ECES 433

Final Design Project

“Bringing an RDBMS to TACACS+”

by Andrew Reitz (reitz@ces.cwru.edu)

December 11th, 1998

Introduction

In the realm of data networking, it currently very useful to provide remote access to a network, for those that must be away from it. This means that it should be possible to access a private network (such as a campus *Intranet*) via some sort of public network (either the telephone system, or as is the case more recently, the *Internet*). Because this is such a useful service, many network companies have created many different access mechanisms over the years. Starting with the primitive *terminal server*, and moving on to the current *PPP access server*, each held the common remote-access goal in mind. The *Internet Engineering Task Force* (IETF), with the aid of companies like *Cisco Systems,* attempted to bring some level of commonality to all of these devices by several means. The devices by which remote users actually connected to the local network were termed the *Network Access Server* (NAS). Typically, a NAS must have some means by which it can *Authenticate*, *Authorize*, and *Account* for its remote users. Before standardization, these roles were handled in unique and proprietary manners. The standardization process specified the exact roles of the NAS and the back-end NAS server, in order to simplify their respective implementations, as well as provide a foundation for interoperability. Thus, a set of protocols were specified, in order to govern the communication between the NAS and the back-end NAS server, which is in essence the central governing body for all of the NAS's in the network. By creating a common protocol, NAS's from many vendors would be able to interoperate with a single NAS server, thus simplifying the *Network Administrator's* role greatly.

For the purposes of this assignment, I am going to focus on the so-called *Terminal Access Controller Access Control System Plus* (TACACS+) NAS server, as designed by Cisco Systems. TACACS+ was designed with complete separation and extensibility of the three basic elements (*authentication*, *authorization*, and *accounting*) in mind. The process of verifying the identity of a user (or entity) is accomplished through *authentication* in TACACS+. Many different username/password authentication methods are supported, via the extensible nature of TACACS+. An *authorization* request in TACACS+ attempts to refine the level of access that a remote user has with the local network. This can be anything from restricting configuration parameters (service type, IP address, how routing information is passed) to actually establishing a per-session *Access Control List* (ACL), limiting which areas of the network to which the user can gain access. Finally, *accounting* is the process by which all of these remote user sessions are logged. The *accounting* mechanism is generalized enough that it can record what a user has or is currently doing on the network, and thus can be used for billing services as well as to perform security auditing.

As implemented, the TACACS+ server stores all of its information in a series of plain ASCII text files. This is fine for most useses of this service, but as Ameritech's use of TACACS+ grows, so do their requirements for it. The requirements range from increased reliability and scalability issues, to those of more flexible reporting/querying of the data. Thus, it is important to describe the benefits (both actual and desired) which Ameritech seeks to gain by implementing TACACS+ with an RDBMS back-end. Firstly, it is hoped that an RDBMS will provide a unified engine for all of the data necessary in order to support a customer. Furthermore, the RDBMS should easily allow concurrent access to the data, from multiple TACACS+ servers. An RDBMS should also allow strict partitioning of data – so that all of the records for each customer could be kept separate (and secure) in a logical manner. Finally, ease of querying (both via SQL and transactions) is a definite plus – some of the more advanced data manipulations are becoming too difficult to do with just flat text files.

Initial description of Data Model

In essence, as previously discussed, there are three basic relations in this model: *Authentication*, *Authorization*, and *Accounting*. Furthermore, each relation (at least initially) will form the basis of a corresponding *table* in the dtabase. Each relation will have it's own unique features, and it's quite possible that one or more of them might act independent of the others. The following is a description of the data model, taken one relation at a time.

Authentication:

All of the information necessary in order to authenticate a user and establish their initial network connection is contained in a *"configuration file"*-style format. The exact nature of the syntax used in this file is actually rather ad-hoc, and probably not in BNF. In essence, the *authentication* consists of a series of attribute-value pairs for each user record, that define the paramaters for his/her acceptance into the NAS. Thus, migrating this relation to an RDBMS contains it's own special set of challenges. Since there can be a variable amount of data, and I cannot use an Object-Relational approach (which, at least in theory, would make it easy to manage a variable amount of data per row), I plan to make liberal use of NULL values. Basically, there are two methods to attack this problem. The first is to attempt to implement a subset of all possible parameters that TACACS+ allows in the *authentication* configuration file, in order to support the most common requirements of most sites. A subset must be supported in order to keep things from getting exponentially complex. The second method of modeling this data involves including only the attributes that are actually used by Ameritech. As it turns out, Ameritech uses a very limited subset of all of the *authentication* options available, in a very uniform mannter. Thus, the second method will actually be implemented during the course of this assignment. The entities in this method include those for a *userid* and *fullname*, as well as *password* and *group-affiliation*.

Authorization:

The *authorization* information is used to constrain the range of connectivity for a session, based upon the *userid*. In the current TACACS+ server, this information is actually presented in the same file (and format) as the *authentication* information. However, I plan to break from this current methodology, and analyze this relation separately from the *Authentication* relation. Like the *authentication* information, the *authorization* information consists of a list of attribute-values (AV) pairs. Each AV pair specifies an option (attribute) and its desired effect (value). However, the *authorization* relation differs in one key area: not all values are mandatory. The attribute and value may be separated either by an equals sign ('=') or an asterisks ('*'). The former indicates that the attribute is mandatory, the latter indicates that the attribute is optional (and thus can be disregarded at the receiver's whim). Implementing this in a relational database would pose a very special challenge: typically, a hard link exists between any given value in a field. The notion of *optional values* doesn't seem to exist in the realm of relational databases. The only method of which I can think that might support this would be some sort of encoding scheme, whereby fields were marked optional or not. This could consist of an extra *"optional-status"* field for every other field in the relation, or as an additonal integer field that acts as an *"optional-mask"*. Fortunately, for the purposes of Ameritech's use of TACACS+, only the mandatory attribute value pairs are ever used. This, this implementation will only consider the mandatory attribute-value pairs.

The exact types of attributes that are contained in an *authorization* record are variable and extensible, based upon the requirements of the NAS and the customer. In general, the entities included in this relation will cover areas such as *access-control*, *timeouts*, and *network addresses*.

Accounting:

The *accounting* table is a repository for all of the log-style type data that a TACACS+ server generates during the normal course of operation. The basic types of data logged falls into three separate categories (or types):

- **START** – Indicates that a particular service is about to begin.
- **STOP** – Indicates that a given service has stopped.
- **UPDATE** – Indicates that a given service is still in progress, and that there is some new data to be considered in this session.

Although these records don't inherently have a *state*, as the TACACS+ server logs them, it is possible (and, as it turns out, very useful) to draw states from the raw information. Using the *START* and *STOP* records, combined with the *task_id* and *NAS-hostname*, it is possible to account for a *call*. Basically, a *call* is one user session, from the initial connection, to when they finally end the connection by hanging-up. Since this state information is so useful (just about all of the accounting queries revolve around this), it makes sense to split the accounting information into two tables: one filled with the raw data received from TACACS+, the other consisting of the constructed *call* records. This would be implemented via some sort of transaction, that culls the *call* records from the raw data, and stores said records in a new table.

Unfortunately, because life is unpredictable, it isn't always true that there will be a corresponding *START* record for every *START* record. The following list explains all of the possibilites that exist:

Possible causes for a dangling STOP record:

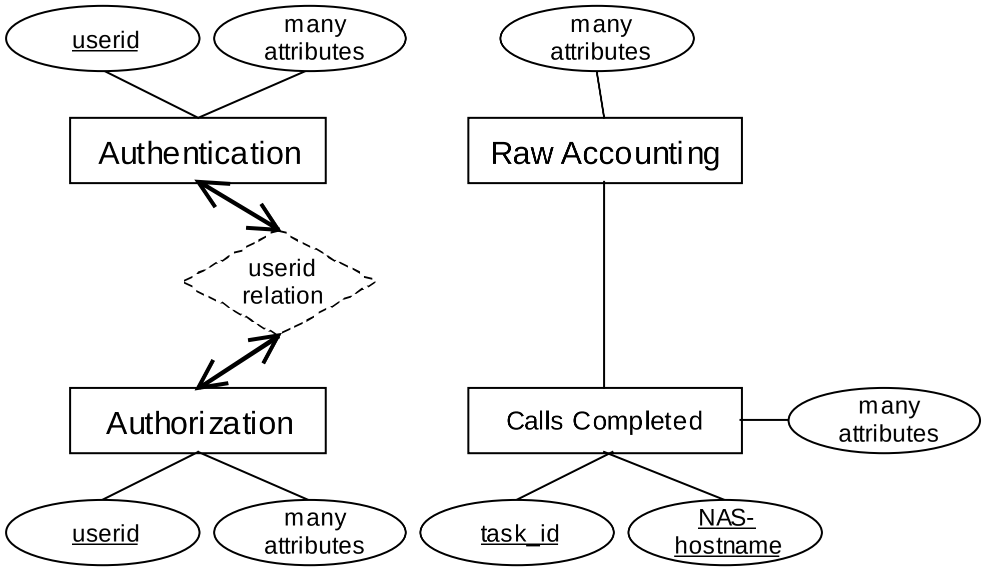
1. The login attempt failed before authentication (or authorization, as the case may be) succeeded. This could be identified by the fact that all of the regular byte counters will be zero.
2. The user attempted to login using a *userid* that wasn't found in the system. The TACACS+ server will log this type of request as a STOP record.
3. The associated START record was in fact generated by the NAS, but was lost in transit to the TACACS+ server. There is no way to reliably detect such an occurrence.

Possible causes for a dangling START record:

1. The call is still active.
2. The call has stopped, but the associated STOP record was lost.

Thus, for these reasons, procuring a *call* state from the raw data isn't always possible. Consequently, any records that were left in the original raw table could be analyzed separately.

ER Diagram:



In summary, this diagram gives a *"feel"* to initial design, and in fact presents some information that hasn't yet been discussed (such as the potential keys for the relations). The *Authentication* and *Authorization* relations are related through the *userid* attribute, in a one-to-one relationship. Basically, we require that for every user record that exists in the *Authentication* relation, that a corresponding record exist in the *Authorization* relation (although, all of the *authorization-specific* values may be NULL, if the customer wishes). It's important to note that the *"userid relation"* doesn't exist anywhere else in the database other than in the ER diagram. It is simply a visual method for displaying the link between the *Authentication* and *Authorization* relations.

The *accounting* relations aren't actually linked by any attributes in the database. Instead, their link is a conceptual one. Because the *Calls_Completed* table is generated from the *Raw_Accounting* table, it makes sense to think of these two tables as *related*, when in actuality, they don't have a hard link.

The following is a listing of possible queries, broken down by service-type, and ordered by *frequency of arrival*.

Authentication:

1. Emulate an *Authentication* request – search for a user, and return his/her values.
2. Determine the total number of users, as well as the number of *active* users (those who have a password other than the default).
3. Display a list of users that have at least one password set to *"tempass1"*. This is the default password, set when an account is first created. Thus, this query can be used to show which users haven't used their accounts as of yet, which could lead to a potential security issue.
4. Find all users that have both the special *"admin"* type attribute, as well as an uninitialized password. This is a very big security hole.

Authorization:

5. Emulate an *Authorization* request – search for a user, and return his/her attributes.
6. Display users of type *admin* that have very permissive security settings.
7. Display a list of *"secure" userid*'s – those that have values for both the *inacl*, and *outacl* attributes. Furthermore, their *routing* attribute should be set to *false*.
8. Display a list of all users who have very *"limited"* access – i.e. their *timeout* value is less than or equal to 60 minutes.

Accounting:

9. Display a listing of users that are logged in on a given NAS for a given time period.

10. Display a *"call history"* (complete list of calls) for a given user.
11. Find the last login for every user, sorted in reverse order.

Design of Relational Database

In the section that follows, I will attempt to actually *design* the database. This will involve designing the schema for each relation, showing the appropriate SQL for each schema, and also re-visiting the potential queries above. Again, this task will be partitioned, and each relation will be analyzed separately.

Authentication:

As previously discussed, there are two different methods for handling the complex TACACS+ *authentication* structure. One method is to attempt to emulate all of the options supported as well as can be supported in a relational database. The conceptual schema (and associated discussion) for this approach follow:

Full_Authentication (*userid:* **string**,
login: **string**,
secondary_login_type: **string**,
service1: **string**,
protocol1: **string**
protocol1_param: **string**
service2: **string**,
protocol2: **string**
protocol2_param: **string**
service3: **string**
protocol3: **string**
protocol3_param: **string**)

The value of the *userid* attribute represents a unique user in this particular *authentication* table. This *userid* designates the start of a record in the config-file format, and each user may have a variable number of configuration-type parameters associated with him/her. The *login* attribute specifies that the user has the capability to login to the remote network. The *secondary_login_type* attribute is optional, and can specify a different sort of login protocol to be used. Possible values include *pap*, *chap*, and *ms-*

chap. If a *secondary_login_type* isn't specified for a user, the NAS may then revert to the value contained in the regular *login* attribute in order to authenticate the user.

The final set of attributes represents an attempt to deal with the arbitrary and variable nature of the TACACS+ *authentication* configuration structure. Basically, TACACS+ allows for an arbitrary number of *service* attributes, and each service attribute may have further *protocol* attributes embedded within it. This is where *object-relational* databases would really be useful, but since I don't have that, the appropriate behavior can be emulated (to a certain extent) by including multiple corresponding *service* and *protocol* attributes.

Another possible way to design the *Authentication* schema is to implement only the attributes that Ameritech requires for the daily use of their TACACS+ servers. While this approach is less flexible, it will readily support the TACACS+ servers that are in the field today, and speed the implementation of the RDBMS. Thus, I will implement this project based around the following conceptual schema:

Authentication (*userid:* **string**
name: **string**
login: **string**
member: **string**
chap: **string**
pap: **string**
type: **string**
global: **string**)

The *userid* and *login* attributes are the same as above. The *pap* and *chap* attributes are derived from the *secondary_login_type* from above, and hence contain similar functionality. The *name* attribute is simply a text string that lists the user's full name. The *member* attribute associates the particular user with a group. A user inherits all AV pairs from all groups of which it is a member. If the AV pairs conflict, then their own local values will take precedence over group values, and group values will take precedence in the order that they are listed. Today, Ameritech only uses one group, and all users are placed in said group. The format of a group looks something like this:

```
group = template {
    Service = ppp protocol = lcp { }
    Service = ppp protocol = ip { }
}
```

In this case, the group contains the *service* parameters necessary in order to define the PPP service that is offered to the remote client. For the time being, since there is only one group, there isn't a need to store it in a separate relation. In the future, it might be more flexible to store the group information in a separate relation, so that those queries could gain a more accurate understanding of a particular user's profile.

The last two attributes that deserve discussion are *type* and *global*. The *type* attribute is optional, and is used to specify whether-or-not the user is of a special type, which entitles them to special privileges. Currently, only the special *"admin" type* is supported. This type allows the user to use the *administration client,* in order to add/modify/delete users on the system. Finally, the value of the *global* attribute is a password that can be applied to any authentication method. Apparently, the implementers of TACACS+ at Ameritech learned about the *global* attribute after a large investment had been made in storing passwords in their individual authentication formats, so both forms were kept. Thus, for completeness, this project has been written to support both the legacy and newer *global* password storage mechanisms.

