no)

Table 1: Aggregated Objects

perceive two broad motivations for the use of aggregated objects. The first reason is to provide a means for characterizing the input and output data for each algorithm and the second is to provide a convenient way of grouping like data. The set of aggregated objects should be defined so that they form a "vertical" partition of the database. That is, each data element is a member of one and only one aggregated object and all data elements of the same type are members of the same aggregated object. In a relational database, each aggregated object consists of one or more tables and in an object oriented database, each aggregated object consists of one or more objects (usually one). Since there is more than one set of algorithms one might use to solve a problem and since the manner in which one might find it convenient to group data is subjective, in general there will be more than one set of aggregated objects one could define for a given database. An example of a set of aggregated objects for the vehicle routing database based on the schema given in Figure 4 is given in Table 1. Note that in forming the schema used in Table 1 and the associated AOs, a certain amount of "disaggregation" has been performed when compared with the more natural schema given in Figure 1. Specifically, the tables orders, clusters and routes could be combined into a single table as could the tables vehicles and seeds. The reasons for doing this have to do with the manner in which the algorithms interact with the database. As mentioned before, one of the major motivations for the use of aggregated objects is to characterize algorithm inputs and outputs. The AOs shown above have been defined to correspond to inputs and outputs of algorithms e.g. the clustering algorithm accepts the AO SEEDS as input and outputs the AO CLUSTERS. Later in this paper we will discuss the possibility of combining some of these tables.

The interaction between each algorithm and the database is characterized in terms of AOs, using the algorithm input/output definition. Specifically, each algorithm has an associated set of input AOs an associated set of output AOs. There are two types of AOs: base AOs and derived AOs. Base AOs contain basic input data which must be entered into the system from an exogenous source. Each derived AO contains data which is output by a single algorithm. All de-

ALGORITHM	INPUT AO	OUTPUT AO
Seed Generation	NETWORK, CUSTOMERS, ORDERS, VEHICLES	SEED
Clustering	NETWORK, CUSTOMERS, ORDERS, VEHICLES SEED	CLUSTERS
Route Generation	NETWORK, CUSTOMERS ORDERS, CLUSTERS	ROUTE

Table 2: Algorithm Input/Output Definitions

rived AOs are initially null. An example of a set of algorithm input/output definitions for the vehicle routing database is given Table 2. Thus, in this case, the base AOs are NETWORK, CUSTOMERS, ORDERS, and VEHICLES. Before the decision processes could proceed it would be necessary that all of the constituent tables contain data. The tables in the other (derived) aggregated objects would initially be empty. Although in this example each algorithm outputs a single AO this need not always be the case. However, it must always be the case that each derived AO is the output of a unique algorithm.

Note that there is a strict precedence that must be observed in the execution of the algorithms. Specifically, the overall process must start with seed generation, then proceed to clustering, since seeds are a required input to clustering, and then end with route generation, since clusters are a required input to route generation. The fact that the algorithms must proceed in this manner is a consequence of certain fundamental properties of the data. In particular, there are certain integrity constraints that must be observed for the data to be valid. The following is one possible set for the vehicle routing example.

Integrity Constraints:

- For each cust_id that appears in the orders table there must be a corresponding cust_id in the customers table.
- For each s_order_id that appears in the seed table there must be a corresponding order_id in the orders table.
- A given s_order_id can appear at most once in the seed table.
- For each veh_id that appears in the seed table there should be a corresponding veh_id in the vehicles table.
- There should be a one to one correspondence between order_id's in the clusters table and order_id's in the orders table.
- For each veh_id that appears in the clusters table there should be a corresponding veh_id in the vehicles table.

- There should be a one to one correspondence between order_id's in the route table and order_id's in the orders table.
- No two rows in the route table that have order_id's with the same veh_id in the clusters table should have the same value of seq_no (one may wish to impose a stronger condition that the seq_no's include 1,2,3,... up to the number of orders with that veh_id).

For nearly all derived AOs, there will some integrity constraint that implies that the data in the derived AO must be consistent with the data in one or more base AOs. In particular, there must be meaningful data in certain base AOs for it to be possible to create the derived AO. We use the integrity constraint graph to characterize the general structure of the integrity constraints.

Integrity Constraint Graph Definition: *There is a one-to-one correspondence between nodes of the integrity constraint graph and AOs. A directed arc (A_1, A_2), from AO A_1 to AO A_2 is included in the integrity constraint graph if there is an integrity constraint involving A_1 and A_2. Furthermore, the integrity constraint should be such that data cannot be placed into A_2 unless there is corresponding meaningful data in A_1, i.e. if A_2 is null then A_1 must be null.*

The reverse implication will not hold, i.e. it is possible for A_2 to be null but not A_1. The integrity constraint graph for the vehicle routing database is given in Figure 5.

Note that there must be consistency between the algorithm input/output definitions and the integrity constraint graph.

Integrity Constraint Graph/Algorithm Consistency Requirement: *If A_2 is an output of some algorithm and A_1 is an input then there cannot exist a directed path in the integrity constraint graph from A_2 to A_1.*

The reason for this requirement is that the algorithm in question both generates data to be placed in A_2, and requires A_1 or a derivative of it as an input. Thus, it would be impossible for A_1 to depend on A_2. Note in Figure 5 that there is no arc from SEEDS to CLUSTERS, even though SEEDS is an input to clustering and CLUSTERS is an output. This situation occurs because the clustering algorithms requires a set of seeds as input, but once the clusters are formed they can be modified completely independently of the seeds to the extent that they could bear no resemblance to the seeds.

The complete conceptual level definition needed to initiate application development would consist of: a standard database schema together with definitions of AOs, the algorithm input/output definitions, the integrity constraints and the integrity constraint graph. In the next sections we show how these are used as the basis for controlling algorithmic flow and for versioning.

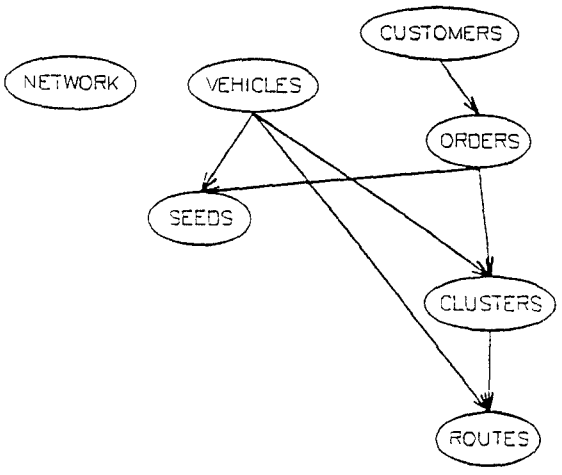


Figure 5: Integrity Constraint Graph for Vehicle Routing Schema

3.2 The Data Level

The AO serves as the basic unit for version control. Specifically, the underlying control system should support the ability to create and maintain multiple versions of each AO. For example, if the user wished to determine the result of modifying the set of clusters output by the clustering algorithm then the user might manually modify the clusters and store the result of this process as a second version of CLUSTERS while still maintaining the original version. At least for conceptual purposes, it should be assumed that each new version is represented by a completely new physical copy of all data. However, it certainly would be possible to use more sophisticated methods for maintaining versions that are more efficient both with respect to storage space and processing efficiency. This will be discussed later in the paper.

