

## Lecture 2

# INTRODUCTION TO DBMS ADMINISTRATION & SECURITY

### *Narration Script*

**M. E. Kabay, PhD, CISSP-ISSMP**

*This script is provided for compliance with the Americans with Disabilities Act.  
The numbers correspond to the slide numbers in the PowerPoint and Flash presentation.*

1. Hello and welcome to the lecture on database management systems and database management system security administration. My name is Mich Kabay and I've been working with databases for 30 years so I hope that I'll be able to give you some background that will be useful to you.

2. As you can see on this slide we're going to be covering a fair amount of territory. Now unlike the fire hose that you see in the picture, this PowerPoint presentation *can* be stopped and restarted at any time. In other words, you don't have to listen to everything all in one extremely long lecture. I encourage you to use this tool as it best suits you: you can listen to a part at a time, or several parts in a row. If you are using PowerPoint itself, you can use the little *Slide Show* icon at the bottom of your window or you can use the Slide Show menu at the top of your window to play the show from any current slide. So you can go back and listen to a section again before going on. Just don't panic because there are 90 slides. Please.

3. In our first section, I'd like to give you a sense of why databases matter to *security* experts. In the context of our first seminar, it is useful for you to grasp what a tremendous improvement the relational database model was for data management in the 1960s and 70s. The problems we faced then were deeply related to security, as you will see.

4. Databases are almost universal in today's information technology. Understanding how databases are constructed speaks to our need in security for a supportive relationship to our users, because data requirements and data relationships are at the heart of security requirements. As I'm sure you've heard many times, it's the rare organization where security is the driving force; we serve the strategic goals of the organization and that means we need to understand data requirements. On another level, there are security implications to how programs and data structures work; understanding how databases work gives us insights into why the user interfaces

# MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

## *Introduction to DBMS Admin & Security*

---

work as they do and, even more important for security personnel, how systems can fail or be abused.

On a practical level, you may yourselves need to create a database and having a solid grasp of the principles will help you assimilate the details of any specific points you need to learn. Similarly, structured query language or SQL is almost universally used throughout the industry and being familiar with these tools increases the likelihood of getting good jobs.

5. Nobody should imagine that storing and using information started with the advent of computers.

At a fundamental level, I suppose that data processing began at its most elementary with the rise of living systems in the deep mists of the Precambrian Era some four billion years ago. But in more significant terms by human standards, we probably need to focus on the transmission of information from one person to another via oral communication some time around a quarter of 1 million years ago through 100,000 years ago (scientists really are not sure exactly when speech developed) and then through what we now call oral traditions – recitation of structured histories from generation to generation.

We do know that mnemonics, that is, techniques for helping to remember information accurately, date back at least 5000 years, as do clay tablets on which large amounts of information could be pressed using a stylus.

At about the same time, people in the near and Middle East were also experimenting with compressed plant fibers to create thin sheets of papyrus on which they inscribed vast amounts of information, some of it quite boring accounting data about commercial traffic.

About two centuries before the common era, thin layers of leather were converted into what we call parchment, which provided an excellent basis for creating scrolls such as those used to write down the oral traditions of the Jewish people in the Torah.

Some 300 years later, in the second century of the common era, the invention of paper expanded the availability of information storage and 300 years later people had the brilliant idea of using rectangular pieces of paper to form what were called codices (singular is the codex) which had the tremendous advantage of allowing what we now call random access in contrast to the serial access forced by the scroll. That is, people could flip a book open to a specific page and even be given a page number to locate information instead of having to roll and unroll scrolls while constantly looking at the contents to locate the information they needed.

We will skip over the invention of printing and point out that the invention of punch cards at the end of the 19th century – and of their readers and tabulating machines – provided yet another drastic increase in information processing capacity.

By the mid-20th century, electronic data storage depended on files in analogy to paper files; and as we will see, those electronic analogs of paper files had all of the functional and operational problems of paper files, but could let people make the mistakes much faster.

# MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

## *Introduction to DBMS Admin & Security*

---

Database management systems developed in the 1970s and have been central to effective data management since.

An interesting recent development is exemplified by tools such as Google Desktop, an indexing system that allows almost instantaneous access to unorganized content via keyword search. This kind of tool has begun to replace tedious cataloging of such items as e-mail messages, which have typically been difficult to place in single containers because they so often deal with more than one topic despite the best efforts of people like me who try to convince everyone to limit messages to a single coherent purpose.

6. But what was the problem with file systems of the 1960s and 1970s? Why did databases provide such a welcome relief? Relief from what? There were some pretty serious difficulties for programmers and system managers in those days and we'll look at each of them in turn.