SQL Implementation of the Authentication schema:

```
CREATE TABLE Authentication( userid VARCHAR2(11),
                             name VARCHAR2(30),
                             login VARCHAR2(18),
                             member VARCHAR2(10),
                             chap VARCHAR2(18),
                             pap VARCHAR2(18),
                             type VARCHAR2(5),
global VARCHAR2(18),
PRIMARY KEY (userid));
```

Authorization:

For the purposes of this assignment, the conceptual *Authorization* schema will be as follows:

Authorization (*userid:* **string**,
inacl: **integer**,
outacl: **integer**,
timeout: **integer**,
idletime: **integer**,
addr: **string**,
routing: **boolean**,
route: **string**)

The *userid* attribute corresponds to an *userid* value in the *authentication* table. Basically, this means that it is a *foreign key* in the *Authorizatoin* relation. Thus, there can only be one *authorization* record per *userid*, since the *userid* must be unique in the *authentication* table. The *inacl* and *outacl* refer to inbound and outbound access lists, identified by a unique integer. Since these access lists are protocol dependent, and thus can be very complicated (For example, Cisco allows them to operate on layer four of the OSI TCP/IP model), they tend to be stored in the NAS. This both eases NAS implementation and increases performance, but impinges upon administrator maintainability of the system. The *timeout* and *idletime* attributes take integer values that represent a span of time, in seconds. The *timeout* parameter represents an absolute time for the session. For example, the CWRUnet free PPP dial-in pool uses a *timeout* of one hour, at which point the user will be automatically disconnected. The *idletime* attribute specifies how long a user may stay connected without displaying any activity (basically, data transferred). When this amount of time has been exceeded, the user will be disconnected by the NAS.

The final three attribute-value pairs deal with the network-layer configuration. The exact format and meaning of these parameters is determined by the protocol specified in the *authentication* database. The *addr* attribute specifies a network address, to be used by the remote host when connecting via SLIP or PPP/IP. The boolean *routing* attribute specifies whether-or-not the NAS is supposed to both send or receive routing information from the client. The *route* parameter specifies any network routes that should be configured automatically when the user-session is first established.

SQL Implementation of the Authorization schema:

```
CREATE TABLE Authorization ( userid VARCHAR2(11),
                             inacl INTEGER,
                             outacl INTEGER,
                             timeout INTEGER,
                             idletime INTEGER,
                             addr VARCHAR2(15),
                             routing VARCHAR2(5),
                             route VARCHAR2(49),
                             FOREIGN KEY (userid) REFERENCES Authentication);
```

Accounting:

We will begin by dissecting the *raw_accounting* table. For the purposes of this project, we will only deal with the *START* and *STOP* records, primarily because Ameritech uses very few *UPDATE* records. In this scenario, the *START* and *STOP* records share a common base, a set of fields that start off the record. However, the *STOP* records add many additional statistical fields to their record. Consequently, in order to ease the analysis of these records, a conceptual schema for the common base will be presented first, and then the additional schema for the *STOP* record will be analyzed later. The following is the conceptual schema for the common *accounting* base:

Raw_Accounting: (*date:* **date**,
NAS-hostname: **string**,
userid: **string**,
NAS-port: **string**,
remote_phone_number: **integer**,
type: **string**,

NasUserName: **string**,
task_id: **integer**,
timezone: **string**,
service: **string**)

The *date* attribute records the current system time when the transaction was received, stored in a modified UNIX *ctime* format (sans timezone). Unfortunately, the *date* cannot be considered as a key for this relation, because it is possible for the TACACS+ server to process multiple requests at the same instant of time (the granularity is only seconds). Furthermore, even if the granularity of the *date* attribute is increased, it still cannot be a key – consider the possibility that multiple TACACS+ servers are using the same RDBMS database for storing their accounting information. In this scenario, multiple transactions could actually occur at the *exact same* time.

The *NAS-hostname* attribute is an identifier that specifies that NAS device that generated the message. Although it does not have to be unique (it's possible for two different NAS devices to have the same hostname), it typically is, because it's much easier on the administrator. The *userid* attribute is a foreign key into the *authentication* relation, which specifies the user that has initiated the current transaction. The *NAS-port* attribute is a string that specifies which port (typically, a NAS will have to ability to handle many simultaneous connections via different physical ports) the current session is using.

The *remote_phone_number* field is generated by the NAS, using CallerID. Basically, the phone number that the client uses in order to connect to the NAS is logged. This information can be used during the authentication process, in order to further establish the user's identity, and also for security auditing purposes, which is why it is relevant to the *accounting* table. It should also be noted that in some instances (but not all), the NAS will append a '/', followed by the number within the NAS that was dialed (sans area code) in order to establish the connection. This functionality could be used in order to share a PRI[1] line between multiple customers in the future, but is currently unused by Ameritech today, so I will ignore it for the course of this assignment.

The *type* field contains one of three text strings: *"START"*, *"STOP"*, or *"UPDATE"*. The *NasUserName* contains the same data as the *name* field from the *authentication* table. The redundency that this attribute causes will be further discussed in the section dealing with the *integrity constraints* of this design.

The *task_id* attribute is a unique value (per NAS) assigned to every *session* that the NAS starts. Thus, it is a very important value, because it can be used to ensure that any given accounting records "go together" when considering the session as a whole. However, this value alone cannot be considered a key for the relation, because multiple *accounting* log entries will make reference to the same *task_id*. For example, there will be a START, STOP, and an arbitrary number of UPDATE records for every *task_id* in the system. However, I believe it might be feasible to combine the *date*, *task_id*, and *NAS-hostname* attributes in order to form a key for this relation. This key assumes that a NAS will not process multiple requests for the same *task_id* in a given instant of time. Unfortunately, during the implementation, this assertion didn't hold, so I decided to not maintain a proper key for this relation.

Finally, the *timezone* attribute specifies the current timezone (such as *UTC*) for the given *date* value. Why this isn't just included in the *date* attribute I'll never know. The very last attribute specifies the *service* used, because it is possible for a user to choose their type of service dynamically at connect time.

An *accounting STOP* record adds the following attribute-value pairs, for statistical purposes:

Accounting_stop: (*protocol:* **string**,
addr: **string**,
disc-cause: **integer**,
disc-cause-ext: **integer**,
pre-bytes-in: **integer**,
pre-bytes-out: **integer**,
pre-paks-in: **integer**,
pre-paks-out: **integer**,
bytes_in: **integer**,
bytes_out: **integer**,
paks_in: **integer**,
paks_out: **integer**,
pre-session-time: **integer**,
elapsed_time: **integer**,
data-rate: **integer**)

The *protocol* attribute is a subset of a service, and is typically also detailed in the *authentication* table. But, since the user may dynamically choose a service, it is also possible that the protocol choice may be dynamic, so it must be logged in the event of a STOP record. The *addr* attribute has the same properties as in the *authorization* table. The *disc-cause* and *disc-cause-ext* attributes highlight the reasons as to why the session was terminated (hence generating the STOP record). The value for the *disc-cause* attribute will be a number, which represents a specific *disconnection code*. The possible values for the *disc-cause-ext* attribute are extended off of the *disc-cause* attribute, and are used for vendor-specific purposes.

Now, we need to break down the actual statistics that are reported. Any attribute with a "*pre-*" modifier represents any transactions that occur **before** authentication succeeds. Thus, everything else represents data that occurs during the course of the actual connection. With that in mind, *bytes-in* and *bytes-out* refer to the number of input bytes and output bytes, respectively, transferred between the NAS and the remote client. The *paks-in* and *paks-out* attributes represent the number of data packets that are input and output, respectively, during the course of the connection.

Similarly, the *pre-session-time* attribute represents the number of seconds that transpire between the time when the connection is first initiated to when it is finally authenticated. The *elapsed-time* attribute represents the duration of the connection, and is useful for NAS devices that do not maintain any sort of internal time. Finally, although the *data-rate* AV pair has been depreciated in more recent revisions of TACACS+, it is still used by Ameritech in order to report the speed of the connection between the NAS and the remote client.

SQL Implementation of the Raw_Accounting schema:

```
CREATE TABLE Raw_Accounting (log_date DATE,
                             NAS_hostname VARCHAR2(11),
                             userid VARCHAR2(11),
                             NAS_port VARCHAR2(16),
                             remote_phone_number CHAR(10),
                             type VARCHAR2(6),
                             NasUserName VARCHAR2(30),
                             task_id INTEGER,
                             timezone VARCHAR2(3),
                             service VARCHAR2(3),
                             protocol VARCHAR2(3),
                             addr VARCHAR2(15),
                             disc_cause INTEGER,
                             disc_cause_ext INTEGER,
                             pre_bytes_in INTEGER,
                             pre_bytes_out INTEGER,
                             pre_paks_in INTEGER,
                             pre_paks_out INTEGER,
                             bytes_in INTEGER,
                             bytes_out INTEGER,
                             paks_in INTEGER,
                             paks_out INTEGER,
                             pre_session_time INTEGER,
                             elapsed_time INTEGER,
                             data_rate INTEGER);
```

Calls_Completed:

During the course of this design, allusions have been made to another *accounting* table, used to store actual *calls* that the users completed. Now that the full schema for the *Raw_Accounting* relation has been developed, it is possible to discuss this second *accounting* relation. Basically, the *Calls_Completed* relation contains attempts to maintain all of the important attributes from a *Raw_Accounting* record. A presentation of the exact schema will clarify this point:

Calls_Completed (*userid:* **string**,
NAS-hostname: **string**,
NAS-port: **string**,
start_time: **date**,
stop_time: **date**,
duration: **date**,
tot-bytes-in: **integer**,
tot-bytes-out: **integer**,
tot-paks-in: **integer**,
tot-paks-out: **integer**)

Most of the fields in this table have already been explained, but a few are worth some special attention. The *start_time*, *stop_time*, and *duration* fields all revolve around the timing for the call. The *start_time* is culled directly from the *START* record, and the *stop_time* is taken from the *STOP* record. The *duration* is actually the difference between the aforementioned *start* and *stop* times.

SQL Implementation of the Calls_Completed schema:

```
CREATE TABLE Calls_Completed ( userid VARCHAR2(11),
                               NAS_hostname VARCHAR2(11),
                               NAS_port VARCHAR2(16),
                               start_time DATE,
                               stop_time DATE,
                               duration INTEGER,
                               tot_bytes_in INTEGER,
                               tot_bytes_out INTEGER,
                               tot_paks_in INTEGER,
                               tot_paks_out INTEGER,
                               PRIMARY KEY (callid));
```

Potential Queries revisited:

Now it is time, once again, to revisit the potential queries that were listed in the data model description. These queries now need to be anayzed (and ordered) in terms of their cost-of-execution. This involves considering the cost the basic query operations, and then examining which operations will be necessary in each query. None of these queries make use of the *Cartesian product* operator, and only a few of them use the *join* operator. As before, the list of queries will be examined by relation, with the most costly queries having the higher number.

Authentication:

1. Determine the total number of users, as well as the number of *active* users (those who have a password other than the default).
2. Display a list of users that have at least one password set to *"tempass1"*. This is the default password, set when an account is first created. Thus, this query can be used to show which users haven't used their accounts as of yet, which could lead to a potential security issue.
3. Find all users that have both the special *"admin"* type attribute, as well as an uninitialized password. This is a very big security hole.
4. Emulate an *Authentication* request – search for a user, and return his/her values.

Authorization:

5. Display users of type *admin* that have very permissive security settings.
6. Display a list of *"secure" userid*'s – those that have values for both the *inacl*, and *outacl* attributes. Furthermore, their *routing* attribute should be set to *false*.
7. Display a list of all users who have very *"limited"* access – i.e. their *timeout* value is less than or equal to 60 minutes.
8. Emulate an *Authorization* request – search for a user, and return his/her attributes.

Accounting:

9. Find the last login for every user, sorted in reverse order.
10. Display a *"call history"* (complete list of calls) for a given user.
11. Display a listing of users that are logged in on a given NAS for a given time period.

Integrity Constraints

The integrity constraints that can be enforced consist of key values and SQL "CHECK" operations. Given this, we'll proceed to analyze the integrity constraints for each of the three separate categories of data:

Authentication:

In this table, the *userid* field is a primary key. Each *userid* must be unique for a customer, and since each customer has their own table, each *userid* must be unique within a table. If *userid's* were **not** unique, it's obvious to see how the TACACS+ server would break. When an authentication request comes in, the server attempts to retrieve the proper record from the proper *authentication* table, based upon the only attribute that it currently has – the *userid*. Hence, if multiple records with the same *userid* existed, only the first such record would ever be found, and users with subsequent records would be denied the ability to login to the NAS.

The only interesting application of the SQL *"CHECK"* functionality in this table might be to make sure that every user record has at least *one* password configured. The *login*, *pap*, *chap*, and *global* attributes are all used to specify passwords, but any one of them may take on a NULL value, depending on the user requirements. However, in order to authenticate with the NAS, at least one password is necessary. Thus, records that do not contain at least one password should be allowed to enter the database. However, it is also possible to rely on Ameritech's password administration utility in order to enforce this constraint. For the sake of performance, it makes sense to rely on the assumption that we will be given valid data to work on (since it is all coming from a computer program, and not directly from users). Thus, the *CHECK* operation will not be implemented.

Authorization:

The *authorization* tables require the use of the *userid* attribute, as specified in the *authentication* table. Thus, the *userid* attribute in this table must be a *foreign key* with respect to the *userid* attribute in the *authentication* table. Furthermore, it's possible for a user record to not need any special authorization parameters, so no *SQL CHECKs* can accurately be performed on this data set.

Accounting:

This raw data in this category has **no** integrity constraints. Typical constraints, such as the *userid* attribute, don't hold in this relation – because it's possible for the system to log *userid's* that **don't exist** in the *authentication* table, for example. However, when the *call* table is constructed, some integrity constraints will magically appear. For any given call, the combination of the *task_id* and *NAS-hostname* must be unique. Thus, these two fields together can form a *superkey* for this relation. (Note that each NAS generates it's own *task_id's*, so the *task_id* cannot be a key all by itself).

The issue of data redundency should be addressed when discussing the *Accounting* relations, since it contains much redundant data. The *raw_accounting* table records values that may repete those stored in the *authentication* table, for example (such as the *userid* and *name*). At first glance, the inclusion of the *NasUserName* attribute (which parallels the *authentication name* attribute) in the *raw accounting* information seems very odd and redundant. However, upon further consideration, an argument for the inclusion of this field appears. Basically, since each *accounting* record represents a transaction that occurred at a fixed point and time, it is therefore necessary to capture the corresponding *name* value for that given period of time. If this attribute were made dynamic (basically, this data could be queried-for whenever it was needed), it could be problematic if the user changes his/her *name* value. The same can be said for the inclusion of the static *userid* field: if a *userid* were removed from the *Authentication* table, we would still want to maintain the accounting information that the user generated. Or even still, we want to be able to account for transactions where an end-user attempts to use an *userid* that isn't currently **in** the *Authentication* table.

Functional Dependencies

The functional dependencies in this database design appear to be weak, at best. It seems that although the TACACS+ data contains many intrinsic functional dependencies, it isn't really possible to deal with these from a database design perspective. In general, there are only two different sorts of functional dependencies in this database: one exists between the *Authentication* and *Authorization* relations, and the other is actually a class of dependencies, that are intrinsic to the TACACS+ data.

The functional dependency that exists between the *Authentication* and *Authorization* tables revolves around the sole attribute that they share – namely the *userid*. I have connected these two relations together by making the *userid* a *primary key* in the *Authentication* relation, and a *foreign key* in the *Authorization* relation. Thus, the RDBMS will ensure that SQL *DELETE* or *UPDATE* statements don't violate the integrity of matching records. Thus, from this dependency we realize two different classes of anomalies: those that occur during unmatched insertions, and those that occur when unmatched deletions are attempted.

The case of insertion anomalies can be thought of thusly: the only way that an insertion anomaly can occur is if a record is inserted into the one relation, but not the other. The RDMBS will allow the case that the record is inserted into the *Authentication* relation but not the *Authorization* relation. However, it will block (with an error) the case that when a record is inserted into *Authorization* before *Authentication*. In either case, these possibilites can only come about due to programmer error, not user error. Ameritech has developed a sort of *"Administration Client"*, by which users are actively maintained on the system. Thus, it is up to this client to insert the proper records in the proper order. We'll assume that programmer error will not pose a very terrible problem (if an error is encountered, it will be fixed).