We can consider a particular version of an AO as an instance of that AO so that at any time, zero to several instances of each AO might exist. To maintain the overall integrity/organization of the data we must also maintain "instances" of the arcs in the integrity constraint graph. That is, we must keep track of the correspondence between the various AO versions. This is done using the version graph.

Version Graph Definition: *There is a one-to-one correspondence between nodes of the version graph and AO versions. Let A be an AO and (B, A) an arc in the integrity constraint graph. Then, for each version of A, $A[i]$, there must*

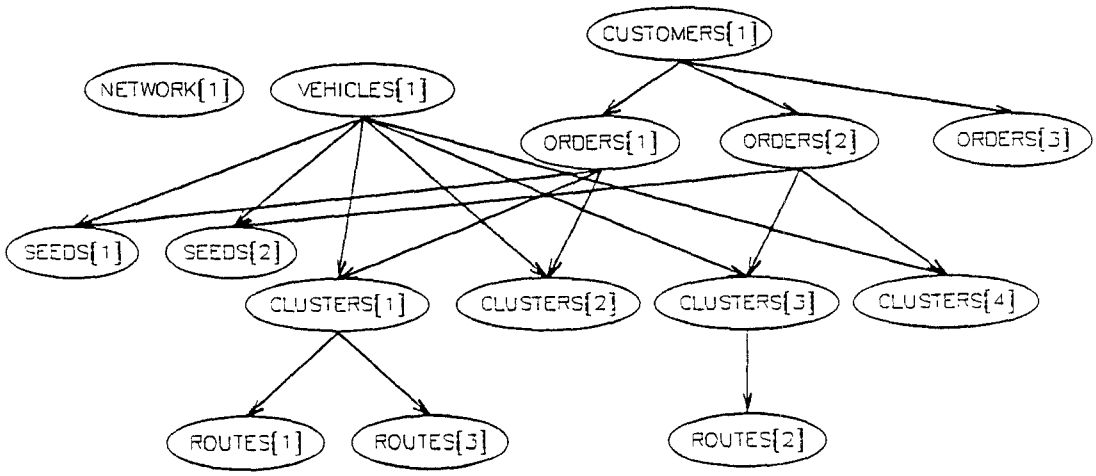


Figure 6: Sample Version Graph

exist a corresponding version of B, $B[j]$ and an arc of the form $(B[j], A[i])$ in the version graph.

For example, when a new version of CLUSTERS is created then arcs must be added which point into the CLUSTERS version node from the related VEHICLES version node and ORDERS version node. An example of a version graph is given in Figure 6.

The version graph allows us to put together complete versions of the entire database from individual versions of each AO. For example, Figure 7 has highlighted in bold a single complete database version. Thus, the version graph captures in a succint manner the information necessary to recover consistent database states.

One might also be interested in information related to how versions were derived and/or what "optimality" properties they might have. We propose to keep a record of the process that generates each derived object. Essentially such a record is an instance of the algorithm input/output definition. Every time an algorithm is executed an algorithm execution identifier is created and data is recorded indicating the AO versions that were input to the algorithm and appropriate algorithmic information, e.g. time and date of execution and parameter settings. The algorithm execution id is stored with each AO version output by the algorithm. It is common to modify a solution output by one algorithm either i.) manually using interactive system features or ii.) automatically using

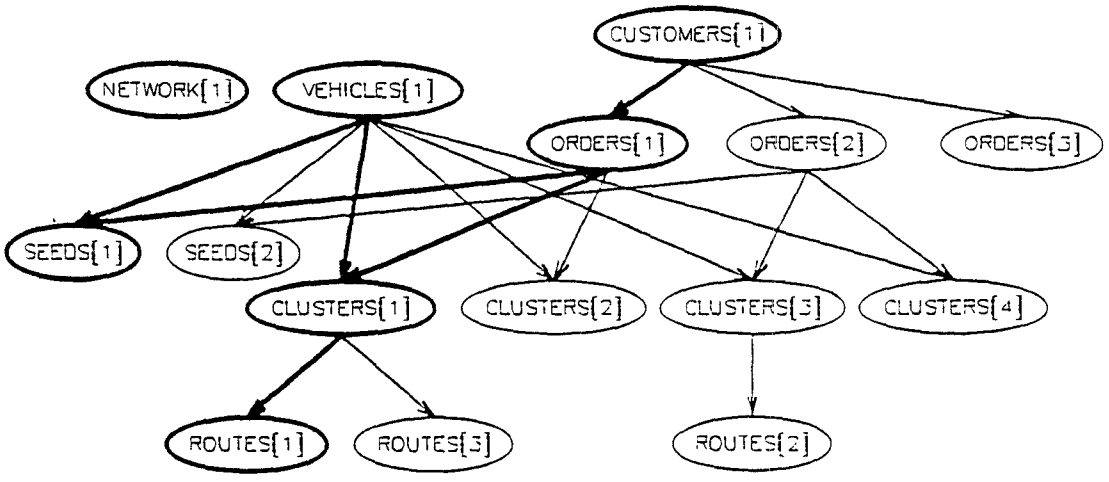


Figure 7: Version Graph with Complete Database Version Highlighted

an improvement algorithm. Since it would be very cumbersome to store the exact sequence of interactive moves the decision maker used in i.) we propose simply to store a record of the algorithmic process that originally generated the solution together with a mark indicating that the solution had been modified. Case ii.) can be treated as any other algorithimic process, while recognizing that there will be input and output AO versions of the same type. An example of this information for the vehicle routing scenario is given in Tables 3 and 4. Table 3 contains an example of the Algorithm Execution Table, which keeps a record of each execution of all algorithms. Table 4 exhibits a Version Derivation Table which indicates which process generated each version. In this example CLUSTERS[2] was generated by manual modification of CLUSTERS[1]. This fact is indicated by the * in the Version Derivation Table. Note that CL-2 and CL-3 have the same input AO set. However, the clustering algorithm's parameter settings could be different, which in general would result in different outputs (CLUSTERS[3] and CLUSTERS[4]). The last row in Table 3 represents the execution of a 'route improvement' algorithm. Improvement algorithms are procedures whose inputs and outputs are of the same type. In this case the input and output consists of a set of routes. The purpose of the algorithm is to improve the quality (objective function value) of the routes.

ALG EXEC ID	ALGORITHM	INPUT AO VERSIONS	OTHER INFO
SE-1	seed generator	ORDERS[1], VEHICLES[1], NETWORK[1], CUSTOMERS[1]	.....
SE-2	seed generator	ORDERS[2], VEHICLES[1], NETWORK[1], CUSTOMERS[1]	.....
CL-1	clustering	ORDERS[1], NETWORK[1], SEEDS[1], CUSTOMERS[1], VEHICLES[1]	.....
CL-2	clustering	ORDERS[1], NETWORK[2], SEEDS[2], CUSTOMERS[1], VEHICLES[1]	.....
CL-1	clustering	ORDERS[1], NETWORK[1], SEEDS[1], CUSTOMERS[1], VEHICLES[1]	.....
RG-1	route generator	CLUSTERS[1], NETWORK[1], CUSTOMERS[1], ORDERS[1]	.....
RG-1	route generator	CLUSTERS[3], NETWORK[1], CUSTOMERS[1], ORDERS[2]	.....
RI-1	route improve	ROUTES[1], CLUSTERS[1] NETWORK[1], CUSTOMERS[1] ORDERS[1]	.....