7. There was nothing to provide linkages among the data stored in files. If a programmer needed to use information in one file as a pointer to locate information in another file – for example to look up an order number for a customer so that she could locate the order details for a current order – everything had to be done in a specific program. That's everything. I mean really everything. Keeping track of record numbers, doing comparisons between a search value and the values in records that were being read serially, defining the record structures for every file – these functions resided in each individual program. Make a mistake in writing one program and the results could ricochet through the entire system by corrupting values that would then be read by other programs. There were no safeguards on what you could put into a file other than what a specific program written by specific programmer happened to include as a limitation. Did you want to change a restriction on the values in a particular field? Sure – go find every single program needing the change and then make all the changes, and then recompile all of those programs and hope you got it right.

8. Another problem was that in the absence of any theory for organizing data it was very easy for programmers to duplicate data. For one thing, there might be more than one programmer on the system; so it wasn't unexpected that, say, a customer record might be duplicated in files used by the accounting department for billing and by the order-processing department for inventory controls. The problem occurred when somebody needed to change the customer information – for example, if the customer got a new telephone number. Because there was no theoretical basis for deciding how to store information relating one type of data to another, it was hard for programmers to figure out what they should do with records of, for example, employees of companies. Suppose you needed to store data about authorized agents who could initiate transactions in a financial system; should you store for details about every person in a file for authorization codes? Would that information include their telephone number? Their address? Great – what if the main address and telephone number of their employer were to change? You would have to change every occurrence of the original value into the new value for the records to

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

be correct. But then what about historical records? Should they be changed or not? There were no principles for people to base their reasoning on.

9. I've already mentioned the relationship between the file structures and the programs; the *file-format dependency problem* referred to the tight coupling of undocumented attributes of the records within those files to the programs using them. I say "undocumented" because there was nothing in the file itself that described the record structure; the only description was in the program itself. Sometimes, programmers would scribble file definitions on pieces of paper; I personally remember from my programming days in the 1960s and 1970s seeing clipboards with file structures hung on nails on the wall in the programming area.

10. File incompatibilities could develop very easily under such circumstances; different programmers could easily diverge in their file structures without knowing it. Even a single programmer might forget the details of a less-used file and find a program wreaking havoc by mistake by reading information and interpreting it wrongly or by trashing existing records with malformed data. Even something as simple as the name of a specific field could cause problems when different programmers or a forgetful individual programmer stored the same data by different names in different files. In the absence of any kind of enforcement or flexibility, woe betide the programming team that changed the order of fields and records that should have been the same.

11. It was also difficult for programmers to provide useful representations of the data in their collections of isolated files. Every report was a struggle; we had to get the right file structures, incorporate them into our data definitions within the programs, create the output record formats in detail – every single position, every punctuation mark, every separator – and even, in the older systems, take into account the specific attributes of the printers we were dealing with.

This is a little aside, but the chaos and disorder of data structures was mirrored by the lack of standardization in printer codes. Every program had to include specific codes for specific printers. If you changed printer, you had to recompile your program; there were no printer drivers that were independent of the programs writing directly to the printers.

For that matter, in the early days, there were no spoolers (SPOOL meant Shared/Simultaneous Peripheral Operation On Line): a program controlled a given printer directly – if by mistake, another program started writing to a printer in use, you might actually get a mixture of data appearing in a jumbled mess on the printer. Spoolers were in a sense a parallel development to database systems: they provided a standardized method for control of printers in a way that eliminated the need for special coding within each program and they took care of the next problem faced by filesystems: concurrency.

12. Concurrency, which we will be discussing in detail later in the lecture, was a nightmare for early programmers. You'll hear this again, but you have to get the idea solidly embedded in your

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

brain: if two programs access a single data file and both programs attempt to make changes to the same data, there is a real possibility that the second one to write into the file will destroy the changes made by the first one. Granted, one approach was to restrict access to the files so that only one process can access the data at a time; however, seizing the entire file was out of the question for multiuser systems where dozens or hundreds of users (that was a lot of people 40 years ago) were supposed to be accessing the data at the same time. Serializing access to the entire file had unacceptable performance consequences. File access privileges were developed early in the history of multiuser operating systems; a standard mnemonic of the time was RWALX to remind programmers that they could open files with read, write, append, or, if we were speaking of a program file, execute access. Let's take a look at concurrency in a just a little more detail on the next slide.

13. I don't think I need to belabor this point – I'm sure you're thinking I've already belabored it – just read the slide and answer the question: how many Widgets are left in reality after Joe and Shakheena both withdraw their amounts from their copies of the original inventory total? Well, you can see that although Shakheena's overwriting of Joe's total seems to put 20 Widgets in inventory, actually there are only 10 left. And you can understand why this problem was labeled the lost update.