Fortunately, the case of deletion anomalies can be handled more directly in SQL. Since this type of anomaly occurs frequently in databases, SQL92 provides the "ON

DELETE CASCADE" construct, which can be applied to any relations that have a *foreign key*. Basically, this construct allows the RDBMS to automatically clean-up whenever a tuple is deleted from the specified relation. Thus, we can ensure, at the RDBMS level, that no deletion anomalies will occur, thanks to this construct. This construct modifies the previously stated *Authorization* SQL, so it now appears like this:

```
CREATE TABLE Authorization ( userid VARCHAR2(11),
                             inacl INTEGER,
                             outacl INTEGER,
                             timeout INTEGER,
                             idletime INTEGER,
                             addr VARCHAR2(15),
                             routing VARCHAR2(5),
                             route VARCHAR2(49),
                             FOREIGN KEY (userid) REFERENCES Authentication
                                  ON DELETE CASCADE);
```

The final class of dependencies to be discussed revolves around those that are intrinsic to the TACACS+ dataset. For example, several of the attributes in the *Accounting* relation depend upon their respective values in the *Authentication*, or *Authorization* relations. One such attribute is the *NasUserName* value, which is culled directly from the *name* value in the *Authentication* relation. I was able to get a clear view of these dependencies when I tried to implement my scripted *faux* data – many of the values that I generated didn't make sense, because I didn't reference previously generated values for the other relations. However, beyond recognizing that these dependencies exist, there isn't much that can be done to minimize them on the RDBMS level. They seem to all be *trivial* dependencies (i.e. one value directly influences another), and as such, don't pose a major problem to my database design.

Query Implementations

Authorization:

1. *Emulate an Authentication request – search for a given userid, and return the values found.*

RA:

$$\sigma_{userid='given\_userid'}(Authentication)$$

TRC:

$$\{U \mid U \in Authentication \wedge U.userid = 'given\_userid'\}$$

SQL:

```
SELECT  *
FROM    Authentication
WHERE   userid = “given_userid”
```

2. *Display a list of users (userid and name) that have at least one password set to the default, “tempass1”.*

RA:

$$\pi_{userid,name}\begin{pmatrix}\sigma_{\text{login='cleartext tempass1'}\vee\text{pap='cleartext tempass1'}}(Authentication)\vee\\ \sigma_{\text{chap='cleartext tempass1'}\vee\text{global='cleartext tempass1'}}(Authentication)\end{pmatrix}$$

TRC:

$$\left\{\begin{matrix}A.userid\\A.name\end{matrix}\middle| A\in Authentication \wedge \begin{pmatrix}A.login='cleartext\ tempass1'\vee\\ A.pap='cleartext\ tempass1'\vee\\ A.chap='cleartext\ tempass1'\vee\\ A.global='cleartext\ tempass1'\end{pmatrix}\right\}$$

SQL:

```
SELECT  A.userid, A.name
FROM    Authentication A
WHERE   login = ‘cleartext tempass1’
        OR pap = ‘cleartext tempass1’
        OR chap = ‘cleartext tempass1’
        OR global = ‘cleartext tempass1’;
```

3. *Find all users that are administrators (i.e. their type attribute is set to “admin”) and have an uninitialized password.*

RA:

$$\pi_{userid,nam}\begin{pmatrix}\begin{pmatrix}\sigma_{\text{login='cleartext tempass1'}\vee\text{pap='cleartext tempass1'}}(Authentication)\vee\\ \sigma_{\text{chap='cleartext tempass1'}\vee\text{global='cleartext tempass1'}}(Authentication)\end{pmatrix}\wedge\\ \sigma_{type="admin"}(Authentication)\end{pmatrix}$$

TRC:

$$\left\{\begin{matrix}A.userid, A.name \mid A\in Authentication \wedge A.type='admin'\wedge\\ \begin{pmatrix}A.login='cleartext\ tempass1'\vee\\ A.pap='cleartext\ tempass1'\vee\\ A.chap='cleartext\ tempass1'\vee\\ A.global='cleartext\ tempass1'\end{pmatrix}\end{matrix}\right\}$$

SQL:

```
SELECT  A.userid, A.name
FROM    Authentication A
```

```
        WHERE   A.type = 'admin'
                AND A.userid IN (
                        SELECT  A2.userid
                        FROM    Authentication A2
                        WHERE   A2.login = 'cleartext tempass1'
                                OR A2.pap = 'cleartext tempass1'
                                OR A2.chap = 'cleartext tempass1'
                                OR A2.global = 'cleartext tempass1')
```

4. *Determine the total number of users, as well as the number of active users.*

SQL:
```
SELECT  COUNT (A1) AS total, COUNT (A2) AS active
FROM    Authentication A1, Authentication A2
WHERE   A2.userid IN (
                SELECT  A3.userid
                FROM    Authentication A3
                WHERE   A3.login <> 'cleartext tempass1'
                        OR A3.pap <> 'cleartext tempass1'
                        OR A3.chap <> 'cleartext tempass1'
                        OR A3.global <> 'cleartext tempass1')
```

Authorization:

5. *Emulate an Authorization request – search for a given userid, and return the values found.*

RA: $\sigma_{userid='given\_userid'}(Authorization)$

TRC: $\{U \mid U \in Authorization \wedge U.userid = 'given\_userid'\}$

SQL:
```
SELECT  *
FROM    Authorization
WHERE   userid = 'given_userid';
```

6. *Display userid's that are "secure" (have values for inacl and outacl attributes; routing attribute that is false).*

RA: $\pi_{userid}(\sigma_{inacl <> NULL \wedge outacl <> NULL \wedge routing = FALSE}(Authorization))$

TRC: $$\left\{\begin{matrix} Z.userid \mid Z \in Authorization \wedge Z.inacl <> NULL \wedge \\ Z.outacl <> NULL \wedge routing = FALSE \end{matrix}\right\}$$

SQL:
```
SELECT  userid
FROM    Authorization
WHERE   inacl IS NOT NULL AND outacl IS NOT NULL
        AND routing = 'FALSE';
```

7. *Display all userids who have a timeout value that is less than or equal to 60 minutes.*

RA: $\pi_{userid}(\sigma_{timeout \leq 3600}(Authorization))$

TRC: $\{Z.userid \mid Z \in Authorization \wedge Z.timeout \leq 3600\}$

SQL:
```
SELECT  userid
FROM    Authorization
```

```
         WHERE   timeout <= 3600;
```

8. *Display a list of administrators who have very permissive security settings.*

RA:

$$\pi_{userid,name}\left(\begin{array}{l}\left(\sigma_{inacl=NULL \lor outacl=NULL \lor routing=TRUE}\left(Authorization\right)\right)\\ \infty\sigma_{type='admin'}\left(Authentication\right)\end{array}\right)$$

TRC:

$$\left\{\begin{array}{l}Z.userid\\ A.name\end{array}\middle| Z \in Authorization \land (\exists A)\left(\begin{array}{l}A \in Authentication \land A.type='admin'\\ \land\left(\begin{array}{l}Z.inacl = NULL\\ \lor Z.outacl = NULL\\ \lor Z.routing = TRUE\end{array}\right)\end{array}\right)\right\}$$

SQL:

```
SELECT   Z.userid, A.name
FROM     Authorization Z, Authentication A
WHERE    A.userid = Z.userid
         AND A.type = 'admin'
         AND (Z.inacl IS NULL
              OR Z.outacl IS NULL
              OR Z.routing = 'TRUE');
```

9. *Query to find the "worst" administrators, basically those that both have the default password, and permissive authorization settings.*

RA:

$$\pi_{userid,name}\left(\begin{array}{l}\left(\sigma_{inacl=NULL \lor outacl=NULL \lor routing=TRUE}\left(Authorization\right)\right)\\ \infty\sigma_{type='admin'}\left(Authentication\right)\end{array}\right)\cap$$

$$\pi_{userid,name}\left(\begin{array}{l}\left(\begin{array}{l}\sigma_{\text{login='cleartext tempass1'}\lor\text{pap='cleartext tempass1'}}\left(Authentication\right)\lor\\ \sigma_{\text{chap='cleartext tempass1'}\lor\text{global='cleartext tempass1'}}\left(Authentication\right)\end{array}\right)\land\\ \sigma_{type="admin"}\left(Authentication\right)\end{array}\right)$$

TRC:

$$\left\{\begin{array}{l}Z.userid\\ A.name\end{array}\middle| Z \in Authorization \land (\exists A)\left(\begin{array}{l}A \in Authentication \land A.type='admin'\\ \land\left(\begin{array}{l}Z.inacl = NULL\\ \lor Z.outacl = NULL\\ \lor Z.routing = TRUE\end{array}\right)\end{array}\right)\right\}$$

$$\cap\left\{\begin{array}{l}\begin{array}{l}A.userid,\\ A.name\end{array}\middle| A \in Authentication \land A.type='admin'\land\\ \left(\begin{array}{l}A.login='cleartext\ tempass1'\lor\\ A.pap='cleartext\ tempass1'\lor\\ A.chap='cleartext\ tempass1'\lor\\ A.global='cleartext\ tempass1'\end{array}\right)\end{array}\right\}$$

SQL:

```
SELECT   Z.userid, A.name
FROM     Authorization Z, Authentication A
WHERE    A.userid = Z.userid
```

```
                AND A.type = 'admin'
                AND (Z.inacl IS NULL
                     OR Z.outacl IS NULL
                     OR Z.routing = 'TRUE')
        INTERSECT
        SELECT  A.userid, A.name
        FROM    Authentication A
        WHERE   (A.type='admin' AND A.login='cleartext tempass1') OR
                (A.type='admin' AND A.pap='cleartext tempass1') OR
                (A.type='admin' AND A.chap='cleartext tempass1') OR
                (A.type = 'admin' AND A.global='cleartext tempass1');
```

Accounting:

10. Display a listing of users that are logged in on a given NAS for a given time period.

RA:

$$\left(\pi_{\substack{userid,name,\\start\_date,\\stop\_date}}\left(\sigma_{stop\_time>'given\_time'}(Calls\_Completed)\right)\bowtie\left(\pi_{\substack{userid,\\name}}(Authentication)\right)\right)$$

$$\bowtie\left(\pi_{userid,name,log\_date}\left(\sigma_{\substack{type='start'\wedge log\_date<'given\_time'\wedge\\NAS-hostname='give\_NAS'}}(Raw\_Accounting)\right)\right)$$

TRC:

$$\left\{\begin{array}{l|l} A.userid & \\ A.name & \\ C.start\_time & A\in Authentication\wedge \\ C.stop\_time & \end{array}\left(\begin{array}{l}(\exists R)\left(\begin{array}{l}R\in Raw\_Accounting\wedge\\ R.userid=A.userid\wedge\\ R.type='START'\wedge\\ R.log\_date<'given\_time'\wedge\\ R.NAS-hostname='given\_NAS'\end{array}\right)\vee\\ (\exists C)\left(\begin{array}{l}C\in Calls\_Completed\wedge\\ C.userid=A.userid\wedge\\ C.stop\_time>'given\_time'\wedge\\ C.NAS-hostname='given\_NAS'\end{array}\right)\end{array}\right)\right\}$$

SQL:

```
SELECT  A.userid, A.name, C.start_time, C.stop_time
FROM    Authentication A, Calls_Completed C
WHERE   A.userid = C.userid
        AND C.stop_time > 'given_time'
        AND C.NAS_hostname = 'given_NAS'
UNION
SELECT  R.userid, R.NasUserName, R.log_date, TO_DATE(NULL)
FROM    Raw_Accounting R
WHERE   R.type = 'START'
        AND log_date < 'given_time'
        AND R.NAS_hostname = 'given_NAS';
```

This query makes several assumptions that should be explained. Firstly, in joining the *Authentication* and *Calls_Completed* relations, it is *assumed* that the *Authentication* table contains accurate names. Since this is only for user-presentation, so its significance can be depreciated. The second assumption is made when records are gathered from the *Raw_Accounting* table. The only START records that are in said table are those that don't have a corresponding *STOP* – the legitimacy of their state, however, is unknown. Thus, it is entirely likely that this query will list tuples where no call is currently in progress. However, since a direct query cannot be performed on the NAS, there really isn't any way to know this information. Finally, this query assumes that the TACACS+ server logs *bad username* login attempts as *STOP* records.

11. Find the last login for every user, sorted in reverse order.

RA:

$$\pi_{\substack{userid,name,NAS-hostname,\\ NAS-port,start\_time}}\left(\begin{array}{l}\sigma_{\substack{start\_time\geq'start\_period'\wedge\\ start\_time<'stop\_period'}}\left(Calls\_Completed\right)\infty\\ \pi_{userid,name}\left(Authentication\right)\end{array}\right)$$

TRC:

$$\left\{\begin{array}{l|l}C.userid, & \\ A.name, & \\ C.NAS-hostname & C\in Calls\_Completed\wedge \\ C.NAS-port & \\ C.start\_time & \\ \multicolumn{2}{l}{(\exists A)\left(\begin{array}{l}A\in Authentication\wedge\\ C.userid=A.userid\wedge\\ C.start\_time\geq'start\_period'\wedge\\ C.start\_time<'end\_period'\end{array}\right)}\end{array}\right\}$$

SQL:

```
SELECT   C.userid, A.name, C.NAS_hostname, C.NAS_port,
         C.start_time
FROM     Calls_Completed C, Authentication A
WHERE    C.start_time >= 'start_period'
         AND C.start_time < 'stop_period'
         AND C.userid = A.userid
ORDER BY C.start_time DESC;
```

12. Display a list of "malformed" logfile entries – unmatched START and STOP records in the Raw_Accounting table.

RA:

$$\pi_{\substack{userid,NasUserName,\\ Nas-hostname,\\ Nas-port,type}}\left(\sigma_{type='STOP'\vee(type='START'\wedge(SYSDATE-log\_date)>20)}\left(Raw\_Accounting\right)\right)$$

TRC: $$\left\{ \begin{matrix} R.userid, \\ R.NasUserName, \\ R.type, \\ R.NAS-hostname \\ ,R.NAS-port \end{matrix} \middle| \begin{pmatrix} R \in Raw\_Accounting \wedge \\ R.type = 'STOP' \vee \\ \left(R.type = 'START' \wedge \left(SYSDATE - R.log\_date \right) > 20 \right) \end{pmatrix} \right\}$$

SQL:
```
SELECT   userid, NasUserName, NAS_hostname, NAS_port, type
FROM     Raw_Accounting
WHERE    type = 'STOP';
```

Due to problems with the Oracle *SYSDATE* function, as well as with comparing data values of type *"DATE"* in Oracle, the part of this query that dealt with the start records wasn't included. For the life of me, I just *couldn't* figure out how to drive the *SYSDATE* function. In fact, during my travails, I manged to crash the *SQL*Plus* client a number of times before finally removing that aspect of the query.

13. *Display a "call history" (complete list of calls) for a given user.*

RA: $\sigma_{userid='given\_userid'}(Calls\_Completed)$

TRC: $\{C | C \in Calls\_Completed \wedge C.userid = 'given\_userid'\}$

SQL:
```
SELECT  *
FROM    Calls_Completed
WHERE   userid = 'given_userid'
```

Query Efficiency

In order to explore the realm of query optimizations that might apply to my design, it is best to divide the following discussion into two parts: One discussing the *Authentication* and *Authorization* relations, the other dealing with the *Accounting* relations. The scope of the queries that deal with *Authentication* and *Authorization* are some of the more complex within this project (they actually use the *join* operation). For the data in its current configuration, the RDBMS must search on practically all of the attributes: the only ones that are never the subject of search are *name*, *addr*, and *route*.

Requiring this many search keys, to be used together in a variety of different combinations, creates a difficult indexing environment.

The easiest queries to evaluate are those that don't require a *join*, and search based upon one or two attributes. The next step up (in terms of complexity) is those queries that don't require a *join*, but search based upon a large number of attributes. The next complexity level is comprised of those queries that require the use of the *join* operation, but only search based upon a limited number of attributes. Finally, the most complex queries of this set are those that require the use of both the *join* operation and many search keys.