Table 3: Algorithm Execution Table

AO VERSION	GENERATING PROCESS
SEEDS[1]	SG-1
SEEDS[1]	SG-2
CLUSTERS[1]	CL-1
CLUSTERS[2]	CL-1*
CLUSTERS[3]	CL-2
CLUSTERS[4]	CL-3
ROUTES[1]	RG-1
ROUTES[2]	RG-2
ROUTES[3]	RI-3

Table 4: AO Version Derivation Table

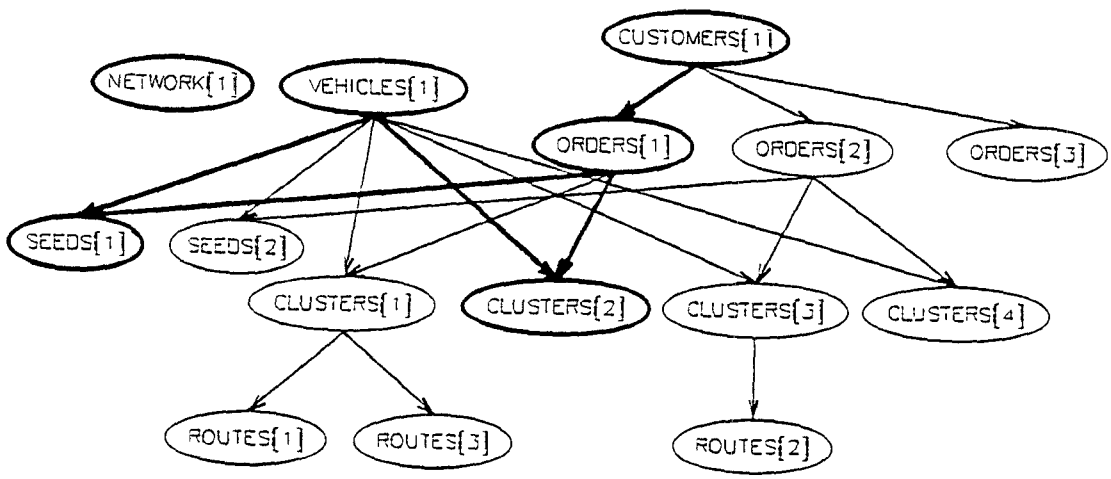


Figure 8: Version Graph with Current Version of ROUTES null

3.3 Controlling System Execution

Although the system will potentially maintain multiple versions of many different AOs, there will always be a single current version of the database presented to the user. This database version will consist of a single version of each AO, allowing the possibilities of null versions. The collections of AO versions must be consistent relative to the version graph. In Figures 8, 7 and 9 three possible current database versions are marked in bold. Note that Figures 8 and 9 certain AOs are null. Some of the fundamental interactions that the system must support include: interactively browsing through data (either in text form or graphically), executing an algorithm, interactively modifying data and changing the current version.

3.3.1 Interactively Browsing through Data

The only unusual aspect in supporting this function would be to insure that appropriate (current) AO versions were always referenced. For example, if the user was viewing CLUSTERS[2], the current CLUSTERS version, and the user wished to access related ORDERS information, then the system must insure that ORDERS[1] was referenced.

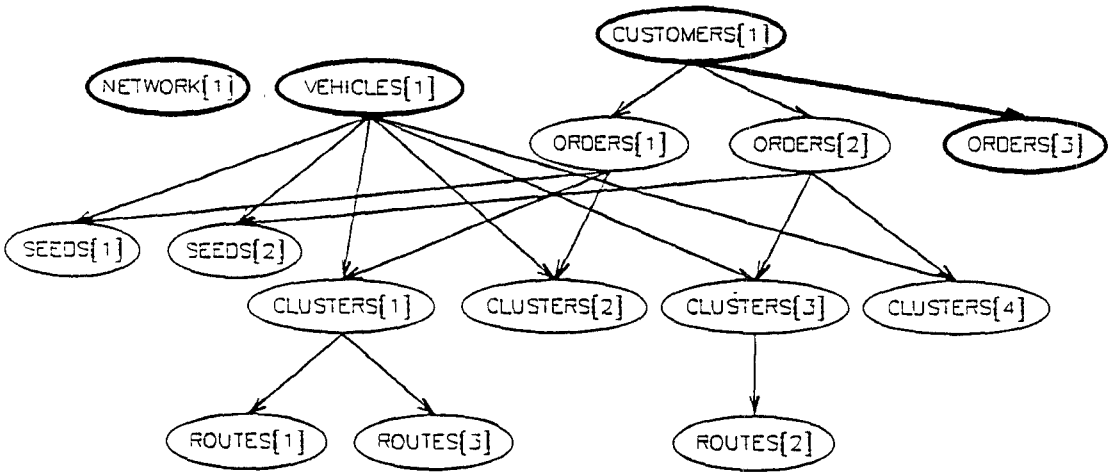


Figure 9: Version Graph with Current Versions of SEEDS, CLUSTERS and ROUTES null

3.3.2 Executing an Algorithm

One of the major system control requirements is the control of algorithmic data flow. Specifically, the system must answer the question, "When can an algorithm be executed?". Also, in the presence of multiple versions the system must determine where to place new algorithm output and how to switch among various versions.

System Control I: Suppose that the user attempts to invoke an algorithm that generates a particular AO version say $A[i]$. Suppose that B_1,...,B_n are the input AOs required by the algorithm and suppose that $B_1[k_1]$,...,$B_n[k_n]$ are the current versions of each of these AOs. Then the system implements the following controls.