14. Dr E. F. Codd was working for IBM in their San Jose research laboratory when he developed the relational database model that he published in 1970. He and fellow IBM scientist Christopher J. Date pursued the relational model and were instrumental in developing the Structured Query Language that we universally know as SQL, usually pronounced "sequel" and sometimes simply by the letters of its acronym. There is no way that I can include a full course on relational database theory in this lecture without taking up a week of time and forcing you back into a lab to do exercises. So instead, all I want to do is to give you a few basic concepts in the hope that it will refresh the memories of those of you who have studied the material and prompt those of you who haven't into reading more about it.

15. There are some pretty significant words in the definition before you. Read it carefully: a database is a self-describing collection of integrated records. And it is a – and this must sound mysterious – model of a model. We will look at each of these in turn.

16. You'll recall how much trouble unstructured collections of individual files can be; databases are collections of files that include self-referential data. Data about the data are called metadata; they can be read and interpreted by the database management system and thus used to communicate with the application programs accessing the data. The data dictionary contains some (not all) of the metadata and defines attributes of all the fields and records, including logical constraints on the data ranges and relationships among the data. Metadata include names, editing constraints such as allowable ranges, and relationships among records such as the number

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

of records that share a common value (called a key). Because the metadata reside with the data instead of inside the compiled programs, maintaining programs becomes vastly simplified. In the same way, the metadata provided documentation about the database, reducing the errors and inconsistencies that are inherent with manually maintained, replicated copies of system documentation.

Incidentally, I happen to continue using words with “data” as plurals because of my early years of learning Latin, but almost no one else does anymore. You can be perfectly correct nowadays saying “data is” or “metadata is.”

17. Metadata extend integration by providing performance-enhancing indexes for common lookups; databases can even store default formatting such as number of visible decimal places in displaying a field, for output such as displays or paper reports.

18. A curious phrase “a database is a model of a model” expresses the notion that databases do not have to be what is called *isomorphic* with reality; that is the structure of the database is an abstraction from reality. Indeed, it is a *second-order* abstraction in the sense that a database designer represents *her* view of what she understands from interviewing *users*. Naturally, the users are expressing *their* perspective on reality. The implication is that a database design should never be viewed as rigidly fixed for all eternity; it is an *instantiation* of one *interpretation* of a *view* of reality.

Hey, deep, man....

Another comment is something that I have been teaching for decades: the availability of the tool determines perceptions of what is possible. One of the most striking experiences I ever had as a consultant occurred in the mid-1980s when I was just striking out on my own. I was helping a clothing factory in Montréal to optimize their databases and the head of IT and I were walking through the offices one day when I stopped dead in my tracks. I pointed to an employee at a desk and asked the VP if we could go speak to him. “Hi,” I said, “whatcha doin’?” Well, said employee, he was calculating subtotals based on a report. You see, it was precisely the sight of someone using a calculator with a computer printout that got my curiosity going. “Have you done this before?” I asked. “Sure,” he said, “every month for the last three years.” “Did you ever ask anybody to put the subtotals into the report for you?” He stared at the VP and me in astonishment and said, “They can DO that?” And this is why I am telling you this story. It seems that nobody had ever walked around finding out what the employees needed or telling them what was possible. The lesson, even for security experts, is that we need to use the time-tested technique of MBWA: management by walking around. There is no substitute for contact with reality. All the reports in the world are just hearsay: go out and see for yourself what’s happening in your working environment.

19. So what is a DBMS? A database management system includes databases and the tools for building, modifying, monitoring, securing, and repairing them. A database contains *files* –

# MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

## *Introduction to DBMS Admin & Security*

---

sometimes people call them *tables* and in some systems they are called *relations* or *datasets* – each of which consists of *records* (also called *rows* or *tuples*) in which there are *fields* (also called *columns* or attributes). Designers establish *relationships* or *links* among tables to help locate data; these linkages among the tables help users navigate through the database.

It's the database *application* that provides a particular way of accessing the database; for example a particular database application might be an accounting program or a production-control program or a patient-records program. Database applications provide for our control over what constitutes acceptable data; for example an inventory system can include provisions for defining reorder points to prevent exhaustion of parts supplies.

Database applications provide user interfaces so that employees can enter data quickly and correctly as well as locating data using interactive queries or stored procedures. Applications normally include capabilities for creating new reports as well as for generating predefined reports.

The DBMS also includes a database application with general capabilities often called a *query* program. Users with little or no programming experience can use the query program to work on subsets of their data to answer specific questions, for calculations and to generate reasonably sophisticated reports with features like sorting, subtotals, headers and even graphs.