The reasoning behind this heirarchy is somewhat obvious – the number of disk operations is directly proportional to the number of records that must be read. This number of records is proportional to the size (*join* leads to a greater size) and efficiency of the index structure (related to the number of search keys). Thus, the no *join* operation and few search keys are used (as in query number one, for example), then an efficient hash-based index structure can achieve an *O(num_results)* performance, even when the number of records range in the thousands.

However, it may be possible to ensure a more consistent level of performance across queries by modifying the schema for the *Authentication* and *Authorization* relations. In principal, two possible modifications exist: the *Authentication* table could be optimized by folding identical passwords into a single *global* attribute, or the *Authentication* and *Authorization* relations could be combined into one relation. When a client attempts to login to the network, TACACS+ searches through the *Authentication* information for a password, in order to match that with the user's request. TACACS+ is very flexible in its password-searching approach – it allows the password information to be in a number of locations. It first searches the user's *authentication* record, looking to see if a specific password has been defined for the authentication method that they are attempting. For example, if a user is attempting to start a PPP session via CHAP[2] password authentication, the TACACS+ server will first attempt to find a *chap* attribute in the user's *authentication* record. If no such attribute is found, it will then check for the *global* attribute. If this attribute isn't found, it will then search for a *CHAP* attribute in any groups to which the user claims membership. Failing this, it will then search these groups for a *global* attribute. Thus, it's possible to take advantage of this password-searching heirarchy in order to reduce the number of attributes in the *Authentication* relation.

Due to historical reasons, Ameritech's current password-administration client creates TACACS+ *authentication* records that contain the same password for the *pap*, *chap*, *login*, and *global* fields. It then enforces this password duplicity across password changes, so the state of these passwords seems to be fairly reliable. Thus, it is possible to take a bit of a *shortcut* in the database design, by removing the *pap*, *chap*, and *login* attributes, so that password authentication can rely solely on the *global* attribute. Thus, any queries that attempted to determine if a user had an *"insecure"* password would be vastly simplified. Unfortunately, the problem with this optimization is that it unduly limits future modifications to the password-administration client. For example, if Ameritech decided to allow different *pap* and *chap* passwords in the future, not only would the *Authentication* relation require modification, but so would most of the queries that deal with this relation. Thus, a decision about future password flexibility needs to be made in order to decide if this particular optimization should be made. For the purposes of this design, the current structure will be left in place, in order to achieve the maximum level of compatibility with Ameritech's current TACACS+ infrastructure.

Another possible optimization concerns simply merging the *Authentication* and *Authorization* relations. Currently, each relation doesn't contain a very large number of attributes (especially if all of the duplicate passwords were to be removed from the *Authentication* relation); thus it is feasible to simply merge them, in order to eliminate the use of the *join* operation. Interestingly enough, the TACACS+ server was designed to keep the *authentication* and *authorization* information together in the context of the same configuration file. Thus, the separate relations that I have created for the purposes of this database design are in fact somewhat artificial. Furthermore, it seems that in order to effectively handle an authentication request, the TACACS+ server must access not only the *authentication* information, but also the *authorization* information as well. If these two sets of information were in the same relation, it would speed up the most frequently used query. Consquently, it seems like this might be a very beneficial design modification. Unfortunately, this change could severely limit the scalability of the database. Attribute growth is a very definite possibility that must be taken into consideration. For example, Ameritech is moving towards the use of VPDNs[3] for some of their customers. In order to provide this functionality, many new attributes would be needed in both the *Authentication* and *Authorization* relations. This is but one example of the many possible modifications to the remote access services that would require the addition of further attributes in either the *Authentication* or *Authorization* relations. Thus, in terms of scalability, maintaining separate *Authentication* and *Authorization* tables could be very beneficial to stable query performance. Under the *“unified”* model, all queries will suffer equally with the addition of each attribute. In the *“split”* model, only the queries that depend on the affected relation may suffer a performance decrease. Yet again, I seemed to be faced with another major design decision, between the scalability of the number of simultaneous authentication requests that a TACACS+ server can handle (a vote for the *unified* model) and the performance of the data-analysis queries (a vote for the *split* model). If this were to be implemented as a *real* project, I would recommend that the *unified* model be used. But, for the purposes of this assignment, it is more interesting to implement the *split* case, so my design will continue in this vein.

The final optimization that can be made is to eliminate *nested queries* whenever possible. *Nested queries* require much more work on the part of the RDBMS, because it must do the inner query first, and join the results of this query with the relations in the outer query. Methods such as pipelining can be used by the RDBMS in order to speed query processing to a degree, but no method can get around the fact that a *join* must be performed. Thus, for the sake of performance, it makes sense to rewrite these sorts of queries whenever possible, producing a non-nested equivalent. Consequently, two of the previously stated queries, numbers three and four from *Authentication*, can be rewritten in the following manner (and have been for my implementation):

```
3.  SELECT  A.userid, A.name
    FROM    Authentication A
    WHERE   (A.type = 'admin' AND A.login = 'cleartext tempass1') OR
            (A.type = 'admin' AND A.pap = 'cleartext tempass1') OR
            (A.type = 'admin' AND A.chap = 'cleartext tempass1') OR
            (A.type = 'admin' AND A.global = 'cleartext tempass1');

4.  SELECT  COUNT (*) AS num_users
    FROM    Authentication;

    SELECT  COUNT (A2.userid) AS active_users
    FROM    Authentication A2
    WHERE   A2.login <> 'cleartext tempass1'
            OR A2.pap <> 'cleartext tempass1'
            OR A2.chap <> 'cleartext tempass1'
```

```
              OR A2.global <> 'cleartext tempass1';
```

Moving on, the *Accounting* relation represents a special challenge, due to the nature of the system-logging information that it stores. One very important performance-affecting design decision has already been made – that of creating the *Calls_Completed* table, in order to maintain the implied state information from the *Raw_Accounting* records. Many of the most important queries that Ameritech (and their customers) desire can be performed on the *Calls_Completed* table, saving the hassle of having to match up the appropriate *START*, *STOP*, and *UPDATE* records every time. Beyond this decision, however, further attention can be paid to both the *Raw_Accounting* and *Calls_Completed* tables.

Of the many attributes contained in the *Raw_Accounting* relation, only the *date*, *task_id*, and *NAS-hostname* attributes will ever be used as search keys. Furhtermore, this relation is only used in three different queries (or transactions): one creates the *Calls_Completed* table, the other two simply analyze the leftover records. Thus, under these conditions, careful indexing can mitigate the relative difficulty of dealing with the large size of the *Raw_Accounting* records. When generating the *Calls_Completed* relation, only the *task_id* and *NAS-hostname* attributes are required as search keys. The other queries will only need to search based upon the *date* and *type* attributes. Thus, two different hash-based index structures could be created to cover each of these situations, in order to create the highest-performing environment possible.

The queries and transactions that run on the *Calls_Completed* table are slightly different in nature than those that have previously been examined. This relation doesn't contain any single attribute that can be treated as the *key* for the relation. Thus, several attributes must be used together in order to determine unique tuples. These attributes consist of the *userid*, *NAS-hostname*, *date*, and *NAS-port*. This means that every query could potentially need to search based upon all of these attributes, plus whatever is necessary in order to gain meaning from the query. Thus, in order to increase potential query performance, it might make sense to reduce the complexity of this *aggregate key*.

One method to reduce the number of attributes in the *aggregate key* would be to import the *task_id* attribute from the *Raw_Accounting* relation. The nature of the *task_id* attribute is such that it is only required inorder to determine which combination of *START*, *STOP*, and *UPDATE* records actually form a *call*. Thus, once all of the components of the *call* have been located, the *task_id* is essentially irrelevant. My initial design decision was to attempt to keep the size of the *Calls_Completed* relation down by only including those attributes from *Raw_Accounting* that were absolutely necessary. Thus, the *task_id* has been left out of the *Calls_Completed* relation. However, if it were added, the *aggregate key* for the relation could be reduced to the combination of the *NAS-hostname* and *task_id* attributes. Thus, another integer field per record could be sacrificed in order to reduce the number of attributes required in the *aggragate key* by half.

Another method possible method would be to simply generate a unique integer for every record that is added to the *Calls_Completed* table. The addition of a *callid* field would add the same amount of data as the *task_id* attribute, but have the additional advantage becoming the *sole primary key* of the relation. Thus, with only a little more PL/SQL and memory space overhead, a radically less complex key can be generated for the *Calls_Completed* relation. To me, this seems to make good design sense, so the *Calls_Completed* relation will now be implemented as follows:

```
CREATE TABLE Calls_Completed ( callid INTEGER,
                               userid VARCHAR2(11),
                               NAS_hostname VARCHAR2(11),
                               NAS_port VARCHAR2(16),
start_time DATE,
stop_time DATE,
duration INTEGER,
tot_bytes_in INTEGER,
tot_bytes_out INTEGER,
tot_paks_in INTEGER,
tot_paks_out INTEGER,
PRIMARY KEY (callid));
```

Other than the performance considerations that have been made, there aren't many other aspects of the *Accounting* relations to analyze. None of these queries that could possibly be implemented could use the *join* operation, thus no effort needs to be expended optimizing for it. Thus, I am free to consider the implications that data growth will have on these relations.

There is absolutely no doubt in my mind that this database design will have to withstand quite a large amount of data. In fact, Ameritech currently has one customer that has over 4,000 user accounts. Not only does this translate to large *Authentication* and *Authorization* relations, but also to massive *Accounting* relations (the user activity in this sort of scenario could be rather high). Furthermore, as Ameritech expands this service (partly due to the ease at which an RDBMS solution allows them to add customers), the amount of data that must be handled will grow in multiple ways. Not only will more user accounts be necessary, but also the pace of Internet technology will assure the fact that the number of attributes required in all of the relations will increase. Furthermore, as Ameritech's customer base increases, the number of access servers that they will employ will increase. Thus, the system will be capable of handling more users simultaneously, and the number of *Accounting* records that enter the system will grow exponentially when compared to the number of users added. Fortunately, I feel that my design, combined with a well-maintained *Oracle Database*[4] will not only be able to handle the challenge, but will also afford better scalability than the current solution as data needs increase.

The current text-file based TACACS+ software implements a memory-based hashing algorithm in order to search for *Authentication* records efficiently. When both the size and number of the records is small, this algorithm will provide very efficient performance – probably better than an RDBMS. However, as the data grows both in size and in number, this algorithm will continue to require additional memory. It is at this point, where the RDBMS will be able to take the performance lead. Since the database software has been designed in order to efficiently handle large sets of data, it should be able to outpace the rather simplistic hashing algorithm used in TACACS+.

Finally, beyond simple data growth, the possibility of changing requirements for the database also needs to be discussed. It is quite possible that different queries will be needed down the road, as customers ask for more statistics on their equipment, and as Ameritech expands the service. However, the nature of these relations is such that these new types of queries will be relatively bounded in what they can do. For example, there won't *ever* be any joins to deal with when concerning the *Accounting* information. Unless some radical, fundamental shift in the database schema is made, all of the *Accounting* queries will be contained to either the *Calls_Completed* or *Raw_Accounting* tables (or sometimes both, but not in *joined* fashion). Thus, I feel that my current design is sound enough to deal with the inevitable growth of the functionality that it will have to support.

Transactions

Beyond simple queries, the RDBMS needs to support many other functions in order to handle every aspect of the TACACS+ environment. All of these functions can be considered as *transactions*, either requiring either special SQL commands, or full-blown programs in order to process. Unfortunately, problems encountered with the Oracle database in the lab, as well as time constraints have limited what I have been able to accomplish. Thus, I will divide my discussion of transactions into two categories: those that *could* be implemented, and the one that I *did* implement. For those transactions that were left unimplemented, some basic description will be provided, as well as a rough examination of possible implementations. Finally, a detailed discussion will be given to the transaction that was implemented.

Authentication:

1. The database needs to support the addition, deletion, and modification of users. Currently, this is implemented through the aforementioned *Administration Client*. The TACACS+ server has been extended to also accept *administration* packets from this software program. Hence, the current implementation involves users connecting to the TACACS+ server, and sending their modifications over. The TACACS+ server then writes these changes to its local files, and reloads them into memory (if necessar). Thus, the easiest thing to do would be to keep the same mechanism, but modify the TACACS+ server to use the appropriate SQL commands in order to make these modifications directly on the database. However, another possibility would be to depreciate the use of the *administration client* in order to make use of some sort of graphical front-end to the database itself. There are many ramifications to this approach, but it would definitely be worth exploring in detail.
2. Another useful feature would be some sort of transaction that supports password aging. In the current TACACS+ sever, this feature is left unimplemented. Consequently, thanks to the ease with which the database can manipulate user-records, we sould be able to easily add this functionalit. The most likely

implementation of this feature would be some sort of script that runs through the data on a regular schedule. It would compare the last update time for each user record to a certain threshold. If exceeded, it would remove the current password. Consequently, the next time that the user attempted to login, they would be required to enter a new password.

Authorization:

3. For this relation, all that is necessary is a method in order to modify user records. This would most likely be implemented alongside the corresponding *Authentication* transaction.

Accounting:

4. Another useful statistic is the number of calls per port (this could either be done over a certain span of time, or as a running total). The underlying purpose is to look for ports that are either underutilized (they could be broken), or over-utilized (could require expansion of the NAS). This is currently implemented via a *perl* script that parses the *accounting* information, matching *START* and *STOP* records, and computing the necessary statistics. The information that it collects is exported as a collection of comma-separated records, which are then imported into *Microsoft Excel*. It is hoped that some of the vendor-supplied tools might afford a better method to both accrue and display the data, so that it might be more possible to gain a higher level of "*interactivity*" with the data.
5. A tranasaction to determine the number of simultaneous calls handled by the NAS in an hour. This transaction is important, because customers pay Ameritech per port, and they need to see that they are getting their money's worth. This could be implemented via a script that loops through every hour in the *Calls_Completed*, tabulation ghte

number of calls that are *"open"* for that hour. A call can be deemed *"open"* if either the starting time occurs within the given hour, or if the duration of the call moves into the given hour. The final step in this process would again be the user presentation. Currently, this is done by the aforementioned *Excel-method*, so it probably makes sense to implement this transaction alongside the former transaction.

6. A transaction to tabulate a number of statistics on the user-level. Basically, it is important to see things such as the number of calls per user and the total time that said user has spent logged into the NAS, as well as the total amount of data that they have sent/received. Collapsing all of this information into one place makes it easier to keep tabs on what each user is doing with the system. An extension of this transaction would be to maintain some sort of history, and *"flag"* users that demonstrate uncharacteristic behavior (this might indicate that a *"malicious"* third party has gained control of this particular user account).
7. Another useful transaction would be to generate a histogram of call durations. This transaction would display the call durations based upon a set of predfined categories. Each category would represent a different length of time, such as 0 – 1 hours, 1 – 2 hours, etc.

Generation of "Calls_Completed":

As has been previously discussed, the notion of a *Calls_Completed* relation simplifies greatly a number of problems that are encountered when dealing with the *Accounting* data. Thus, in order to complete this assignment, it was necessary to implement a transaction that generates the *Calls_Completed* table, based upon records contained in the *Raw_Accounting* table. The specific implementation of this transaction will follow in *"Appendix B"*, and the current discussion will be limited to a discussion of this implementation.

In general, this transaction did prove itself to be invaluable. As witnessed, the implementation of many a query was greatly simplified by the presence of a *Calls_Completed* table. Unfortunately, my implementation of this transaction did have some problems. Basically, the nested loop structure of this script causes an undue amount of computation on the RDBMS' end. As such, when given roughly four thousand *Raw_Accounting* records, this transaction required over half-an-hour to run to completion in the lab. This level of performance is unacceptable, if this transaction is to be run agains the *Raw_Accounting* table at regular intervals (as is expected – many of the queries require as up-to-date information from the *Calls_Completed* relation as is feasible). Thus, if this project were to be implemented in the "real world", much effort would have to be put into a design that performs better.