1. If any of $B_1[k_1]$,...,$B_n[k_n]$ contain null (or invalid) data then the system will not allow the process to be invoked.
2. If $A[i]$ is null then the algorithm is invoked and the results are placed in $A[i]$.
3. If $A[i]$ contains non-null data then the system presents the following options to the user:

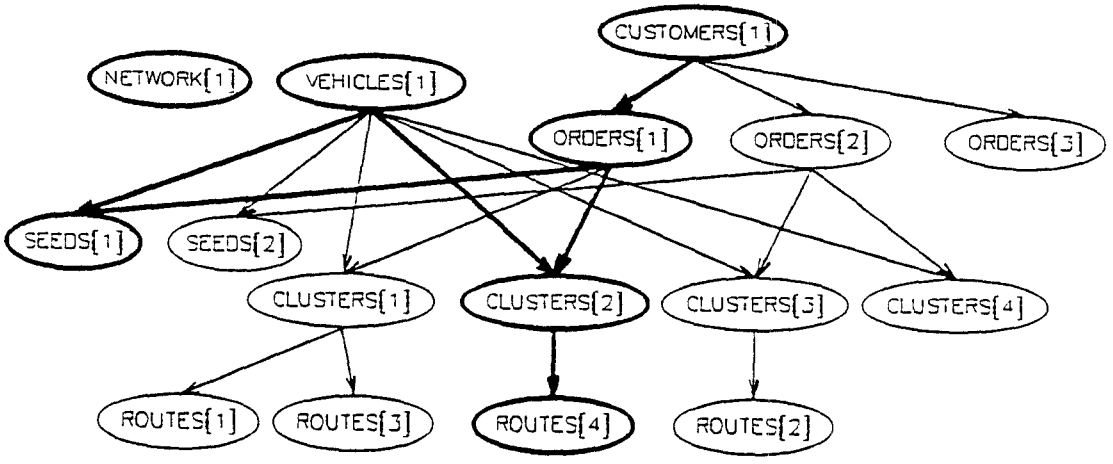


Figure 10: New Version of ROUTES created based on application of control I-2

(a) $A[i]$ and all its descendents in the version graph are made null, i.e. the data in all of these AO versions is dropped; the algorithm's results are placed in a new verion of A, say $A[i_2]$,

(b) A new version of A, say $A[i_2]$ is created; $A[i]$ and all of its current descendants are made non-current; $A[i_2]$ is made current and all of its (new) current descendants are null.

Suppose that the current database version was as illustrated in Figure 8 and that the user wished to invoke route generation. Then control ii.) would apply since the current version of ROUTES is null. The output of the algorithm would be placed in a new version of ROUTES and the new current database version would be as illustrated in Figure 10. Suppose that the current database version was as illustrated in Figure 7 and the user wished to invoke clustering. Then since the current version of CLUSTERS is non-null control I-3 would apply and the user would be offered two options. If option I-3(a) were chosen then the result would be as in Figure 11, i.e. CLUSTERS[1], ROUTES[1] and ROUTES[3] would be dropped and a new CLUSTERS version created. If option I-3(b) were chosen then the result would be as in Figure 12, i.e. a new CLUSTERS version would be created and CLUSTERS[1] and ROUTES[1] would be made non-current.

Suppose that the current database version was as illustrated in Figure 9 and the user wished to invoke route generation. Then control i.) would apply since the current version of CLUSTERS is null. Thus, the user would be prevented from executing the algorithm.

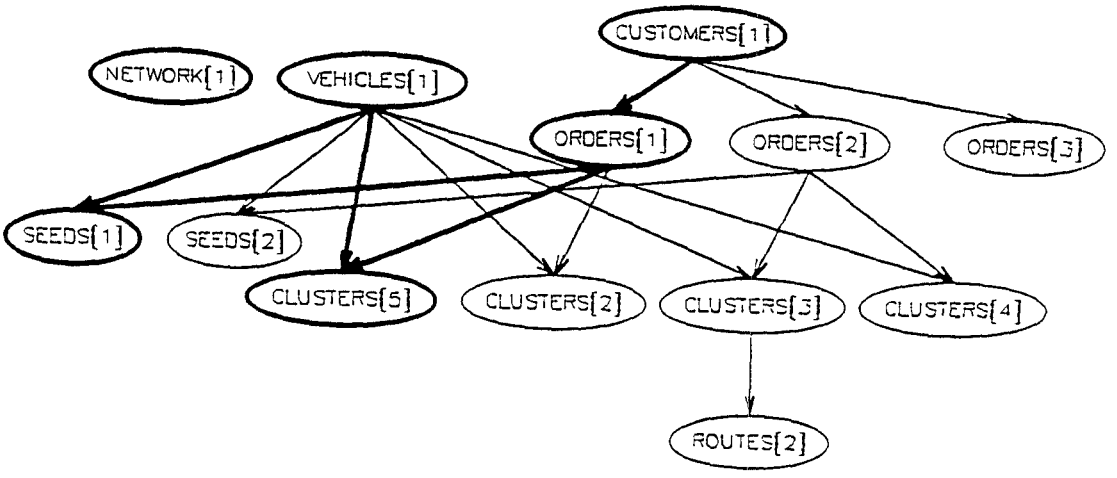


Figure 11: CLUSTERS[5] is created and CLUSTERS[1], ROUTES[1] and ROUTES[3] have been dropped based on application of control I-3(a)

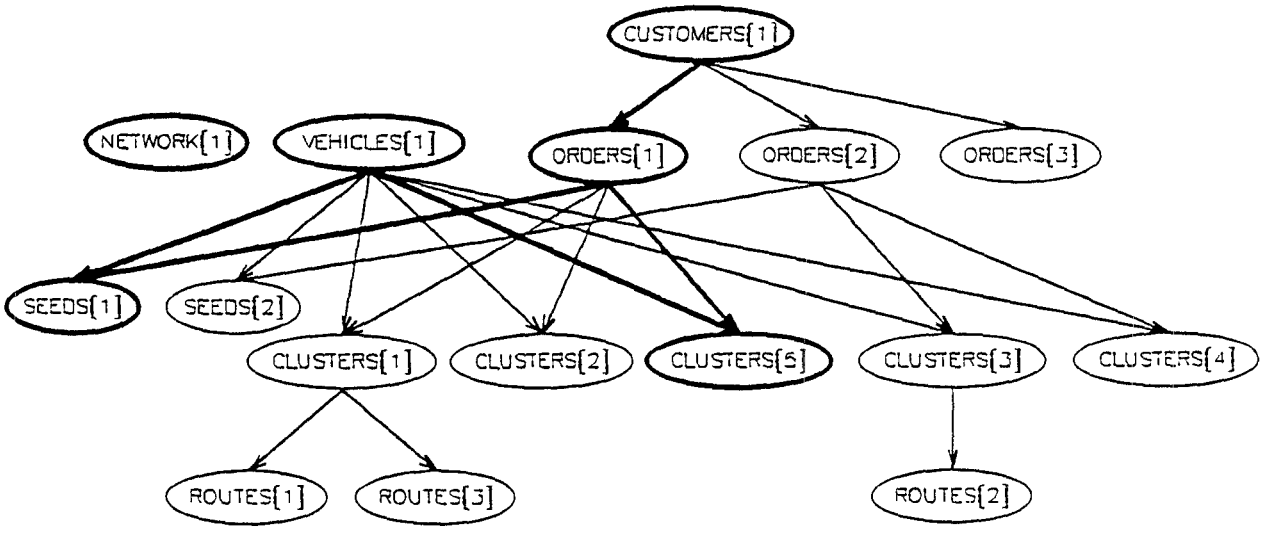


Figure 12: CLUSTERS[5] is created and CLUSTERS[1] and ROUTES[1] have been made non-current based on application of control I-3(b)

3.3.3 Interactively Modifying Data

The second important class of system control requirement relates to the use of interactive data modifications.

System Control II: Suppose that the user invokes interactive system features that results in the modification of a particular AO version say $A[i]$. Suppose that $B_1[k_1]$,...,$B_n[k_n]$ are the set of AO versions such that there exist arcs of the form ($B_j[k_j]$,$A[i]$) in the version graph.