20. The diagram in this slide represents the relationship among components of a database management system. Access by an application program to the data flows through an application program interface or API which in turn depends on the internals of the DBMS to interpret metadata from the data dictionary. The metadata translate requests for functional definitions of the data such as the name of a patient into pointers to records and descriptors of the specific parts of the records that correspond to the needed data.

21. In the next few slides will look at some of the basics of relational database theory. Again, this is not a substitute for a course in database, but it will suffice to get some key (no pun intended) ideas across that we can use in our discussions of security.

22. I have already alluded to some of the synonyms that are used in describing the components of databases. Here are some nifty diagrams from undergraduate courses I teach that should clarify the relationships.

23. The relational model has a number of strict requirements that are summarized on this slide. The most important is that every single record must be unique. That means in practice that we are going to have to name an attribute of the information that is naturally unique or alternatively, to impose one that we can force to be unique. We call the unique identifier of the record the *key*.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

24. Keys make an enormous difference to the structure and performance of databases. Keys can consist of a single field or of several fields that are concatenated to form a *compound key*. Keys can be used to create a special set of pointers called an *index* (plural *indexes* or *indices*) that can greatly speed access to records. For example, if *patient\_ID* is a key for treatment records, then one could have the database almost instantly retrieve the records of treatments for a specific patient without having to read all of the patient records to find the right ones. The *patient\_ID* key allows the DBMS to use random access (direct I/O) with a specific record number instead of serial access (serial I/O).

25. Defining appropriate keys is critically important in designing databases. The choice of keys depends very much on the kinds of questions that users typically ask; one of the concerns is that keys add to the overhead of the database – both in terms of requiring extra storage space for the pointers (some define a chain of values with the same key and others point to the start or end of the chain) and in imposing a performance cost whenever we add or delete records (because we have to modify pointers to keep the chain descriptors correct).

As the slide points out, choosing the right keys is at the heart of the database designers skillset. Just as an example, imagine an order-entry system in which the only key to the order dataset were mistakenly defined as *customer\_ID*; it would be impossible to have more than one order in the dataset per customer – a ridiculous constraint.

26. Continuing our imaginary order-entry system example, suppose an amateur database designer decided to include full customer information (name, address, telephone number) in the same record as the order number and the date of an order (what we call an order-header, which corresponds to the top of an order form). Suppose a very large customer had 3,000 orders in the database – are we to accept that there should be 3,000 copies of the same information stored uselessly in the order header file? Ridiculous. We prevent these problems in the process called *normalization*.

27. Non-normalized data structures can cause a real mess. Here's a terrible design for keeping track of what clients are buying at a store. What on earth are we going to do to keep track of the items in our store if we delete the last record that includes the item information? How do we store information about an item that nobody has bought yet? Why are we storing copies of the information about the same items and the same people?

28. This slide gives a couple more examples of the *deletion anomaly* that results from badly normalized data structures. You can see that these erroneous record layouts mix information about fundamentally different entities. There's no good reason to store details about a doctor in the record pertaining to a patient. There's no good reason to store information about a car in the same record as price information about types of repairs. These non-normalized data structures don't make sense even if we don't know much about databases.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

29. Just as we saw in previous slides, a dataset that mixes detailed information about a part with radically different information about where the parts are stored causes a real headache. We need to separate the information about the parts from the information about how many there are and where they are kept. What we can do is to keep a record that links a specific part number to a specific bin number and lists how many of that part are in that bin. Now that is a normalized design and it easily handles information about parts that are out of stock, bins that are empty, and parts that are in more than one bin.

30. We've already seen some of the problems that can occur in a poorly structured database when we delete records. Here are some more. In particular, I draw your attention to what could happen if we have a library database in which we delete the record with information about the publisher when there are many books that point to that publisher. You can see that removing the only record that tells us the name, address, telephone number and so on for the publisher could be very harmful for the library and would make the book records that point to the nonexistent publisher record meaningless (for example, they might have the number 2345 which used to point to Random House which now wouldn't mean anything at all).

Databases have strict rules which make it impossible to cause this kind of havoc; we call these rules *referential integrity constraints*. Other examples include not being able to add a record that duplicates a unique key value in a dataset or not being allowed to add a record that points to a nonexistent key. As an example of the latter, an order-entry system would prevent an operator from adding an order for a client that does not yet exist in the database. First you add the client record, then you can add the order placed by that client.

You are familiar with such constraints simply from having bought things on the Web, where it is perfectly normal for us all to fill out an identification form about ourselves *before* we can place our order containing the items we want to buy. The next time, however, if we allow the vendor to keep our information, we can just identify ourselves and fill out the details of the new order.

31. As I said before, the basic idea of normalization is that we should not mix up information about distinct entities. As a general rule, if you can define a set of data that refers exclusively to a single entity, it should be stored in a single record in one table. For example on this slide, information about a doctor that is unique to that doctor gets stored in a single record: the doctor's identifier, the doctor's name, and whatever else you want – the address, telephone number and medical registration – that would all be part of the doctor's record.