Implementation

The discussion surrounding the implementation of my design will be broken up across several appendicies, due to the lengthy nature of some the inputs and outputs. Thus, this discussion will deal with the *"problems encountered"* during the implementation of my design.

The first problem that I encountered revolved around the sample data. It was impossible for me to get "real world" data, so I implemented a *perl* script in order to generate some *faux* data. A full discussion of this solution appears in *Appendix A*.

The second, and much more difficult hurdle, was the actual RDBMS that I used. I wanted to use an *Oracle* database for this assignment, because *Oracle* currently has quite a bit of support within Ameritech. Furhtermore, I also know that *Oracle* runs just great on

Sun Microsystem's *Solaris* operating system. *Solaris* is the current operating platform for Ameritech's current TACACS+ servers, thus, any RDBMS solution that gets implemented must fully support this environment.

That choice made, the only *Oracle* solution to which I had access was in the *Jennings Computer Lab*, running on *Microsoft Windows NT*. Needless to say, after this experience, I am not a firm believer in *Oracle* version 7.x for NT. To be fair, many of the problems that I encountered were not the fault of *Oracle* or *Microsoft*. The *Jennings* lab as of late 1998 is understaffed and under funded. Thus, the physical hardware backing up the *Oracle* server wasn't exactly the greatest. It wasn't working when I wanted to start on my implementation, so I had to find an administrator, and have him reboot the server a number of times until it allowed me to login.

Once I was connected to the server, I was receiving transient *"Shared Memory Allocation"* errors when importing all of my data into the database. Unloading and reloading the data several times seemed to clear up these problems.

The next set of problems that I encountered revolved around *Oracle's* built-in functions. In particular, I was interested in several functions that dealt with the *DATE* datatype. Unfortunately, I was never able to decipher the documentation to the point that I actually fully understood how to utilise these functions. Furhtermore, as I attempted to explore these functions on my own (using the given examples asa guide), I was able to confuse the RDBMS, to the point that no single query (no matter how simple) would run. Basically, every command generated some form of internal error in the database. It was at this point that the *SQL*Plus* front-end crashed, prompting me to re-login. Upon doing so, everything appeared to be well, but I was still unable to use any of the built-in functions.

I ended up crashing *SQL*Plus* a few more times before I finally decided to give up on these functions.

Nevertheless, all of the SQL that I used, as well as the output that it generated, will be presented in a series of *appendicies* at the end of this report.

Conclusion

In order to conclude the discussion of this database design, some thought needs to be given to its possible commercial-grade implementation. In order to implement this database, Ameritech would have to carefully consider all of the costs involved, and weigh them against the perceived value of the database implementation. The costs in moving to the database model involve programmer time, as well as the purchase of additional hardware and software. Once this solution has been implemented, it could add additional day-to-day costs, in the form of an increased need to keep well-trained staff in order to support this solution.

Furthermore, much of the increased functionality that the RDBMS offers isn't very concrete, which makes this solution a "hard sell". For example, the possibility of increased reliability and scalability afforded by the *Authentication* and *Authorization* relations won't be visible until the current solution *breaks*. Thus, the data partitioning features of the RDBMS (the fact that it can allow the data from multiple customers to be stored on one machine) must be emphasized instead. But all told, less motivation exists for using the RDBMS to store the *authentication* and *authorization*.

The RDBMS offeres much more tangible functionality when it comes to the *accounting* information, however. There are many problems and unimplemented features with the data reporting methodolgies that are currently in use. With the RDBMS, however, many of these problems can be easily and efficiently solved.

Thus, it is my final recommendation that the *accounting* aspects of this design be given a serious look. It would be a simpler task to start using a database for this data, and if the RDBMS proved itself, then it would make more sense to apply it to the other two areas as well.

Appendix A: Sample Data

Unfortunately, I was not able to use *"live"* customer data from Ameritech in the database that I created for this project. Many of the attributes in the data set contain highly sensitive values (such as passwords, userid's, and phone numbers, to name a few), and thus, it was clearly impossible for me to include this data in a silly report. So, I created a fairly effective mechanism by which my own, *faux* data could be created. Basically, I wrote an approximately 700-line perl script that produces all of the SQL necessary in order to insert an arbitrary number of *Authentication*, *Authorization*, and *Accounting* records into my database. I attempted to make my *faux* data adhere to the actual data as much as possible, and although some of the intrinsic dependcies don't make sense, on a superficial level the data looks great.

It is too much to attempt to include all of the data that I generated in this document. The *Accounting* information alone amounted to almost a megabyte of text. Thus, I will simply include the perl script, and make the actual data available externally.

```
#!/usr/local/bin/perl
##############################################################################
# Andy Reitz                                               reitz@ces.cwru.edu
# ECES 433 Final Project                                     December 8, 1998
##############################################################################
# The purpose of this 'generate_data' script is to produce a very realistic
# set of sample data for my Database Design Project. In particular, this
# program will output three ".sql" files, each one containing a number of
# records expressed in SQL92 format. Whenever possible, data has been
# generated that 'makes sense', or at least, approximates real-world data.
#
# As an aside, it was harder to develop this program that I initially thought
# that it would be -- the total programming time was about nine (9) hours.
# But, this was one of the more 'fun' aspects of this project, so it was
# well worth the effort.
##############################################################################
# Internal Script-configuration elements.

use strict;                      # Keeps me honest.

##############################################################################
# Global Configuration Section.
my ($NUM_DATA) = 500;                  # Number of records to generate.

# Output filenames.
my ($authen_fn) = "authentication_data.sql";
my ($author_fn) = "authorization_data.sql";
my ($acct_fn) = "accounting_data.sql";

###############################################################################
# Global Variable Declarations.
my ($counter);
my ($routing, $route);
my ($userid);
my ($passwd);
my ($username);
my (@userids, @usernames);
my ($inacl, $outacl, $timeout, $idletime);

###############################################################################
# Begin Main Program.

# We’ll use this seed throughout the program.
srand (time());

#
# The two ’if’ statements that follow check to see if a file exists, and if it
# doesn’t, they will attempt to open them for writing.
#
if (-e $authen_fn)
       {
       print "Error: the Authentication output file, \"$authen_fn\", already
exists!\n";
       exit (3);
       }
else
       {
       if (!open (AUTHEN, "> $authen_fn"))
              {
              print "Error: Couldn’t open the Authentication output file,
\"$authen_fn\", for writing. Reason: $!\n";
              exit (4);
              }
       }

if (-e $author_fn)
       {
       print "Error: the Authorization output file, \"$author_fn\",  already
exists!\n";
       exit (5);
       }
else
       {
       if (!open (AUTHOR, "> $author_fn"))
              {
              print "Error: Couldn’t open the Authorization output file,
\"$author_fn\", for writing. Reason: $!\n";
              exit (6);
              }
       }

#
# This juicy little ’for’ loop generates both the ’Authentication’ and
# ’Authorization’ datafiles. Basically, for every tuple that is generated in
# the ’Authentication’ file, an co-inciding tuple (using the same userid)
# is generated for the ’Authorization’ datafile. All of the userid’s and
# username’s are stored into arrays, for later use.
#
for ($counter = 0; $counter < $NUM_DATA; $counter++)
```

```
	{
	print AUTHEN "INSERT INTO Authentication VALUES (";
	$userid = create_userid();
	$userid .= $counter;
	push @userids, $userid;
	$username = create_name();
	push @usernames, $username;

	print AUTHEN "’$userid’, $username, ";

	$passwd = create_passwd();

	print AUTHEN "$passwd, ’template’, $passwd, $passwd, ";
	print AUTHEN create_auth_type(), ", ";
	print AUTHEN "$passwd)\;\n";

	print AUTHOR "INSERT INTO Authorization VALUES (";
	print AUTHOR "’$userid’, ";
	($inacl, $outacl, $timeout, $idletime) = create_authz_ints();
	print AUTHOR "$inacl, $outacl, $timeout, $idletime, ";

	print AUTHOR create_addr(0), ", ";

	($routing, $route) = create_routing();
	print AUTHOR "$routing, $route);\n";
	}

close (AUTHEN);
close (AUTHOR);

#
# Spit-out the accounting information.
#
create_accounting (\@userids, \@usernames, $acct_fn);

# End of Main Program.
##############################################################################

##############################################################################
# Begin Subroutines.

#
# This subroutine returns a pretty funky userid. It’s basically 6 random
# lower-case letters thrown together. Generating meaningful userid’s (based
# off of the name), would have been much more difficult. We’ll append a unique
# number to each, in order to keep them unique. So, in essence, these letters
# are just for effect.
#
sub create_userid {

	my (@char_ary) = (’a’ .. ’z’);
	my ($counter);
	my ($ret_userid);

	for ($counter = 0; $counter < 5; $counter++)
		{
		$ret_userid .= $char_ary[rand($#char_ary) + 1];
		}

	return ($ret_userid);

} # End create_userid().
```

```
#
# This routine returns a an 8-character-long password, composed of pretty
# random characters 60% of the time. The other 40% of the time, it just
# returns the default "tempass1" password.
#
sub create_passwd {

	my (@char_ary) = ('A' .. 'Z', 'a' .. 'z', '1' .. '9', '!', '_');
	my ($counter);
	my ($ret_passwd) = "'cleartext ";

	if (rand > 0.3)
		{
		for ($counter = 0; $counter < 8; $counter++)
			{
			$ret_passwd .= $char_ary[rand($#char_ary) + 1];
			}
		$ret_passwd .= '\'';
		}
	else
		{
		$ret_passwd .= "tempass1'";
		}

	return ($ret_passwd);

} # End create_passwd().

#
# This subroutine creates a random person-name. It requires the use of two
# input files, "first.dat" (containing a list of first names), and
# "last.dat" (containing a list of last names). It will pick a random word
# from each file, as well as a random middle initial (only 20% of the time),
# and return all of the data in the "Last_Name, First_Name Middle_Initial"
# format.
#
sub create_name {

	my (@fn, @ln);
	my (@initials) = ('A' .. 'Z');
	my ($mi);
	my ($ret_ln, $ret_fn);

	if (!open (FIRST, "first.dat"))
		{
		print "Error: Couldn't open input file \"first.dat\". Reason
$!\n";
		exit (1);
		}

	if (!open (LAST, "last.dat"))
		{
		print "Error: Couldn't open input file \"last.dat\". Reason $!\n";
		exit (2);
		}

	@fn = <FIRST>;
	@ln = <LAST>;

	if (rand > 0.6)
		{
		$mi = $initials[rand ($#initials) + 1];
		}
```

```
        $ret_ln = $ln[rand ($#ln) + 1];
        chop ($ret_ln);

        $ret_fn = $fn[rand ($#fn) + 1];
        chop ($ret_fn);

        return ("'$ret_ln, $ret_fn $mi'");

} # End create_name().

#
# This subroutine returns the string "admin" 10% of the time.
#
sub create_auth_type {

        if (rand > 0.9)
                {
                return ("'admin'");
                }
        else
                {
                return ("NULL");
                }

} # End create_auth_type().

#
# This subroutine creates a random IP address, 20% of the time.
#
sub create_addr {

        my ($dorand) = shift (@_);

        my ($oct1, $oct2, $oct3, $oct4);

        if ($dorand || (rand > 0.8))
                {
                $oct1 = int (rand (255));
                $oct2 = int (rand (255));
                $oct3 = int (rand (255));
                $oct4 = int (rand (255));
                }

        if (defined ($oct1))
                {
                return ("'$oct1.$oct2.$oct3.$oct4'");
                }
        else
                {
                return ("NULL");
                }

} # End create_addr().

#
# This subroutine returns two values -- the 'routing' attribute, followed
# by the 'route' attribute. If we decide that 'routing' should be FALSE (as
# it will be 60% of the time), then we'll just return 'NULL' for the 'route'
# attribute. Otherwise, we'll actually generate a random route statement.
#
sub create_routing {
        my ($o1, $o2, $o3, $o4);

        if (rand > 0.4)
                {
                return ("'FALSE'", "NULL");
                }
        else
                {
                #
                # Creating a valid route is very difficult, and since the
                # point of this is just to get some data flowin', I'm not
                # going to try very hard. The TACACS+ spec says that it
                # expects 'route' attributes in the form of:
                #
                #      <dst_address> <mask> <routing_addr>
                #
                # So, in order to make things easier, I'm going to assume that
                # everything is 'class C' masked.
                #
                $o1 = int (rand (255));
                $o2 = int (rand (255));
                $o3 = int (rand (255));
                $o4 = int (rand (255));

                return ("'TRUE'", "'$o1.$o2.$o3.0 255.255.255.0
$o1.$o2.$o3.$o4'");
                }

} # End create_routing().

#
# This subroutine creates the four integers necessary for an Authorization
# record.
#
sub create_authz_ints {

        my ($inacl, $outacl, $timeout, $idletime);

        #
        # Choose the access list values.
        #
        if (rand > 0.6)
                {
                $inacl = int (rand (456));
                $outacl = int (rand (456));
                }
        else
                {
                $inacl = "NULL";
                $outacl = "NULL";
                }

        #
        # Choose the timeout parameter. We'll allow it to range from zero to
        # 1200 minutes (20 hours).
        #
        if (rand > 0.8)
                {
                $timeout = 60 * int (rand (1200));
                }
        else
                {
                $timeout = "NULL";
```

```
		}

	#
	# Now, choose the idle timeout.
	#
	if (rand > 0.7)
		{
		$idletime = 60 * int (rand (10));
		}
	else
		{
		$idletime = "NULL";
		}

	return ($inacl, $outacl, $timeout, $idletime);

} # End create_authz_ints().

#
# Ohboy, what fun here. This function attempts to generate a whole mess of
# 'Raw_Accounting' records. Whereas the previous two relations had a one-to-one
# relationship, this doesn't hold for Accounting (think about it -- people like
# to login more than once!). So, this function handles all of that, in the
manner
# that I best saw fit. Note that the dependencies between the Accounting
records
# and the other two relations aren't very accurate -- with the exception of the
# userids/names, everything else is randomly independent. Realistically, the
only
# way to have gotten better data would have been to setup a sample TACACS+
server,
# and generate it that way.
#
sub create_accounting {

	#
	# Gather the parameters from the parent. Note, I'm using variable
	# referencing here in order to pass two arrays. Wierd.
	#
	my ($userids) = shift (@_);
	my ($usernames) = shift (@_);
	my ($acct_fn) = shift (@_);

	my ($userid);       # The current userid.
	my ($user_cnt) = 0; # Counter; Steps through userid array.
	my ($host);         # The current NAS-hostname.
	my (@task_id) = (0, 0);    # Array of task_id's, one-per-NAS.
	my ($type);         # START, STOP, or BOTH?

	# Do that funky file stuff again.
	if (-e $acct_fn)
		{
		print "Error: the Accounting output file, \"$acct_fn\", already
exists!\n";
		exit (7);
		}
	else
		{
		if (!open (ACCT, "> $acct_fn"))
			{
			print "Error: Couldn't open the Accounting output file,
\"$acct_fn\", for writing. Reason: $!\n";
			exit (8);
```

```
                        }
                }

        #
        # We start off by generating the 'good' data -- i.e. valid START/STOP
        # pairs.
        #
        # For every userid, we have to generate a number of unique (?)
        # accounting elements:
        #       NAS-hostname - We'll assume that each person sticks to one NAS.
        #       remote_phone_number - Assume that they call from one location.
        #
        foreach $userid (@$userids)
                {
                if (rand > 0.5)
                        {
                        $host = "IL_as2516";
                        $task_id[0] = emit_acct_rec (@$userids[$user_cnt],
@$usernames[$user_cnt], 'BOTH', $task_id[0], $host);
                        }
                else
                        {
                        $host = "OH_as5200";
                        $task_id[1] = emit_acct_rec (@$userids[$user_cnt],
@$usernames[$user_cnt], 'BOTH', $task_id[1], $host);
                        }

                $user_cnt++;
                } # foreach

        # Now, generate some bogus START/STOP records.
        for ($user_cnt = 0; $user_cnt < ($NUM_DATA * 0.1); $user_cnt++)
                {
                # What type shall we make?
                if (rand > 0.5)
                        {
                        $type = 'START';
                        }
                else
                        {
                        $type = 'STOP';
                        }

                # Make it so!
                if (rand > 0.5)
                        {
                        $host = "IL_as2516";
                        $task_id[0] = emit_acct_rec (@$userids[$user_cnt],
@$usernames[$user_cnt], $type, $task_id[0], $host);
                        }
                else
                        {
                        $host = "OH_as5200";
                        $task_id[1] = emit_acct_rec (@$userids[$user_cnt],
@$usernames[$user_cnt], $type, $task_id[1], $host);
                        }
                } # for

        close (ACCT);

} # End create_accounting().

#
```

```
# This function attmepts to emit a variable number of START/STOP records,
# based upon it’s given parameters. It’s all quite a hack, really (but
# then again, so is /perl/, if you think about it).
#
sub emit_acct_rec {

       # Start off by getting all of our parameters.
       my ($cur_userid) = shift (@_);
       my ($cur_username) = shift (@_);
       my ($passed_type) = shift (@_);
       my ($task_id) = shift (@_);
       my ($cur_host) = shift (@_);

       my ($cur_ph);             # Phone Number.
       my ($cur_port);                  # NAS-port.
       my ($cur_type);                  # START/STOP/etc.
       my ($start_time, $stop_time);    # Starting and Stopping times.
       my ($num_run);                   # Counter; number of outputs to do.
       my ($run_count) = 0;             # Counter; current output being made.

       # Bulk variables for STOP record.
       my ($pre_bytes_in, $pre_bytes_out, $pre_paks_in, $pre_paks_out);
       my ($bytes_in, $bytes_out, $paks_in, $paks_out);
       my ($pre_session_time, $elapsed_time);

       #
       # This whole notion of START/STOP/BOTH is pretty hackish, but it
       # works (and you can’t argue with that).
       #
       if ($passed_type eq "BOTH")
              {
              # Okay, we’re doing the ’real thing’. Generate a random
              # number of START/STOP pairs.
              $cur_type = ’START’;
              $num_run = int (rand (10));
              if ($num_run == 0)
                     {
                     $num_run = 1;
                     }
              }
       else
              {
              # Faux-record, only do it once.
              $cur_type = $passed_type;
              $num_run = 1;
              }

       # Get our guy’s phone number.
       $cur_ph = gen_phone_number($cur_host);

       #
       # This is a pretty fun loop. It generates all of the data that is unique
       # per START/STOP pair, and then emits the pair. It of course will only
       # emit one record if it wasn’t given ’both’ to start with.
       #
       for ($run_count = 0; $run_count < $num_run; $run_count++)
              {
              #
              # Now, we have to generate START/STOP pairs. Unique
              # to each pair will be:
              #     The start/stop times (duh)
              #     NAS-port - we’ll assume it differs
              #     task_id - counter; unique to a NAS.
```

```
             #
             ($start_time, $stop_time) = gen_dates();

             $cur_port = gen_port ($cur_host);

             # Emit a beginning -- could be all if we're 'START'.
             print ACCT "INSERT INTO Raw_Accounting VALUES (";
             print ACCT "'$start_time', '$cur_host', '$cur_userid',
'$cur_port', $cur_ph, '$cur_type', $cur_username, ";
             print ACCT "$task_id, 'UTC', 'PPP'";

             if ($passed_type eq "BOTH")
                    {
                    $cur_type = 'STOP';
                    # finish off START, start again.
                    print ACCT ", NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL)\;\n";
                    print ACCT "INSERT INTO Raw_Accounting VALUES (";
                    print ACCT "'$stop_time', '$cur_host', '$cur_userid',
'$cur_port', $cur_ph, '$cur_type', $cur_username, ";
                    print ACCT "$task_id, 'UTC', 'PPP'";
                    }

             #
             # Generate the 'STOP' portion of the record, if necessary.
             #
             if ($cur_type eq "STOP")
                    {
                    # It's true, we'll always get an addr.
                    print ACCT ", 'IP', ", create_addr (1), ", ";

                    # Disconnection will always be same.
                    print ACCT "1, 1045, ";

                    # I really hate all of the stupid counters.
                    $pre_bytes_in = int (rand (200));
                    $pre_bytes_out = int (rand (200));
                    $pre_paks_in = int (rand (12));
                    $pre_paks_out = int (rand (12));

                    $bytes_in = int (rand (1000000000));
                    $bytes_out = int (rand (1000000000));
                    $paks_in = int ($bytes_in / 1500) + int (rand (230));
                    $paks_out = int ($bytes_out / 1500) + int (rand (230));

                    $pre_session_time = int (rand (20));
                    $elapsed_time = int (rand (72000));

                    print ACCT "$pre_bytes_in, $pre_bytes_out, $pre_paks_in,
$pre_paks_out, ";
                    print ACCT "$bytes_in, $bytes_out, $paks_in, $paks_out, ";
                    print ACCT "$pre_session_time, $elapsed_time, ";
                    print ACCT "56000)\;\n";
                    $cur_type = 'START';
                    }
             else
                    {
                    # Wrap-up the dangling START.
                    print ACCT ", NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL)\;\n";
                    }

             # We'll need this to be different next time 'round.
```

```
                $task_id++;
                } # for

        # Send our updated $task_id back to momma.
        return ($task_id);

} # End emit_acct_rec().

#
# This subroutine computes that starting time and stopping time for a call,
# and returns said values. All records will start on the same day, but
# at different times. From there, they can end at maximum 20 hours later.
#
# I took a quick spin of the Oracle 7 On-Line documentation, and it seems that
# the format for their 'date' datatype is something like this:
#
#               DD-MON-YYYY 12:00:00a.m.
#
# Now, I don't /exactly/ support that format. I think that the notion of
# 'a.m.'/'p.m' is only for humans, not computers. So, I'll see if Oracle will
# take time in the 24-hour format. If it doesn't, I'll be back to edit this
# code.
#
# And I'd just like to say that this bit of code doesn't represent the
# amount of time that it took to craft this function. I coded it about
# two other ways first, that just didn't pan out. This is nice and tight,
# though, and I think I'll be able to use this again...
#
sub gen_dates {

        my ($now, $later);
        my (@rn, @rl);
        my ($start, $stop);
        my (@months) = ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec');

        #
        # Now is really later (by no more than 10 hours). Sometimes, this code
        # produces times that are actually 'ahead' of the current time.
        #
        # I have no idea why this happens.
        #
        $now = time();
        $now -= int (rand (36000));

        # 72000 = 60sec * 60min * 20hr
        $later = $now + int (rand (72000));

        #
        # gmtime() converts-out the time into a whole LIST of values. In
        # particular, these are of importance:
        #     0 - seconds
        #     1 - minutes
        #     2 - hour
        #     3 - day of the month
        #     4 - month number (starts @ zero)
        #     5 - year number (# years from 1900)
        #
        @rn = gmtime ($now);

        # Pad-out anything that needs it (grr...)
        if ($rn[3] < 10)
                {
```

```
			$rn[3] = "0$rn[3]";
			}

		if ($rn[2] < 10)
			{
			$rn[2] = "0$rn[2]";
			}

		if ($rn[1] < 10)
			{
			$rn[1] = "0$rn[1]";
			}

		if ($rn[0] < 10)
			{
			$rn[0] = "0$rn[0]";
			}

		$rn[5] += 1900;
		$start = "$rn[3]-$months[$rn[4]]-$rn[5] $rn[2]:$rn[1]:$rn[0]";

		@rl = gmtime ($later);

		if ($rl[3] < 10)
			{
			$rl[3] = "0$rl[3]";
			}

		if ($rl[2] < 10)
			{
			$rl[2] = "0$rl[2]";
			}

		if ($rl[1] < 10)
			{
			$rl[1] = "0$rl[1]";
			}

		if ($rl[0] < 10)
			{
			$rl[0] = "0$rl[0]";
			}

		$rl[5] += 1900;
		$stop = "$rl[3]-$months[$rl[4]]-$rl[5] $rl[2]:$rl[1]:$rl[0]";

		return ($start, $stop);

} # End gen_dates().

#
# Generates a 10 digit phone number, area-code first. I think that it’s pretty
# spiffy that the area codes actually ’make sense’.
#
sub gen_phone_number {

		my ($hostname) = shift (@_);
		my (@area_codes);
		my ($ret_num);
		my ($counter);

		if ($hostname =~ /OH/)
			{
```

```
		@area_codes = (440, 216, 330);
		}
	else
		{
		@area_codes = (630, 708, 312, 847, 713);
		}

	$ret_num = '\'';
	$ret_num .= $area_codes[int (rand ($#area_codes))];

	for ($counter = 0; $counter < 7; $counter++)
		{
		$ret_num .= int (rand (10));
		}

	$ret_num .= '\'';

	return ($ret_num);

} # End gen_phone_number().

#
# This subroutine returns a port name, based upon the NAS-hostname it is given.
# It appends a random number to each port, the size of which is based upon
# my notion of each NAS's port capacity.
#
sub gen_port {

	my ($hostname) = shift (@_);
	my ($ret_port);

	if ($hostname =~ /2516/)
		{
		# These things max out at 32 ports.
		$ret_port = "2516_Async_";
		$ret_port .= int (rand (32));
		}
	else
		{
		# These things max out at 256 ports.
		$ret_port = "5200_Async_";
		$ret_port .= int (rand (256));
		}

	return ($ret_port);
} # End gen_port().
```