1. Any interactive changes to $A[i]$ must be carried out in such a way that consistency with $B_1[k_1]$,...,$B_n[k_n]$ is maintained.

2. If $A[i]$ contains any non-null data then the system presents the following options to the user:

   (a) I-3(a)

   (b) I-3(b)

   (c) Let C[j] be a version of any AO that is a descendant of A[i]. Then a set of procedures can be invoked which check consistency between $A[i]$ and C[j] and alter C[j] when necessary.

The two examples related to Figures 10 and 11 both apply to this case where the user is attempting to manually modify CLUSTERS rather than trying to invoke an algorithm to generate a new version. In addition, under control II-2(c), if CLUSTERS[1] were manually modified then the user would be given the option of invoking a procedure which would change ROUTES[1] and ROUTES[3] in such a way that they remain consistent with the modified CLUSTERS[1].

3.3.4 Changing the Current Version

The interactive system will typically present to the user the current version of a single AO or a subset of all AOs. Thus, when the user requests a current version change the system must, at least implicitly, change to the appropriate, consistent new version of the entire database. For example, suppose the user was interacting with CLUSTERS[2] and the current database version was as is illustrated in Figure 7. Suppose further that the user requested to change the version of CLUSTERS to CLUSTERS[3]. Then the explicit change the user would see would be that the CLUSTERS[2] data was replaced with the CLUSTERS[3] data. However, the current version of ORDERS would have to be changed form ORDERS[1] to ORDERS[2]. This change could, in a certain sense, be implicit since the user might not immediately see any ORDERS data. The appropriate implementation of version changes is actually part of the implementation of the interactive browsing features mentioned in section 3.3.1.

3.4 Physical Storage of Versions

The physical storage of versions could form a research area by itself. We deal with the issue extensively in [8]. Below we provide a brief discussion of possible ways to implement versions in a relational scenario.

3.4.1 Direct Strategies

The basic question being addressed in this section is as follows: *how should distinct versions of AOs be stored so that they may be easily retrieved and manipulated?* First note that since each table is completely contained in a single AO, we can store a version of an AO by storing a version of each of the tables that make it up. There are two obvious direct ways to store versions of given table. The first is to create a different table each time a new version is generated. This option has two noticeable disadvantages:

- Given that algorithms are run often this will result in a proliferation of tables, affecting the efficiency of retrieving versions.
- Making a particular version the current version would involve either setting a referencing pointer to the appropriate table or moving the contents of the previously non-current table to a designated current table. Both of these approaches are very cumbersome due to the limitations of relational database systems in general and SQL in particular with respect to dynamic manipulation of tables, e.g. SQL does not support the use of variable table names.

The second approach is to maintain two tables: a current table which contains the current version and an "archival" table that contains all non-current versions. The archival table would include an additional attribute called *version_id* which distinguishes each version instance. Whenever currency has to be changed (e.g. when a new version is generated or when the user wishes to make an archived version current) the current version is moved to the current table and the contens of the current table are written to the archival relation. We have used this approach in practice as it is fairly easy to implement. However it has some clear performance limitations. If the user makes minor changes to an existing version, or if very similar instances are output by algorithms, complete new versions with a large number of identical rows would have to be created for each instance. Relatively large amounts of computing time could be spent moving identical rows in and out of tables. Thus, this approach is reasonable for relatively small tables but can have significant performance problems for large tables.

For these reasons we propose another strategy that stores versions of AOs using hierarchical viewcaches (HVCs) and manipulates them using a *lazy update strategy.* Below we briefly discuss this scheme and illustrate it using the

vehicle routing example. For a complete treatment of the procedure the reader is encouraged to consult [8].

3.4.2 Versioning Using Relational Viewcaches

In a relational database system, a database is basically a collection of base and derived relations. A derived relation may be a *view* or a *snapshot* [19]. Views do not contain any data tuples but are abstract representation of real data which is precomputed at invokation time. The purpose of defining multiple views is to provide different users or different applications with their own customized picture of the database. The *Hierarchical View Cache (HVC)* [21] is a method for maintaining views based on pointers. In HVC, the view is represented as a collection of virtual tuples. For views defined by unary operations, such as selection or projection a virtual tuple is simply a pointer to a candidate tuple in a base relation. For views defined by binary operations like join or union, a virtual tuple is a pair of pointers to two tuples in the defining view or base relations. In HVC a view is materialized whenever an instance of the view is needed. In other words, the real data tuples referenced to by the abstract view extension are fetched from the underlying base relations or views and processed to yield the fully materialized view extension. In [8], we propose to store versions using HVC.

To manipulate versions (e.g. update the HVCs when new versions are generated through algorithmic process or manual intervention) we propose to use a *lazy* or *deferred* update strategy. This tremendously improves the efficiency of our versioning mechanism. The basic idea behind our lazy update strategy is as follows: let us assume a new version of an AO is generated. To record this version pointers have to shuffled in viewcaches, such that they point to the newly created data tuples. In one extreme of this process the new version is going to contain all new data tuples (in the case where the version is created by the run of the algorithm) while in the other extreme the new version is basically the same as an old version, excepting a few modified tuples (in the case where a user modifies an old version). In the first case a whole new set of pointers would have to be referenced towards the new tuples while in the second case a few pointers would have to be referenced towards the modified tuples. If, as in conventional DBMS views, this was done as soon as the view is defined, the process would be cumbersome as constant pointer de-refencing and re-referencing would have to be maintained. In our case we propose to do the pointer manipulation when an updated view is invoked by the user and not as soon as the new or modified data tuples are created. This can be accomplished by maintaining a log of the insertion, deletion and modification to existing data tuples together with time stamps. Thus in our scheme, if a new version is created, but never invoked all the pointer manipulation overhead required in materializing that version would be saved.

To implement our storage scheme the following information needs to be

maintained:

a. A relation containing the most current version of the decision object, e.g. route(**order_id**, seq_no,time_stamp).

b. A relation containing all alternatives of that object – it should be noted that this relation is augmented when changes occur in the current relation, i.e. whenever the current relation accepts a new version, the old information is deleted from the current relation and written to this relation. In other words this relation together with the current relation contain all data necessary to generate any version of the object. An example of this relation may be: route_alt(**order_id**, seq_no,time_stamp).

c. Viewcaches containing version identifiers (vers_id), tuple pointers (tid) and time stamps. The subset of rows with the same version_ids will contain tuple pointers for all tuples in the appropriate version. The tuple ids store access paths to the appropriate data. The time stamps are necessary to determine which modifications have already been incorporated into the viewchache and which have not, e.g. route_version(**vers_id**, **tid**, time_stamp).

d. A log of all modifications to a. and b. including time stamps of such modifications – the time stamping is for the same reasons as in c., e.g. route_blog(operation_id, **tid_old**, tid_new, **time_stamp**)

In [8], we describe how these structures can be used to maintain versions in a highly efficient manner. To make best use of these ideas the database system itself should provide internal support for these operations. Ideally, the user should only be aware of the current version. All other tables (b,c,d) should be system controlled and hidden from the user. Alternatively, these ideas could be implemented on top of a commercial database system. By building appropriate application controls, the support tables (b,c,d) could be hidden from the user. However, since the application programmer would only have limited control over physical storage structures and limited access to physical storage location pointers (the tuple ids) a less efficient system would result.