Should the specialty of the doctor be part of that record? It's quite possible for a doctor to have more than one specialty and therefore, if that's allowed in the database – if we make the assumption that the doctor can have more than one specialty – we *can't* put the specialty in the same record as the doctor identifier which we are going to make the unique identifier. We would have to have a *separate table* which might define specialties and have an identifier for each specialty; and then we would have a *cross-index table* that simply lists the *doctor\_ID* and the

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

*specialty\_ID*. There will be as many records for a given doctor as he or she has specialties. So if the doctor had three specialties there would be three records in the little cross-index file.

As a side note, normalization forbids putting three slots into the doctor record to make allowance for up to three specialties; that non-normalized design leads to a mess as soon as there is a doctor with a fourth speciality. At that point, we would have to re-structure the database and recopy the old data into the new dataset. Not only that, but since most doctors have one speciality, most doctor records would have wasted space. Finally, any programs with a reference to the three allowable specialties as a hard-coded element of their global data definitions would have to be patched and (if compilable) recompiled.

Going back to our little diagram on this slide, the same considerations apply to patients. We put information that is unique to a given patient in the patient record. That cannot include appointments, since it's perfectly reasonable that a patient could have more than one appointment. In addition, an appointment naturally involves information about which doctor the patient will be seeing. Clearly, we cannot put a *doctor\_ID* in the patient record. We need a separate dataset: a relationship table that links a doctor to patient.

By the way, what do you think the unique key has to be in the appointment dataset? Think about it:

- *doctor\_ID*? Nope. Obviously a doctor can see many patients, so there will be many records with the same *doctor\_ID*.
- Can it be the *patient\_ID*? No, because a patient can have more than one appointment.
- Can it be the concatenation of *doctor\_ID* and *patient\_ID*? No, because that would mean that the dataset would force a constraint that a doctor could see a patient only once.
- Aha! It has to be the concatenation of *doctor\_ID* and *patient\_ID* and *date*, right? Well, almost. Even that compound key embodies the assumption that there can never be more than one appointment in a single day for a specific patient with a specific doctor.
- Presumably an effective compound key would include *patient\_ID*, *doctor\_ID*, *date*, and *time*. Even that design makes the assumption that a patient could never be seen by more than one doctor at a single time – but what if there were a joint consult?

Sigh. I hope you're getting a sense of what is involved in database design!

32. Now, I absolutely refuse to go any further in normalization theory at this point. As you can see on the diagram, where NF refers to normal form, there are many levels of normalization and you can study them, if you haven't already, in any textbook on databases.

33. Acceptance of the relational model was not instantaneous; for example, IBM itself resisted Codd's model for some years, preferring to support its own non-relational databases, much to the frustration of Codd and Date. Similarly, HP3000 computers, for example, were using IMAGE/3000, a *network-model* database that allowed only two levels of linkages among files.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

When microcomputers swept into the world of industry, the most popular database of the time was the non-relational dBase II.

However, by the mid-1980s, relational databases were spreading throughout the industry and were at the heart of the *client/server model*. The relational model is ideal for the growing use of *distributed databases*, in which specific datasets can be allocated to particular servers and their data aggregated through the telecommunications lines to summary datasets as required.

34. As the World Wide Web took off in the early 1990s with the opening of the Internet to the *.com* world, databases played an increasing role in the *distributed-computing* model that used *browsers* on client systems as a universal interpreter for the graphical user interface to databases residing on provider servers.

- In what we call *Web 2.0*, where an increasing amount of information can be generated by users of Websites as well as by the owners of those sites, databases continue to grow in importance. Many production Websites use backend databases to store all of the content that is dynamically generated as HTML pages; even user input is stored in the databases.
- *MySQL* is an immensely popular DBMS that has an open-source version as well as a commercial, professional version available.
- And now, as we close out the first decade of the 21st century, *cloud computing* has become an exciting model for distributing our computational tasks and, inevitably, sharing data through distributed databases.
- *Software as a service* (SaaS) depends on backend databases in many ways such as registering licensed users and providing fast authentication of those users when they connect to the SaaS servers.