Appendix B: "generate_calls_completed.sql"

```
/****************************************************************************/
/* ECES 433, Final Design Project                                           */
/*     'generate_calls_completed.sql' - Implements the PL/SQL functionality */
/*            necessary in order to convert stateless 'Raw_Accounting'      */
/*            records into the stateful 'Call' type records.                */
/* by Andy Reitz (reitz@ces.cwru.edu)                                       */
/* Date: 12/10/98                                                           */
/****************************************************************************/

/* To start off, the output table must be created. */
CREATE TABLE Calls_Completed (
	callid INTEGER,
	userid VARCHAR2(11),
	NAS_hostname VARCHAR2(11),
       NAS_port VARCHAR2(16),
       start_time DATE,
       stop_time DATE,
       duration INTEGER,
       tot_bytes_in INTEGER,
       tot_bytes_out INTEGER,
       tot_paks_in INTEGER,
       tot_paks_out INTEGER,
       PRIMARY KEY (callid));

DECLARE
       /* This cursor points to all of the START-rows. */
       CURSOR start_cur IS
              SELECT  *
              FROM   Raw_Accounting
              WHERE  type = 'START';

       /* This cursor points to all of the STOP-rows. */
       CURSOR stop_cur IS
              SELECT *
              FROM   Raw_Accounting
              WHERE  type = 'STOP';

       callid INTEGER := 0;       /* The current call found.    */
       duration INTEGER;   /* The length of said call.   */
       tot_bytes_in INTEGER;      /* Aggregated bytes input.    */
       tot_bytes_out INTEGER;     /* Aggregated bytes output.   */
       tot_paks_in INTEGER;       /* Aggregated packets input.  */
       tot_paks_out INTEGER;      /* Aggregated packets output. */

BEGIN
       /*
        * This function is implemented as two nested loops. The outer loop
        * steps through every START record. For each such record, we look
        * through all of the STOP records for the one that has the same
        * NAS_hostname and task_id. Once found, this data is inserted as a
        * new record in the 'Calls_Completed' table, and the original tuples
        * are deleted from 'Raw_Accounting'.
        */
       FOR start_rec IN start_cur LOOP
              FOR stop_rec IN stop_cur LOOP
                     IF (start_rec.task_id = stop_rec.task_id) AND
(start_rec.NAS_hostname = stop_rec.NAS_hostname) THEN
                            /* Compute duration */
                            duration := stop_rec.log_date - start_rec.log_date;

                            /* Compute totals */
                            tot_bytes_in := stop_rec.pre_bytes_in +
stop_rec.bytes_in;
                            tot_bytes_out := stop_rec.pre_bytes_out +
stop_rec.bytes_out;
                            tot_paks_in := stop_rec.pre_paks_in +
stop_rec.paks_in;
                            tot_paks_out := stop_rec.pre_paks_out +
stop_rec.paks_out;

                            /* Insert this call. */
                            INSERT INTO Calls_Completed VALUES (callid,
                                   start_rec.userid, start_rec.NAS_hostname,
                                   start_rec.NAS_port, start_rec.log_date,
                                   stop_rec.log_date, duration, tot_bytes_in,
                                   tot_bytes_out, tot_paks_in, tot_paks_out);
```

```
                          /* Increment callid number */
                          callid := callid + 1;

                          /* Delete the stop_rec */
                          DELETE
                          FROM   Raw_Accounting
                          WHERE  task_id = stop_rec.task_id
                                 AND NAS_hostname = stop_rec.NAS_hostname;

                          /* Delete the start_rec */
                          DELETE
                          FROM   Raw_Accounting
                          WHERE  task_id = start_rec.task_id
                                 AND NAS_hostname = start_rec.NAS_hostname;

                   END IF;
            END LOOP;
      END LOOP;
END;
```

Appendix C: "db_init.sql"

```
/****************************************************************************/
/* ECES 433, Final Design Project                                           */
/*     'db_init.sql' - Configures the Oracle 7 environment and creates the  */
/*            the three main relations.                                     */
/* by Andy Reitz (reitz@ces.cwru.edu)                                       */
/* Date: 12/10/98                                                           */
/****************************************************************************/

/* Setup the environment. */
set linesize 500
set pagesize 1000
set wrap off

/* This makes the dates in the 'Accounting' relation work better. */
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';

/* Clean-up any existing tables. */
drop table calls_completed;
drop table raw_accounting;
drop table authorization;
drop table authentication;


/* Create the 'Authentication' table. */
CREATE TABLE Authentication (
      userid VARCHAR2(11),
      name VARCHAR2(40),
      login VARCHAR2(18),
      member VARCHAR2(10),
      chap VARCHAR2(18),
      pap VARCHAR2(18),
      type VARCHAR2(5),
      global VARCHAR2(18),
      PRIMARY KEY (userid));


/* Create the 'Authorization' table. */
CREATE TABLE Authorization (
      userid VARCHAR2(11),
      inacl INTEGER,
      outacl INTEGER,
       timeout INTEGER,
       idletime INTEGER,
       addr VARCHAR2(15),
       routing VARCHAR2(5),
       route VARCHAR2(49),
       FOREIGN KEY (userid) REFERENCES Authentication ON DELETE CASCADE);


/* Create the 'Accounting' table. */
CREATE TABLE Raw_Accounting (
       log_date DATE,
       NAS_hostname VARCHAR2(11),
       userid VARCHAR2(11),
       NAS_port VARCHAR2(16),
       remote_phone_number CHAR(10),
       type VARCHAR2(6),
       NasUserName VARCHAR2(40),
       task_id INTEGER,
       timezone VARCHAR2(3),
       service VARCHAR2(3),
       protocol VARCHAR2(3),
       addr VARCHAR2(15),
       disc_cause INTEGER,
       disc_cause_ext INTEGER,
       pre_bytes_in INTEGER,
       pre_bytes_out INTEGER,
       pre_paks_in INTEGER,
       pre_paks_out INTEGER,
       bytes_in INTEGER,
       bytes_out INTEGER,
       paks_in INTEGER,
       paks_out INTEGER,
       pre_session_time INTEGER,
       elapsed_time INTEGER,
       data_rate INTEGER);
```