3.5 Comparison to CAD Versioning

A large amount of work in the versioning area has been pursued in the field of Computer Aided Design (CAD) and we have drawn some ideas from that work. However our model diverges widely from the CAD versioning model and the purpose of this section is to briefly clarify that divergence.

To start out with, our model is simpler than the versioning schemes to support engineering design functions as we have to support fewer complexities. For example, a factor that makes our scenario much easier to handle is the fact that our system is designed to operate in a single user mode – the decision making

is perceived to be localized. Thus our system does not have to worry about concurrency control and consequently, does not have to support requirements like *check-in/check-out*, *dynamic configurations* etc. A second difference is that CAD version schemes typically maintain "is-derived-from" relationships. That is, if the user modified object version O1 to produce object version O2 then an "is-derived-from" pointer from O1 to O2 is maintained. While this would be of some interest in our applications we do not feel that the overhead warrents it. For example, if the user used the route generation algorithm to generate ROUTES[1] and then manually modified ROUTES[1] to generate ROUTES[2], we would not store a pointer from ROUTES[1] to ROUTES[2]. Rather, we would store the relationship between ROUTES[2] and the corresponding version of CLUSTERS. The Algorithm Execution Table and the AO Derivation Table would indicate the ROUTES[2] was generated by manually modifying a set of routes output by a particular run of the route generation algorithm. However, there would not be an indication that ROUTES[2] was generated directly from ROUTES[1] (there could have been intermediate versions). In addition, there would be nothing that prevented the user from deleting ROUTES[1], while still maintaing ROUTES[2].

The most fundamental difference between our approach and the CAD approach lies in the way we see the entities that need to be versioned – what we call *aggregated objects* (AOs). The objects in a CAD model are defined *hierarchically*, in many cases strict *component hierarchies* are defined. The basic reason for doing this is to facilitate top down design – specifying the detailed makeup of individual components after having described the inter-relationship among higher level components. The motivation for AOs is quite different from the motivation for CAD objects. The major purpose of CAD object definitions is encapsulation; the major purpose of AOs is to capture the algorithmic input/output process. As a matter of fact CAD applications are not constrained as we are by a strict algorithmic flow – in designing automobiles, the design of the interior is independent to a large degree of the design of camshafts while in the vehicle routing application, *routes* may not be generated without *clusters* which in turn may not be generated without *seeds*. In other words, CAD systems are built to allow the user a high degree of freedom in generating designs having a multitude of structures. Our system imposes a strict structure on the context for user problem solving.

4 Analysis of Design and Operator Alternatives

In this section we consider the computational efficiency of the types of applications we have been proposing. Two specific questions are addressed: how efficient are systems built in a database environment when compared to systems

based on a simpler file environment and how do certain logical and physical design alternatives affect the efficiency of the resultant applications. To address these questions we consider two simple generic schema alternatives. There is a single table, input_table(id, attr1, attr2,...), which contains input data used by the algorithm. The algorithm's output can be written as a set of attributes, soln_attr1, soln_attr2,..., of the input table. The alternative designs are:

Design 1: input_table(**id**, attr1, attr2,...,soln_attr1, soln_attr2,...)

Design 2: input_table(**id**, attr1, attr2,...) soln_table(**id**, soln_attr1, soln_attr2,...)

The philosophy behind these alternatives is that in Design 1, a single table holds both algorithm inputs and outputs and that single table is both read and updated. In Design 2, input_table is treated as a "read only" table and the algorithm outputs are stored in soln_table using insert operations. A Design 1 approach would lead to the vehicle routing schema given in Figure 1. A Design 2 approach would lead to a vehicle routing schema given in Figure 4. Design 1 is more natural and is generally more efficient and convenient with respect to the creation of output reports, forms and database queries. Design 2 is generally more efficient relative to the storage of algorithmic results and version manipulation.

The particular schemas we used for our experiments were:

Design 1: customers(**customer_id**, customer_name, cust_node_id, cust_address)
orders(**order_id**, cust_id, amount, veh_id, seq_no)

Design 2: customers(**customer_id**, customer_name, cust_node_id, cust_address)
orders(**order_id**, cust_id, amount)
routes(**order_id**, veh_id, seq_no)

In this case, orders corresponds to input_table and route to soln_table. The solution attributes are veh_id and seq_no. The table customers is a secondary input table. It is read in together with orders prior to the execution of any algorithm and is involved in the ad hoc query.

We compared six different options, three based on Design 1 and three based on Design 2. We constructed these options in order to evaluate the effects of different designs, to evaluate different approaches to logging and to compare a database solution with a file solution. The first Design 1 alternative, which we call *logged update*, used the following process for database retrieval and insert.

Step 1: Read all rows from customers into arrays cid[j], cia1[j], ia2[j],...; read all rows from orders into arrays oid[i], oia1[i], oia2[i],...

Step 2: Calculate values of solution values and place in sa1[i] and sa2[i].

Step 3: For i=1 to number of rows: update orders, set veh_id=sa1[i] and seq_no=sa2[i] where orders.order_id=oid[i].

The second Design 1 alternative, which we call *non-logged drop/insert*, used Steps 1 and 2 from the update alternative and replaced Step 3 with:

Step 3a: Drop orders.

Step 3b: Create orders.

Step 3b: For i=1 to number of rows: insert oid[i], oia1[i], oia2[i],...,sa1[i], sa2[i] into orders.

The third Design 1 alternative, which we call *file*, is essentially equivalent to non-logged drop/insert except that files are used in place of tables.

The principle advantage of Design 2 is that we can maintain logging and use table inserts rather than table updates. The first Design 2 option is called *logged insert*. For the database retrieval and insert we used Steps 1 and 2 from the Design 1 update alternative together with the following Step 3.

Step 3: For i=1 to number of rows, insert oid[i], sa1[i], sa2[i] into routes.

The second Design 2 alternative, which we call *non-logged insert*, used a non-logged version of the Step 3 given for logged insert. The particular set of operators necessary to carry out a non-logged insert may differ from database system to database system. For INGRES we could accomplish a non-logged insert with the following version of Step 3.

Step 3a: For i=1 to number of rows: insert id[i], sa1[i], sa2[i] temp_file.

Step 3b: Copy temp_file into routes through a non-logged bulk loading procedure.

The third Design 2 alternative, which we call *file*, is essentially equivalent to logged insert except that files are used rather than relational tables.

To determine representative ad hoc query processing times the following query was run: *Get all customer names, amounts of their orders and the vehicles that supply them.* This query was not run for the file system alternative since file environments do not support the solution of complex ad hoc queries.

Tables 5, 6 and 7 give the results of our computational experiments. The computing environment consisted of a SUN SPARC3 platform running INGRES. The algorithms were implemented in C with embedded INGRES-C function calls. Table 5 shows run times for Design 1, Table 6 shows the run times for Design 2 and Table 7 gives the query run times.