35. Throughout the history of database development, there have always been questions about the ethical and legal constraints on what kind of information can be collected and stored about data subjects, especially when those data subjects are human beings. Take for example the automated recording of license plate numbers at tollbooths only for people who don't pay tolls; abusing those data could be difficult. But what about databases containing time-stamped records of books borrowed at a library or purchased at a bookstore that are collected as part of an anti-terrorist watch; could such records be abused if the political climate changed to something more like the McCarthy era of the early 1950s in the US? These issues go far beyond technical questions about database design; they are fundamental questions about the use and misuse of personally-identifiable information. One of the ways we discuss the technical security issues that can underlie the ethical and political issues is by referring to the Parkerian Hexad of which you have read already in this seminar and will study later as well.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

36. Okay. Let's plunge into part two of this long introduction to database management and security. In this section, we're going to look at some elements of database administration and then discuss in more detail specific issues of concurrency control.

37. Database administration is very much what it sounds like: adapting the contents, metadata, and sometimes (not always) the structure of databases. As you can see on the slide, the DBA would be the person to add authorized users for the accounting database, remove a userID for the engineer who has left the company, initiate a scan for database errors after a system crash, investigate slow performance reported by the customer service department, and so on. DBAs have a variety of backgrounds; not all of them come from the programming staff. One recommendation that I've always made is that DBAs should either have or receive sound training in security principles and practice and especially in the tools they need to secure the databases for which they are responsible.

38. Here's a short list of functions typically performed by DBAs. The DBA for production databases must be involved in business continuity and disaster recovery planning. They should also have a service orientation towards their customers, and I use the word deliberately. Like all IT and security personnel, these folks must remember that their primary concern is helping other people in the organization get their jobs done efficiently and effectively. Nobody except hobbyists runs databases simply for the joy of running databases. Keeping track of performance and knowing how to improve it are essential components of the DBA's job.

39. DBAs can constructively be part of system design teams; their insights into database internals and functionality can greatly help system analysts and programmers. DBAs *must* be part of the production team that handles change requests; they can provide accurate information on the amount of work involved in making alterations in database structure or altering functional parameters. The DBAs *should* be part of the quality assurance function, at least in so far as they can help in the design of testing procedures. It's unlikely that the DBA would have the time available to actually run extensive tests as part of the SQA team. DBAs can contribute their insights to the computer security incident response team, again with an eye to providing accurate, realistic information about database usage, number of users, critical systems depending on the databases, tolerance for downtime in specific cases, and so on.

40. Keeping accurate documents is part of the DBA's job. DBAs must collaborate with the version-control group to ensure that every production version of the database is fully documented; there must never be any doubt about exactly which change was implemented – and why – in any specific version of the database. Good documentation also helps new staff members come up to speed quickly.

This slide reminds me of an experience I had in the late 1980s when I was doing a whole year of database performance analyses for the government of Canada – 12 systems in 12 months. In one

# MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

## *Introduction to DBMS Admin & Security*

---

case, having been told that the system was running slowly and had been getting worse, I asked for the database layouts and the system documentation, including the source code. To my astonishment, the administrators said sheepishly that there was no record of the source code! I couldn't believe my ears. Apparently, the database had been built by a vendor who claimed to have turned over all of the system documentation and source code on a magnetic tape seven years before; however, nobody could find the tape and the vendor claimed to have deleted all records of the system they had sold the government department. What a mess. It took a great deal of work to reconstruct the system design through reverse engineering and black-box data collection. Don't let this happen to you!

A different kind of documentation is the *log files*. These records of activity in the database are not only required by law to whatever extent the regulatory environment dictates; they are also invaluable for statistical analysis supporting effective capacity planning and maintenance of *service level agreements* or SLAs. We'll talk about capacity planning and statistics at the end of the course.

41. At this point, we get to look in some detail at methods for controlling processes that access databases concurrently. I should remind you that the classic definition of the *process* is *the unique execution of a particular piece of code at a particular time by a particular user on a particular processor*. Two users running the same program at the same time have generated at least two concurrent processes. A user doing online transaction processing in a database and a batch program writing a global report from that database has generated at least two concurrent processes. And in the following slides we will look at methods for preventing the concurrency problems that you saw in earlier slides as well as learning about additional problems.

42. A fundamental concept in database theory is the *transaction*, which is *a set of operations, all of which must be completed for a database to return to a consistent state*. An obvious example of transactions is an order-entry system where it is quite common that people include the total number and cost of the line items in the order header. That information will be updated at the end of entering each of the details. This is an example of how a *non-normalized* design, one that is actually repeating information, can sometimes be used to provide faster reporting than if we had to read all the details every time we run a report on orders.

So consider the exact sequence as an order-entry operator enters three records and the system crashes after the third entry but before the order-header has been updated to take into account that third record. So what are the values in the order header's total fields? You can understand that the order header has not yet been updated with that third record information and that we have what is called an inconsistent state: the totals in the header correspond to two, not three, records and are missing the information from the third record. Even though three of the order detail records have been put into the database, which you could think represents the order, the fact that the system crashed before the final calculation was written into the order header makes the transaction incomplete. But how would we know? As far as the user is concerned everything went well. It's almost impossible to know just by timing whether the final calculation was done;

# MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

## *Introduction to DBMS Admin & Security*

---

you would have to look at the order header and do the arithmetic to see that there is an inconsistency.