Appendix D: "authentication_queries.sql"

```
/**************************************************************************/
/* ECES 433, Final Design Project                                         */
/*     'authentication_queries.sql' - Performes the given 'Authentication'  */
/*            Queries.                                                    */
/* by Andy Reitz (reitz@ces.cwru.edu)                                     */
/* Date: 12/10/98                                                         */
/**************************************************************************/

/*
 * Authentication Query 1 (Find the record for particular userid)
 */
SELECT  *
FROM    Authentication
WHERE   userid = 'jekmf0';

/*
 * Authentication Query 2 (Find all of the users that have an unitialized
 * password)
 */
SELECT  A.userid, A.name
FROM    Authentication A
WHERE   login = 'cleartext tempass1'
        OR pap = 'cleartext tempass1'
        OR chap = 'cleartext tempass1'
        OR global = 'cleartext tempass1';
```

```
/*
 * Authentication Query 3 (Find any administrators that have an unitialized
 * password)
 */
SELECT  A.userid, A.name
FROM    Authentication A
WHERE   (A.type = ’admin’ AND A.login = ’cleartext tempass1’) OR
        (A.type = ’admin’ AND A.pap = ’cleartext tempass1’) OR
        (A.type = ’admin’ AND A.chap = ’cleartext tempass1’) OR
        (A.type = ’admin’ AND A.global = ’cleartext tempass1’);

/*
 * Authentication Query 4 (Determine the number of actual users, as well as the
number of
 * active users)
 */
SELECT  COUNT (*) AS num_users
FROM    Authentication;

SELECT  COUNT (A2.userid) AS active_users
FROM    Authentication A2
WHERE   A2.login <> ’cleartext tempass1’
        OR A2.pap <> ’cleartext tempass1’
        OR A2.chap <> ’cleartext tempass1’
        OR A2.global <> ’cleartext tempass1’;
```

Appendix E: “authentication_queries.out”

```
DOC> * Authentication Query 1 (Find the record for particular userid)
DOC> */
```

***** NOTE: The original output to this query was lost. I believe this to be an accurate reconstruction. *****

```
USERID      NAME                                     LOGIN             MEMBER
CHAP             PAP                TYPE  GLOBAL
----------- ---------------------------------------- ----------------- -------
--- ----------------- ----------------- ----- -----------------
jekmf0      Tiddleflip, Greedo                       cleartext fG4KbIhk
template   cleartext fG4KbIhk cleartext fG4KbIhk NULL  cleartext fG4KbIhk

DOC> * Authentication Query 2 (Find all of the users that have an unitialized
password)
DOC> */

USERID      NAME
----------- ----------------------------------------
wpznp3      Zebo,Roto-Rooter N
xswbk5      Funtz, Toadstool D
hslxe7      Vueigez, Mussolini
jbonl8      Lumpwump, Santos
jcrxm11     Lewis, Bartley
plitn12     Qureshi, Professor
oecpo15     Herder, Fips
nxjct16     Preen, Long
fovyb18     Lester, Chief
qfpcc21     Bator, Fyvush
gdmdn27     Merat, Duke
emylp30     Blatch, Dominic H
rlukh31     McGooter, Ahura
zxbbd32     Keenan, Chief
njnuj39     Roxwox, Binky
```

```
clntv41     Toothpaste, Crowly
ozbdy46     Horn, Sascha X
dwknj48     Toothpaste, Duke
cbfio55     Keenan, Stephan
dedtu58     Funtz, Crunch
ljmdj62     Pip, Frink
svjyx65     Brown, Marc E
rmdbs68     Zabubadoofski, Professor
fsxcw72     Oro, Debbie O
vpjcj80     Finn, Jean-Louis I
hqjmv81     Rassoodock, Irene U
urpkb88     Ringworm, Chewie
gwbtv92     Merat, Eblis
dqmui94     Hekkelman, Jean-Louis U
crozz96     Hrumpf, Flaudvie K
quhrt99     Goesh, Boutros-Boutros
izbvw101    Twinkie, Cousin S
juzrv104    McGooter, Buzz G
dfzjs106    Papachristou, Chunk
omsbp111    Mingus, Polly I
itmok115    Shaughnessy, Chups D
jjszh116    Bologna, Elmo B
wjojc117    Wallop, Hap V
ojrtl120    Johnson, Roto-Rooter
xohfz124    Toothpaste, Shirley
ffdwq125    Brown, Lumps D
puqii132    Styrofoam, Ralph
oszbo136    Dilmont, Rodney
llifj138    Esch, Luan
dtens139    Gapeev, Nerf W
lvyph141    Mazda, Gilligan N
ernun147    Trier, Butch
byccq148    Shucker, Liz
mkwuv150    Harley, Chunk
nsqyk152    Ringworm, Polly
ucupj154    Puldup, Maximilian
izuyu163    Puldup, Gerp U
cbhpr164    McGrooter, Onnie U
kovyb166    Windex, Cletus
cjggr167    Tabukalloli, Gilligan O
umxbd169    Hermfik, Roto-Rooter
torbc170    Harasmatari, Shampoo X
jpqel173    Tabukalloli, Pam
qmnik182    Brown, Bobo
kycox188    Dent, Liz
yqubs189    Brown, Jean-Louis
rmfri195    Buddha, Futon
fmbgm196    Tunklebit, Mussolini M
cqcbp203    Roxwox, Eblis G
buuyv209    Reed, Gorbin
miooc214    Torvalds, Sancho
tdbqs216    Dannowski, Shampoo
ecvjt217    Wallop, Elmo
fpvjp220    Klink, Onnie D
hxhcs222    Nihlen, Ulek
vfrwb226    Tanenbaum, Fyvush
uyvuo229    Pip, Francis
remld234    Uber, Hap G
gotee254    Prune, Fivel
mklhi255    Blatch, Orange
cdjqv256    Yummy, Polly
xddbx259    Beanie, Jean-Louis
zkckk263    Cuervo, Gilligan
```

oecks264	Gouda, Bobo
qugcf265	Crumchuck, Illapotin
xktqy270	Ozsoyoglu, Boutros-Boutros R
qebuj272	Torvalds, Gilligan
qzgzl275	Lopdop, Liz
sbgtk278	Pip, Professor G
rjtph283	Shucker, Luan D
xhnok288	Smee, Morton
dhoim289	Torvalds, El P
jlmfn292	Styrofoam, Toadstool
umfij294	Bologna, Frisky
vdebt298	Strunker, Nogo
hejgs300	Vader, Brent
yurlj304	Lipster, Tex D
tpeuk308	Leech, Linus
vysro309	Fumbucket, Darth
wpyqw310	Quiggle, Marc
tvrgt312	Dipdip, Al
bzwfg314	Qureshi, Frink L
zgoqr316	Pulsifer, Gorbin
xvytx321	Palmer, Porky
ugezy328	Dent, Uwe R
wsdfl332	Cuervo, Polly R
kmepv333	Shucker, Futon
sdhju336	Hon, KentHavnoovy
ndrup340	Budupadupa, Fyvush E
dsjwo341	Crumchuck, Thelonious
qrkuv342	Shaughnessy, Sara R
puzjv344	Preen, Natarajan
btkvc347	Dent, Minga
oxrks348	Styrofoam, Meep
zfmiv350	Andreesen, Nogo E
huuxi352	Hrumpf, Seymor J
uxgen354	Gompers, Mugwhump F
tynng356	Strunker, Nerf
wkcfw357	Palmer, Marc
ilcqj358	Hrumpf, Hewlett B
ipugv363	Ocasio, Alexei
wwide366	Reitz, Fivel
qiucm368	Dave, Cousin
jrnhk369	Palmer, Maximilian
nzukm371	Trier, Yitner T
hpxjb377	Lopdop, Gilligan
ztxzb382	Uber, Luan D
kvkzb383	Yaxmutt, Andy
yoqol384	Chiller, Gorbin Q
mjqgl387	Gouda, Brent
sfpym389	Finkel, Clunky
bgqgc392	Young, Fritz F
jjred393	Vueigez, Krusty
lxogw398	McGrooter, Tex
wqrhf402	Harley, Bartley
ngifz404	Sned, Mitzencrom
xdcmu410	Bologna, Arafat
owbrb417	Finkel, Futon
dcshv418	Tabukalloli, Goomba S
lxzny426	Ferndip, Natarajan T
rgxgd427	Storrs, Scooter P
pxhoj430	Nihlen, Roto-Rooter
urhgj431	Hekkelman, Shirley
ixvqy432	Smee, Luke U
yzvgf433	Tanenbaum, Maximilian
djcwl443	Zipper, Professor R

```
gfnsp445    Hootenanny, Santos
xsiux448    Reed, Pops
mrxoh453    Pulsifer, Lamar
tmjiq460    Hambone, Luan X
oihhr463    Leech, Sara S
wcqdd473    Cheesenose, Linus
rugrb474    Ghali, Chief
whzed481    Lewis, Buck
ejkqh486    Yada, Gorbin U
dkbss488    Ozsoyoglu, Fyvush O
jwrjf491    Beanie, Lamar
ursyr492    Packard, Eblis J
gsbsk499    Shaughnessy, Mugwhump H

154 rows selected.

DOC> * Authentication Query 3 (Find any administrators that have an unitialized
password)
DOC> */

USERID      NAME
----------- ----------------------------------------
oecpo15     Herder, Fips
vpjcj80     Finn, Jean-Louis I
juzrv104    McGooter, Buzz G
ffdwq125    Brown, Lumps D
byccq148    Shucker, Liz
yqubs189    Brown, Jean-Louis
dhoim289    Torvalds, El P
xvytx321    Palmer, Porky
ndrup340    Budupadupa, Fyvush E
qrkuv342    Shaughnessy, Sara R
huuxi352    Hrumpf, Seymor J
gfnsp445    Hootenanny, Santos

12 rows selected.

DOC> * Authentication Query 4 (Determine the number of actual users, as well as
the number of
DOC> * active users)
DOC> */

NUM_USERS
---------
      500


ACTIVE_USERS
------------
         346
```

Appendix F: "authorization_queries.sql"

```
/******************************************************************************/
/* ECES 433, Final Design Project                                             */
/*     'authorization_queries.sql' - Performes the given 'Authorization'    */
/*            Queries.                                                        */
/* by Andy Reitz (reitz@ces.cwru.edu)                                         */
/* Date: 12/10/98                                                             */
/******************************************************************************/

/*
 * Authorization Query 5 (Find the record for a given userid)
 */
SELECT  *
FROM    Authorization
WHERE   userid = 'jekmf0';

/*
 * Authorization Query 6 (Find the users how have strict security settings)
 */
SELECT  userid
FROM    Authorization
WHERE   inacl IS NOT NULL AND outacl IS NOT NULL
        AND routing = 'FALSE';

/*
 * Authorization Query 7 (Find the users that have strict timeouts)
 */
SELECT  userid
FROM    Authorization
WHERE   timeout <= 3600;

/*
 * Authorization Query 8 (Find the administrators that have permissive
 * security settings)
 */
SELECT  Z.userid, A.name
FROM    Authorization Z, Authentication A
WHERE   A.userid = Z.userid
        AND A.type = 'admin'
        AND (Z.inacl IS NULL
             OR Z.outacl IS NULL
             OR Z.routing = 'TRUE');

/*
 * Authorization Query 9 (Find the 'worst' administrators [those from
 * Queries 8 and 4])
 */
SELECT  Z.userid, A.name
FROM    Authorization Z, Authentication A
WHERE   A.userid = Z.userid
        AND A.type = 'admin'
        AND (Z.inacl IS NULL
             OR Z.outacl IS NULL
             OR Z.routing = 'TRUE')
INTERSECT
SELECT  A.userid, A.name
FROM    Authentication A
WHERE   (A.type = 'admin' AND A.login = 'cleartext tempass1') OR
        (A.type = 'admin' AND A.pap = 'cleartext tempass1') OR
        (A.type = 'admin' AND A.chap = 'cleartext tempass1') OR
        (A.type = 'admin' AND A.global = 'cleartext tempass1');
```

Appendix G: "authorization_queries.out"

```
DOC> * Authorization Query 5 (Find the record for a given userid)
DOC> */
```

***** NOTE: The original output to this query was lost. I believe this to be an accurate reconstruction. *****

```
USERID      INACL    OUTACL   TIMEOUT IDLETIME ADDR            ROUTING ROUTE
----------- -------- -------- ------- -------- --------------- ------- --------
--------------------------------
```

```
jekmf0      11       242      NULL    NULL     NULL            TRUE
10.120.152.0 255.255.255.0 10.120.152.251

DOC> * Authorization Query 6 (Find the users how have strict security settings)
DOC> */

USERID
-----------
crjgq2
wpznp3
fhhry4
hslxe7
iguzr10
plitn12
oecpo15
nxjct16
fovyb18
uifuk19
qfpcc21
hpshb22
zjzvh23
gdmdn27
emylp30
oqhor36
zeizm37
perzy38
nquxl40
cjekx54
zxlzc56
ejnxs57
dedtu58
gcdid61
scohp66
wvred69
fsxcw72
vpjcj80
hhcsc84
eobyf85
izbvw101
juzrv104
ovwbo105
zigvq109
smqrr114
yrdet119
ojrtl120
olsye123
npicx131
pgvkv134
ingps135
oszbo136
dtens139
mrgxb142
pmrfq146
nsqyk152
ucupj154
wkzkl162
izuyu163
cbhpr164
umxbd169
zhmoc171
wumyj177
juewo181
ormzt198
```

```
yqlkd199
whjtv202
sqvxk208
buuyv209
lqwnm211
miooc214
xfjej219
fpvjp220
nsmgx228
uyvuo229
oihqv232
izsvs238
eicsv245
vkgku246
gejhz260
lfedu262
rjtph283
qesng290
wepgf291
viinv293
zcvds302
yolpq305
iqzoq307
vysro309
zgoqr316
lcduq330
wsdfl332
kmepv333
nhuet339
qrkuv342
pgoqn345
urgzq353
ilcqj358
qlygb359
zdxvg360
ipugv363
mqiic364
oobiv367
ikjyw370
zexxp372
rjpuv374
ufljg378
mjqgl387
sfpym389
hkvtp394
uscss396
lgjqi403
mwntk405
xqpxl406
xdcmu410
bptby416
cxnsn421
pryeg423
rgxgd427
ixvqy432
yljxh438
dkvhr440
dxpjv444
coyew446
unwjb447
qjlhu450
dhmcy456
yigiu461
```

```
fmqzg462
mvlvs464
oknjj472
pefvg475
yhwlu477
ewsvm479
lytvz480
cfduh484
efgbq487
jcsty493
obwjc496
eywsw497
gsbsk499

131 rows selected.

DOC> * Authorization Query 7 (Find the users that have strict timeouts)
DOC> */

USERID
-----------
plitn12

DOC> * Authorization Query 8 (Find the administrators that have permissive
security settings)
DOC> */

USERID      NAME
----------- ----------------------------------------
nwxhb13     Ocasio, Lumps
oewfc43     Monk, Ringo X
izift44     Groening, Bartley L
mwnxr53     Lipster, Flaudvie K
johko64     Crups, Zonker K
qximm102    Prune, The
rccwm107    Tanenbaum, Long G
scqnk113    Pytte, Amanda
bjixi121    Yada, Nadge V
ffdwq125    Brown, Lumps D
djenj143    Zabubadoofski, Boutros-Boutros
byccq148    Shucker, Liz
vtlst158    Hon, Elmo
emjbr186    Yokel, Hap
yqubs189    Brown, Jean-Louis
pzurd201    Puckett, Erasmus
rqqfo210    Fargo, Shirley U
chfsv221    Preen, Linus T
ysvby239    Daras, Andrew
xrphm253    Headroom, Irene Z
zymru287    McGriff, Paul
dhoim289    Torvalds, El P
pyymk320    Ringworm, Ivan F
xvytx321    Palmer, Porky
fpbml325    Dent, Seymor
pyesn331    Brown, Fedbo
iwxtp334    McGrooter, Meep I
ndrup340    Budupadupa, Fyvush E
huuxi352    Hrumpf, Seymor J
vzpmr375    Finn, Cletus G
xnlcw390    Crups, Minga
mxmym407    Puldup, Cousin S
zlecz429    Torvalds, Debbie V
tvhor434    Wang, Lanfried K
```

```
kscmg442    Tunklebit, Chups
gfnsp445    Hootenanny, Santos
zdjno452    McGee, Alexei
vjors458    Lopdop, Mussolini R
xfzxc494    Vader, Goofball

39 rows selected.

DOC> * Authorization Query 9 (Find the ’worst’ administrators [those from
Queries 8 and 4])
DOC> */

USERID      NAME
----------- ----------------------------------------
byccq148    Shucker, Liz
dhoim289    Torvalds, El P
ffdwq125    Brown, Lumps D
gfnsp445    Hootenanny, Santos
huuxi352    Hrumpf, Seymor J
ndrup340    Budupadupa, Fyvush E
xvytx321    Palmer, Porky
yqubs189    Brown, Jean-Louis

8 rows selected.
```