These computational results provide insight into several different logical and physical design issues.

Logged Database vs. Flat Files: It is clear that the insert/update performance of a file based approach is significantly better than performance of logged database solutions, sometimes by as much as 2 orders of magnitude.

# of tuples	Operations	File	Logged Update	Non-logged Drop/Insert
5000	read customers	1.93	2.12	3.1
	read orders	2.26	1.34	2.12
	insert/update	4.0	1154.71	7.31
	total time	8.19	1158.17	12.53
10000	read customers	3.11	4.21	4.27
	read orders	7.92	3.21	3.58
	insert/update	7.12	5909.41	12.0
	total time	18.15	5916.837	19.85
15000	read customers	5.46	6.32	6.67
	read orders	16.11	5.71	5.02
	insert/update	11.18	8723.14	18.12
	total time	33.75	8735.17	29.81
20000	read customers	6.33	9.22	7.73
	read orders	18.17	6.23	6.45
	insert/update	11.89	11948.54	20.01
	total time	36.29	11963.99	34.19
25000	read customers	8.92		8.51
	read orders	21.11		6.43
	insert/update	13.19		23.04
	total time	43.22		37.97

Table 5: Running Times for Algorithm I/O for Design 1 Alternatives (CPU time required to calculate solution not included)

# of tuples	Operations	File	Logged Insert	Non-logged Insert
5000	read customers	2.26	3.21	2.92
	read orders	2.34	1.29	1.29
	insert/update	1.93	273.26	5.21
	total time	6.53	277.76	9.42
10000	read customers	3.79	5.21	5.27
	read orders	5.69	2.46	2.12
	insert/update	1.21	602.31	6.32
	total time	10.69	609.98	13.71
15000	read customers	5.91	6.19	6.71
	read orders	6.02	5.31	4.03
	insert/update	3.41	870.21	8.07
	total time	14.0	881.71	18.81
20000	read customers	6.84	7.75	9.11
	read orders	9.74	4.21	4.34
	insert/update	3.21	1167.45	10.0
	total time	19.79	1179.41	23.45
25000	read customers	8.0	8.12	8.12
	read orders	11.44	5.54	5.78
	insert/update	4.76	1394.25	12.87
	total time	24.20	1407.91	26.77

Table 6: Running Times for Algorithm I/O for Design 2 Alternatives (CPU time required to calculate solution not included)

# of tuples	Design 1	Design 2
5000	5.32	25.12
10000	10.21	32.45
15000	17.37	46.82
20000	28.91	60.76
25000	33.04	70.58

Table 7: Query processing times

Non-logged Database vs. Flat Files: The non-logged database approaches are very competitive with the file approach for the insert/update operations. This leads one to consider very carefully the value of logging for the operations in question.

Design 1 vs. Design 2: The Design 2 performance is better than Design 1 for the non-logged and file system approaches. However, the difference is never more than a factor of 2. More importantly, the Design 2 logged insert alternative is better than the Design 1 logged update alternative by a factor of 6 to 10. It was an anticipated performance gain in this area that originally motivated consideration of Design 2. Of course, Design 2 does have poorer performance than Design 1 for query processing. It should be noted however, that the query times are substantially less than the insert/update times.

Several related issues should be considered together with the performance issues just mentioned.

The Importance of Logging: Logging allows a database to recover from system crashes that occur during update operations. A *transaction* is a collection of atomic updates that form a single logical unit. By grouping a set of updates together into a transaction the system designer is specifying that either all or none of the constituent updates should be carried out. That is, if some, but not all, of the constituent updates are executed then an inconsistent/illegal database state could result. The only transaction definition that makes sense in the context of storing the output of an algorithm is that all the updates or inserts needed to store the algorithm's results should be grouped into a single transaction. When compared to more typical database applications these are rather large transactions. The impact of logging would be that if a crash occurred in the middle of processing of the algorithm's results then the database would be rolled back to its state prior to the execution of the algorithm. Without logging it is possible that the database would be left in a state where the updates or inserts had only been carried out in part.

The Non-Logged Design 2 Solution: If a crash occurred during the insert operation of Design 2 then soln_table might not contain a complete set of rows. To rectify this situation the algorithm would have to be rerun. Such a situation can be identified and appropriate notice given to the user with relatively simple application level programming. We suggest the creation of a status relation of the following form: status_table(**soln_table_id**, flag). This relation would contain one row for each solution table. Flag would take on values 'good', indicating that the contents of the solution table was valid, and 'bad', indicating that the contents of the solution table was possibly invalid. Just before commencing the storage of results into soln_table the corresponding flag attribute would be set of 'bad'. After completion of the inserts into soln_table it would be set to 'good'. If a crash occurred during the insertion process then the flag value would be 'bad'; otherwise, it would be 'good'. Subsequently, when an access to soln_table

was attempted, a 'bad' flag value would be interpreted as equivalent to the table containing null data.

The Non-Logged Design 1 Solution: The effect of using non-logged operators for Design 1 is more severe than for Design 2. The reason is that in the Design 1 solution input_table is dropped temporarily. Thus, a crash during the execution of the algorithm/insert process could potentially result in the loss of the original input data. Application level recovery procedures could be implemented using the ideas presented above but they would be more complex. Specifically, backup versions of the input attributes would have to be maintained. An alternative approach would be to store the results in a second table, e.g. input_table2 and to delay dropping input_table until after the completion of the process. Afterwards, input_table2 itself would be used or it would be copied into a newly created ipnut_table. A further, related exposure is that errors in the procedures could result in the corruption of the input attributes which might be hard to detect. While all of the problems mentioned can be solved, they put a significant burden on the application programmer for recovery and integrity control. For these reasons we generally do not recommend the non-logged Design 1 solution.

Efficiency of Versioning: Most of the analysis given in Section 3 is based on a Design 2 approach. Consequently, if one wishes to make use of all of the constructs from Section 3 and to achieve the best versioning efficiency then Design 2 should be used. However, we should note that it is possible to use hybrid approaches and/or to use a Design 2 schema to organize system design, versioning and algorithmic flow while using a Design 1 schema for the actual implementation. As an example of a hybrid approach, in the schema given in Table 4 tables Clusters and Routes could be combined into a single table, rte_clust(**order_id**, veh_id, seq_no). Then, for versioning, one might only store versions of table of rte_clust or, alternatively, use the original clusters and routes tables to store versions. This would in turn require a more complicated process for storing and retrieving versions.

Our recommended strategy is to start with a logged Design 2 approach. Specifically, a Design 2 schema should be constructed and a complete application designed based upon it, as is outlined in Section 3. Once this is done modifications to the design should be considered to achieve certain objectives. For examples, to achieve performance improvements in the storing algorithmic results one would make use of non-logged operators. To achieve better application aesthetics and/or improve query processing times one would consider a Design 1 approach to the entire schema or portions of it. Whenever the Design 2 approach was used one would have to redesign the method for implementing versioning. Alternatively, one might not require versioning for the entire schema or portions of it.

We feel that the preceding analysis quantifies very definitively several important tradeoffs and can be used by application developers as a guide during the design process. A very important second result is that by using non-logged

operators one suffers very little in terms of recoverability while at the same time achieving performance competitive with file systems.