This is a good time to mention that database applications normally include diagnostic routines that scan the data for compliance with the logical requirements imposed by the system designers. In this case, the diagnostic program would total the information from each order and compare it with the total recorded in the order header; the program then records the error.

Do you think the utility should also repair the error?

I don't: unless a human operator verifies that the data in the order-detail are complete and correct, how is the program to "know" whether the total should be corrected or whether the entire order is corrupted? Typically, such interrupted transactions are actually removed from the database using rollback or roll-forward recovery, which we will discuss later.

43. The problem we just looked at illustrates the importance of defining atomic transactions in database management systems. We want everything in a transaction to complete or none of the intermediate steps to be left in the database. Theorists have referred to this principle as a demand for *atomic transactions*.

I couldn't help indulging my passion for etymology by pointing out the origin of the word *atomic*. Other words with this root that you may have come across include tomography as in CAT scans – *computerized axial tomography* – where tomography literally comes from slice-writing.

But anyway, in modern DBMSs, programmers defining atomic transactions use some definition of the start and a marker to show the end; normally the end is known as a *commit* marker or record. These tools permit what is called *rollback* should the transaction be interrupted. We'll come back to the types of recovery later in this lecture.

44. One of the applications of atomic transactions is what we called the *locking strategy*. Now, the concepts described in this section apply equally well to other aspects of programming and operating systems; they're not limited to databases. So let's move straight into the basic concepts of locking on the next slide.

45. Locking is an element of *interprocess communication*, or IPC. A lock is one form of what operating system theorists call a *semaphore*; semaphores can be *binary* – that is having exactly 2 states – or they may have more than two states. Specifically, a lock is used to prevent simultaneous access to a critical resource; the effect of locking and checking locks is to *serialize* access to the resource. *Serializing* means that concurrent processes have to wait in line, much like people in a crowd might have to line up at a single turnstile to enter a building one by one. In a database, the *resource* can be a single record, a group of records in a single dataset, or even records spanning multiple datasets. Think back to our example of the order header that has to be updated as a result of changes in the details; imagine if two people accidentally began adding

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

information to the same order. What a mess! Using the techniques that we'll be discussing in a minute, one could lock order #12345 so that the second person trying to update order #12345 would have to wait until all the operations were complete from the first operator. That way, we wouldn't end up with multiple, repeated order lines and distorted order totals in the header.

46. There's a number of concepts used in detailed discussion of locks.

For example, locks can be set *implicitly* by the DBMS or they can be started and stopped *explicitly* because the programmer asks for them.

The *granularity* of the lock refers to how much of the database is locked; typically and ideally, one locks as few records as possible to prevent performance problems. You understand that if a program locks a large number of records or an entire dataset or, heaven forbid, the entire database, access by large numbers of users is going to be significantly slowed. If you normally have 1,000 people working on 1,000 different orders at the same time because the database locks each order individually, it might take a few seconds on a modern mainframe to handle all of the orders simultaneously. There is no conflict. However, if the locking strategy were to, for example, lock the entire order dataset it would serialize access. Let's suppose it takes, say, 2 seconds to complete each transaction but you can only do one transaction at a time, then 1,000 concurrent transactions would end up being done over 2,000 seconds. Since an hour has 3,600 seconds, we're talking more than half an hour here. Many of the performance problems that I solved back when I was doing database performance optimization turned out to be due to bad locking strategies.

Another useful distinction among locks is its *exclusivity*. We call a lock exclusive when the process which obtains the lock is the only one that can read and write using the locked resource. In a *shared* lock, although one and only one process has both *read and write* capability, other processes may be able to *read* from the resource. There are questions about what happens when those reading processes access records that are being modified; we'll discuss those questions in the section on access modes and cursors later on in the lecture.

47. The next attribute of a lock that makes a big difference to performance is whether the lock is *conditional* or *unconditional*. Now, the word *conditional* in this context has a special meaning. It refers to whether any condition or status is returned to the calling program if the lock cannot be obtained. Whenever a lock of either type is obtained, a condition is returned to the calling program and execution continues. The question is what happens when the lock can't be implemented at once because the resource – for example some specific record – is locked?

In conditional locking, as you can see on the side, control returns to the calling program if the resource is already locked. At that point the program logic determines what happens next. Normally, one programs the logic to loop back, at least a certain number of times, to see if it's possible to obtain the lock in a reasonable interval. If not, the program can return a report to the user and let the user decide what to do. For example, the user might receive a message that says,

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

“Someone else is updating this client record; would you like to continue waiting or would you like to do something else?”