Appendix H: “accounting_queries.sql”

```
/****************************************************************************/
/* ECES 433, Final Design Project                                          */
/*     ’accounting_queries.sql’ - Performes the given ’Accounting’ Queries. */
/* by Andy Reitz (reitz@ces.cwru.edu)                                      */
/* Date: 12/10/98                                                          */
/****************************************************************************/

/*
 * Accounting Query 10 (Find the users that are currently logged into a
 * given NAS)
 */
SELECT  A.userid, A.name, C.start_time, C.stop_time
FROM    Authentication A, Calls_Completed C
WHERE   A.userid = C.userid
        AND C.stop_time > ’08-Dec-1998 07:00:00’
        AND C.NAS_hostname = ’IL_as2516’
UNION
SELECT  R.userid, R.NasUserName, R.log_date, TO_DATE(NULL)
FROM    Raw_Accounting R
WHERE   R.type = ’START’
        AND log_date < ’08-Dec-1998 07:00:00’
        AND R.NAS_hostname = ’IL_as2516’;

/*
 * Accounting Query 11 (Find the last logins for a given time period, in
 * reverse order)
 */
SELECT   C.userid, A.name, C.NAS_hostname, C.NAS_port, C.start_time
FROM     Calls_Completed C, Authentication A
WHERE    C.start_time >= ’07-Dec-1998 12:00:00’
         AND C.start_time < ’08-Dec-1998 23:59:59’
         AND C.userid = A.userid
ORDER BY C.start_time DESC;

/*
 * Accounting Query 12 (Find ’malformed’ logfile entries)
 */
SELECT   userid, NasUserName, NAS_hostname, NAS_port, type
FROM     Raw_Accounting
WHERE    type = 'STOP';

/*
 * Accounting Query 13 (Find the usage history for each user, for a given
 * time period.)
 */
SELECT   A.name, C.userid, C.duration, C.start_time, C.NAS_hostname, C.NAS_port
FROM     Calls_Completed C, Authentication A
WHERE    C.userid = A.userid
         AND C.start_time >= '07-Dec-1998 12:00:00'
         AND C.start_time < '08-Dec-1998 23:59:59'
GROUP BY C.userid, A.name, C.duration, C.start_time, C.NAS_hostname,
C.NAS_port;
```

Appendix I: "accounting_queries.out"

```
DOC> * Accounting Query 10 (Find the users that are currently logged into a
given NAS)
DOC> */

USERID      NAME                                     START_TIME
STOP_TIME
----------- ---------------------------------------- -------------------- -----
---------------
btlee243    Dilmont, Tex                             08-DEC-1998 03:43:53 08-
DEC-1998 23:17:55
bwdun17     Hermfik, Darth                           07-DEC-1998 21:38:01
clntv41     Toothpaste, Crowly                       07-DEC-1998 20:16:33
flbsg33     Gompers, Shampoo N                       08-DEC-1998 01:59:00
fovyb18     Lester, Chief                            07-DEC-1998 20:14:28
gdmdn27     Merat, Duke                              07-DEC-1998 18:00:58
ifbsq74     Bator, Natarajan F                       08-DEC-1998 02:36:36 08-
DEC-1998 22:00:53
jbonl8      Lumpwump, Santos                         07-DEC-1998 19:07:16
jcrxm11     Lewis, Bartley                           07-DEC-1998 23:37:19
jekmf0      Tiddleflip, Greedo                       08-DEC-1998 02:13:58
ldvif6      Dipdip, Luke R                           07-DEC-1998 23:39:44
nquxl40     Wallop, Orange M                         07-DEC-1998 18:28:53
nuixz399    Dent, Ringo                              08-DEC-1998 03:47:12 08-
DEC-1998 23:07:27
perzy38     Gassee, Toadstool                        08-DEC-1998 03:21:07
plitn12     Qureshi, Professor                       08-DEC-1998 03:31:37 08-
DEC-1998 23:24:15
rlukh31     McGooter, Ahura                          08-DEC-1998 03:01:42
rucwe25     Crumchuck, Meep                          07-DEC-1998 20:22:08
vpjcj80     Finn, Jean-Louis I                       08-DEC-1998 03:09:28 08-
DEC-1998 22:00:34
vzvtt49     Yokel, Minga J                           07-DEC-1998 20:59:39
wpznp3      Zebo, Roto-Rooter N                      07-DEC-1998 18:24:29
wupdv165    Yummy, Liz                               08-DEC-1998 02:54:26 08-
DEC-1998 22:01:17
yyptm159    Bavarian, Goofball                       08-DEC-1998 02:57:19 08-
DEC-1998 22:08:36
zeizm37     Hrumpf, Pam C                            07-DEC-1998 20:35:41
zjzvh23     Kagy, Fedbo                              08-DEC-1998 00:01:42

24 rows selected.

DOC> * Accounting Query 11 (Find the last logins for a given time period, in
reverse order)
```

```
DOC> */

USERID      NAME                                     NAS_HOSTNAM NAS_PORT
START_TIME
----------- ---------------------------------------- ----------- -------------
-- --------------------
ndrup340    Budupadupa, Fyvush E                     IL_as2516   2516_Async_24
08-DEC-1998 01:59:56
owtht315    Leech, Santos                            OH_as5200   5200_Async_15
08-DEC-1998 01:59:51
bjxcw319    Young, Bartley                           OH_as5200   5200_Async_132
08-DEC-1998 01:59:15
zyeod9      Batur, Ben O                             OH_as5200   5200_Async_43
08-DEC-1998 01:57:09
fpbml325    Dent, Seymor                             OH_as5200   5200_Async_132
08-DEC-1998 01:56:53
emjbr186    Yokel, Hap                               OH_as5200   5200_Async_177
08-DEC-1998 01:56:47
nwxhb13     Ocasio, Lumps                            IL_as2516   2516_Async_28
08-DEC-1998 01:56:38
pxquv26     Lewis, Chaz L                            OH_as5200   5200_Async_121
08-DEC-1998 01:56:33
oqhor36     Broom, Roto-Rooter N                     OH_as5200   5200_Async_216
08-DEC-1998 01:56:28
nsmgx228    Rabiniwitzin, Irene                      IL_as2516   2516_Async_12
08-DEC-1998 01:56:10
hpshb22     Bologna, Ernie T                         OH_as5200   5200_Async_64
08-DEC-1998 01:55:56
ixvqy432    Smee, Luke U                             IL_as2516   2516_Async_7
08-DEC-1998 01:54:49
hfmgm250    Ernst, Bartley                           IL_as2516   2516_Async_14
08-DEC-1998 01:54:46
tynng356    Strunker, Nerf                           OH_as5200   5200_Async_76
08-DEC-1998 01:54:32
cyyjv126    Godse, Roto-Rooter                       IL_as2516   2516_Async_19
08-DEC-1998 01:54:29
sbgtk278    Pip, Professor G                         IL_as2516   2516_Async_20
08-DEC-1998 01:54:29
qximm102    Prune, The                               IL_as2516   2516_Async_0
08-DEC-1998 01:54:28
clxnh365    Tunklebit, Homer                         OH_as5200   5200_Async_45
08-DEC-1998 01:54:00
clntv41     Toothpaste, Crowly                       OH_as5200   5200_Async_44
08-DEC-1998 01:53:58
pyymk320    Ringworm, Ivan F                         IL_as2516   2516_Async_26
08-DEC-1998 01:53:46
pxhoj430    Nihlen, Roto-Rooter                      OH_as5200   5200_Async_174
08-DEC-1998 01:53:15
wkpyu179    Oro, Illapotin J                         IL_as2516   2516_Async_23
08-DEC-1998 01:52:58
gdmdn27     Merat, Duke                              IL_as2516   2516_Async_9
08-DEC-1998 01:52:57
coyew446    Horn, Lanfried                           IL_as2516   2516_Async_0
08-DEC-1998 01:52:45
fjylc118    Torvalds, Binky                          OH_as5200   5200_Async_200
08-DEC-1998 01:52:36
pshwz86     Duggan, Ulan Z                           OH_as5200   5200_Async_59
08-DEC-1998 01:51:57
izuyu163    Puldup, Gerp U                           IL_as2516   2516_Async_4
08-DEC-1998 01:51:54
torbc170    Harasmatari, Shampoo X                   OH_as5200   5200_Async_90
08-DEC-1998 01:51:38
```

```
...
...
```

The rest of this output has been removed from the printed version, due to length. The full text can be found on the electronic submission media.

```
...
...

254 rows selected.

DOC> * Accounting Query 12 (Find 'malformed' logfile entries)
DOC> */

USERID      NASUSERNAME                              NAS_HOSTNAM NAS_PORT
TYPE
----------- ---------------------------------------- ----------- --------------
-- ------
gxlzq1      Gouda, Gilbert N                         OH_as5200   5200_Async_108
STOP
crjgq2      Nutter, Pops                             OH_as5200   5200_Async_0
STOP
fhhry4      Hooloo, Liz                              OH_as5200   5200_Async_176
STOP
xswbk5      Funtz, Toadstool D                       OH_as5200   5200_Async_225
STOP
hslxe7      Vueigez, Mussolini                       IL_as2516   2516_Async_21
STOP
zyeod9      Batur, Ben O                             IL_as2516   2516_Async_14
STOP
iguzr10     Pup, Freep W                             IL_as2516   2516_Async_11
STOP
plitn12     Qureshi, Professor                       IL_as2516   2516_Async_31
STOP
nwxhb13     Ocasio, Lumps                            IL_as2516   2516_Async_21
STOP
qsdyc14     Fumbucket, Buck                          IL_as2516   2516_Async_25
STOP
uifuk19     Vader, Clunky                            IL_as2516   2516_Async_9
STOP
mzvgt20     Chiller, Fivel                           IL_as2516   2516_Async_24
STOP
qfpcc21     Bator, Fyvush                            IL_as2516   2516_Async_27
STOP
pxquv26     Lewis, Chaz L                            IL_as2516   2516_Async_29
STOP
tqjnv28     Horn, Frink G                            OH_as5200   5200_Async_56
STOP
emylp30     Blatch, Dominic H                        OH_as5200   5200_Async_38
STOP
sgrxo35     Goesh, Tryfon I                          OH_as5200   5200_Async_102
STOP
oqhor36     Broom, Roto-Rooter N                     IL_as2516   2516_Async_29
STOP
kkygo42     Neff, Crunch                             IL_as2516   2516_Async_3
STOP
dwknj48     Toothpaste, Duke                         OH_as5200   5200_Async_135
STOP

20 rows selected.

DOC> * Accounting Query 13 (Find the usage history for each user, for a given
time period.)
```

```
DOC> */

NAME                                     USERID       DURATION START_TIME
NAS_HOSTNAM NAS_PORT
---------------------------------------- ----------- --------- ----------------
---- ----------- ----------------
Wang, Geezy                              bbwwg266            0 07-DEC-1998
22:07:52 OH_as5200   5200_Async_98
Cuervo, Nogo J                           bcieu449            0 07-DEC-1998
23:02:36 OH_as5200   5200_Async_37
Cuervo, Nogo J                           bcieu449            0 07-DEC-1998
23:55:42 OH_as5200   5200_Async_244
Cuervo, Nogo J                           bcieu449            0 08-DEC-1998
00:55:54 OH_as5200   5200_Async_116
Cuervo, Nogo J                           bcieu449            0 08-DEC-1998
03:08:29 OH_as5200   5200_Async_156
Cuervo, Nogo J                           bcieu449            1 08-DEC-1998
02:14:33 OH_as5200   5200_Async_179
Neff, Long                               bdqir75             0 07-DEC-1998
19:51:10 OH_as5200   5200_Async_160
Neff, Long                               bdqir75             0 07-DEC-1998
21:31:00 OH_as5200   5200_Async_159
Neff, Long                               bdqir75             0 07-DEC-1998
22:12:19 OH_as5200   5200_Async_33
Neff, Long                               bdqir75             1 07-DEC-1998
17:57:29 OH_as5200   5200_Async_50
Neff, Long                               bdqir75             1 07-DEC-1998
21:16:35 OH_as5200   5200_Async_239
Young, Fritz F                           bgqqc392            0 08-DEC-1998
01:13:38 IL_as2516   2516_Async_13
Young, Fritz F                           bgqqc392            0 08-DEC-1998
02:29:08 IL_as2516   2516_Async_26
Young, Fritz F                           bgqqc392            1 08-DEC-1998
01:12:32 IL_as2516   2516_Async_14
Young, Fritz F                           bgqqc392            1 08-DEC-1998
01:22:39 IL_as2516   2516_Async_4
Young, Fritz F                           bgqqc392            1 08-DEC-1998
03:03:44 IL_as2516   2516_Async_27
Yada, Nadge V                            bjixi121            1 07-DEC-1998
23:01:36 IL_as2516   2516_Async_6
Young, Bartley                           bjxcw319            0 07-DEC-1998
18:21:28 OH_as5200   5200_Async_23
Young, Bartley                           bjxcw319            0 07-DEC-1998
20:01:31 OH_as5200   5200_Async_29
Young, Bartley                           bjxcw319            0 07-DEC-1998
23:56:25 OH_as5200   5200_Async_3
Young, Bartley                           bjxcw319            0 08-DEC-1998
00:10:35 OH_as5200   5200_Async_155
Young, Bartley                           bjxcw319            1 07-DEC-1998
21:20:43 OH_as5200   5200_Async_22
Young, Bartley                           bjxcw319            1 08-DEC-1998
01:27:06 OH_as5200   5200_Async_190
Young, Bartley                           bjxcw319            1 08-DEC-1998
01:32:08 OH_as5200   5200_Async_80
Young, Bartley                           bjxcw319            1 08-DEC-1998
01:59:15 OH_as5200   5200_Async_132
Headroom, Ripper                         bptby416            0 07-DEC-1998
19:21:54 OH_as5200   5200_Async_39
Headroom, Ripper                         bptby416            0 07-DEC-1998
19:51:11 OH_as5200   5200_Async_154
Headroom, Ripper                         bptby416            0 07-DEC-1998
23:32:02 OH_as5200   5200_Async_249
```

```
Headroom, Ripper                        bptby416           0 07-DEC-1998
23:45:11 OH_as5200   5200_Async_243

...
...
```

The rest of this output has been removed from the printed version, due to length. The full text can be found on the electronic submission media.

```
...
...

2170 rows selected.
```

[1] A PRI is an acronym that stands for "Primary Rate Interface". A PRI is basically a bundle of 23 telephone lines that share a common phone number. When a user dials the phone number, the telephone system will select a free one, and pipe it into the NAS. It is further possible to segregate the PRI, giving it multiple phone numbers. In this case, the NAS will log what number was actually called.

[2] CHAP stands for the Challenge-Handshake Authentication Protocol, and implements a three-way handshake between the client and the NAS, in order to ensure a secure authentication exchange.

[3] VPDN is an acronym, short for *"Virtual Private Data Network"*. This technology allows the private LAN to be extended securly and seamlessly across the WAN, to the remote client.

[4] Oracle is currently running a rather impressive challenge that implies that their database is the top performer in today's market. Reference: http://www.oracle.com/challenge/.