5 Conclusion

In this paper, we discussed approaches to two of the most significant problems that arise when embedding OR models within database environments. We now conclude with a list of additional problem areas that we view a starting points for additional research in this area.

5.1 The Sequence Number Problem

The solution to many types of problems involve sequences in one form or the other. For example, solution to a vehicle routing problem includes the order in which a vehicle visits nodes (warehouses, depots) in a network, solution to a machine scheduling problem involves an ordering of jobs on machines and so on. This problem can be broadly stated through the following questions:

- How does one store a sequence?
- Having stored a sequence, how does one manipulate related information?

Three obvious alternatives for storing a sequence are:

1. by using consecutive sequence numbers,
2. by using non-consecutive sequence numbers,
3. by using predecessors.

Performance and logical design tradeoffs exists when deciding among these alternatives.

5.2 The Partitioning Problem

As we mentioned at the outset, it is often desirable to solve pieces of a problem, as opposed to solving the entire problem at one time. This may be because the user wishes to focus on a specific subproblem, or because of algorithmic reasons as outlined before. Whatever the reason, a number of data management functions are required in a system that supports partitioning. A partial list of required functions are:

1. Managing the integrity of the overall process: the system must insure that the decision variables associated with each partition do not overlap and that each decision variable from the global problem is contained in some subproblem.

2. Providing flexible partition manipulation functions: the user will very often wish to manipulate partitions in a variety of ways. For example, after an initial solution is obtained the user might wish to combine three partitions and from these generate two new ones. Alternatively, the user might wish to take a small portion of a solution from one partition and merge it into the solution from an adjacent partition.

3. The ability to create a global solution from the solutions on each partition: in the end the solutions over each partition will typically be merged into a single global solution.

5.3 The Numbering Problem

To illustrate the problem under consideration, let us assume that our relational database contains information about customers. In nearly all practical situations, these customers would be identified by character names. The algorithms would typically treat customer information in some kind of sequential numerical form to be read into arrays, vectors etc. Thus, a mapping between character identifiers stored in tables and numerical array indices is required. This mapping is crucial at two distinct stages:

a. A transformation must be achieved when reading the data in,

b. a consistent retransformation must be affected when inserting new information back into the database.

What this requires is maintaining some kind of mapping representation while the algorithm is running. The questions that arise therefore are:

1. Should the mapping be represented as a temporary relation in the database, which involves frequent updates because the mapping could potentially change with every run of the algorithm or,

2. Should the mapping be maintained in some kind of RAM representation while the algorithm is running which involves programming jugglery and some lost main memory?

We note that these issues are complicated by the fact that it is rare that problems are solved over all entities in a table. For example, a vehicle routing problem is typically solved over a sometimes small subset of the entries in the customer table. Thus, any solution must take into account the requirement that the set of customers will be redefined for each algorithm run.

6 Acknowledgements

The work of the first two authors was supported by National Science Foundation grant number NSFD CDR-8803012.

References

[1] S. Abiteboul, M. Scholl, G. Gardarin and E. Simon, *Towards DBMSs for supporting new applications*, Procs. of the 12th Intl. Conf. on VLDB (Kyoto, August 1986), pp. 423-435.

[2] A. Aho and J. Ullman, *Optimal partial-match retrieval when fields are independently specified*, ACM Transactions on Database Systems, 4, 168-179 (1979)

[3] M. Ball and H. Benoit, *A Langranian Relaxation Based Heuristic for the Urban Transit Crew Scheduling Problem*, Computer Aided Scheduling of Public Transport: Proceeding of the Fourth International Workshop on Computed-Aided Scheduling of Public Transport, J. Daduna and A. Wren, eds., Springer-Verlag Lecture Notes in Economic and Mathematical Systems #308, Heidelberg, (1988) pp. 54-67.

[4] D. Batory, *Genesis: A project to develop an extensible database management system*, Procs. 1986 Intl. Workshop on Object oriented Database systems, Sept. 1986

[5] D. Batory, *Physical storage and implementation issues*, A Qtly. Bulletin of the IEEE Computer Society Technical Committee on Database Engineering, 7, 49-52 (1984)

[6] A. Bjornerstedt and C. Hulten, *Version control in an object oriented architecture*, in Object Oriented Concepts, Databases and Applications, ACM press, 1991, pp 451-485

[7] R. Dahl, J. Greenberg, J. Sanborn, C. Skiskim, M. Ball and L. Bodin, *A Relational database approach to vehicle and crew scheduling in urban mass transit systems*, in Computer Scheduling of Public Transport, North Holland, 1985, pp 327-342

[8] A. Datta, *Using Relational Viewcaches to Implement Versions in Decision Support Systems that use Optimization Models*, College of Business and Management, University of Maryland at College Park, Working Paper #MS/S-92-015, 1992

[9] M. Fisher, *Interactive Optimization*, Annals of Operations Research, 4, 541-556 (1985).

[10] M. Fisher and R. Jaikumar, *A Generalized Assignment Heuristic for Vehicle Routing*, Networks, 11, 109-124 (1981).

[11] B. Gavish and K. Altinkemer, *Backbone Network Design Tools with Economic Tradeoffs*, ORSA Journal on Computing, 2, 236-252 (1990)

[12] A. Geoffrion, *An introduction to structured modelling*, Management Science, 33, 547-588 (1987)

[13] S. Ghosh, Data Base Organization for Data management, Academic Press, NY, 1977

[14] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms et al, *Starburst Mid-Flight: As the dust clears*, IBM Research Report: RJ7278, IBM Almaden Research Center

[15] W. Kim and D. Batory, *A model and storage technique for versions of VLSI CAD objects*, Foundations of Data Organization, Plenum Publishing Corporation, NY, 1985, pp 427-439

[16] S. March, *Techniques for structuring database records*, ACM Computing Surveys, 15, 45-80 (1982)

[17] C. Monma and D. Sheng, *Backbone Network Design and Performance Analysis: a Methodolgy for Packet Switching Networks*, IEEE Journal on Selected Areas in Communications, SAC-4, 946-965 (1986).

[18] W. Plouffe, W. Kim, R. Lorie and D. McNabb, *Versions in an Engineering Database system*, IBM Research Report: RJ4085, IBM Almaden Research Center

[19] M.E. Adiba and B.G. Lindsay, *Database Snapshots*, Proc. of the 6th Int'l Conf. on VLDB, 1980 pp. 86 - 91.

[20] R.H. Katz, *Toward a Unified Framework for Version Modeling in Engineering Databases*, ACM Computing Surveys, December 1990 pp. 375 - 407.

[21] N. Roussopoulos, *View Indexing in Relational Databases*, ACM TODS 7, (1982)

[22] M. Stonebraker, *Implementation of Integrity constraints and Views by Query Modification*, Proc. ACM-SIGMOD, June 1975, pp 65 - 78.

[23] K. Terplan, Communication Networks management (Prentice Hall, New Jersey, 1992).

[24] P. Valduriez, *Join Indices*, ACM TODS 12, (1987)