In contrast, unconditional locking, which is generally viewed as more dangerous and therefore to be used with great care – does *not* return control to the program if it cannot be granted immediately. Once the program puts in a request to lock a resource, the process is put in a *lock queue* and the program is suspended until the lock is granted.

The analogy for conditional locking that comes to mind is the difference between waiting to be served by a clerk in a normal store, where you can decide whether to leave the queue or not. If the queue looks long, you can always go do something else. But in the store from hell, you might have to enter a locked room to be able to see if the clerk were busy; once in the room, you would be forced to wait until the customers ahead of you were served before you could get out.

Unconditional locking reminds me of that nightmare scenario.

As you will see in the next few slides, unconditional locks in combination with bad programming can lead to *deadlocks*.

48. Let’s watch the sequence that creates a *deadlock*, also known as a *deadly embrace*. Starting at around 10:03 in the morning, Process #1 locks Resource A. Immediately afterwards Process #2 locks Resource B. The next step is that Process #1 now locks Resource B – and this is very important – *unconditionally*. Then Process #2 locks Resource A unconditionally. So you see what’s happened: Process #1 is waiting for Process #2 to release Resource B; however Process #2 can’t release Resource B until Process #1 releases Resource A – which it isn’t going to do, ever. The only way out is to abort one of the processes.

49. What you have just seen is an example of a *race condition*: an unexpected problem that takes place only under specific conditions of timing. For example, in the previous slide, if process #1 obtains both locks before process #2 tries to lock anything, there won’t be a deadlock. In general, a simple rule for avoiding deadlocks is that you will always have all processes lock resources in exactly the same order and you must unlock the resources in the reverse order of what you lock them in. You may want to take a look at the previous slide and see how these rules would have prevented the deadlock. Enforcing this rule requires coordination among programmers; a global list showing the priority order of resources in the database is a common tool to help everyone follow the sequence properly. Another approach is simply to rely on the DBMS or the operating system to prevent deadlocks by enforcing limitations on locking and unlocking, as discussed in the next slide.

50. Another locking strategy in addition to the sequence of locking discussed on the previous slide is called *two-phase locking*. In this approach, the *growing* phase of the lock allows any process to lock additional resources such as records or record sets. However, as soon as any one resource is unlocked, the process is prevented from acquiring any further locks until all of them are released by that process. If you think about it you’ll see that this approach prevents

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

transactions which affect the same records from ever overlapping. However, the more common strategy for serializing transactions is simply to state that *none of the locks can be released at all* until either the entire transaction is *committed* – that is, written to disk – or *dropped* – called a rollback instruction.

51. In the early days of programming, designers adopted what came to be known as a *pessimistic locking strategy*. Their programs were based on the notion that collisions were bound to occur and therefore programs should use restrictive locking to prevent any concurrent access at all. Each process would lock all the records involved, do its stuff, and then release the locks. Sounds great, right? Yeah, until the data entry operator who had locked some critical resources went away for lunch leaving the resources locked. Naturally, the next person who tried to lock whatever was locked – the dataset for example – would have their process hang until the first user came back and finished the original transaction and unlocked the resource. Sometimes, you could get dozens or hundreds of people whose sessions were simply hung during lunchtime until one o'clock, when everybody would suddenly start getting access one after another once the original operator completed the first transaction that locked the resources.

52. That kind of problem is why programmers in the 1970s and 1980s eventually settled on the nearly universal *optimistic locking strategy* of today. In this design, first of all we lock as little as we can; that is, we don't lock the dataset if we can lock only a few records. Second, we don't lock until the last possible moment. First, we read the records that have to be modified and store the original values in memory buffers (that is, variables). Then we work in memory buffers to prepare the changes. Only then do we lock the original data records and compare their current values to the values that we stored in the memory buffers; if the original values and the current values are the same, we can go ahead and commit the transaction. That is, we write the new values to disk and unlock the records.

In contrast, if there have been changes in the records since the last time we looked, then we have to unlock the records at once and go back to the user to find out what to do. For example, the program can show the user the current state of the records and ask whether the user wants to apply the same changes to the current value of the records. A particular instance might involve an inventory record the user initially receives with information that there are 20 widgets in stock and so he arranges to take out five. By the time the application program is ready to initiate the transaction, someone else has taken out 10 widgets, leaving 10 in stock. The program can therefore ask the user something like, "There are now only 10 widgets left; do you still want to withdraw five?" If the user says yes, the program locks and reads the record again; perhaps this time there are still 10 widgets in stock, so the transaction proceeds, correctly leaving five widgets in stock.

53. So looking at optimistic locking overall, it's particularly appropriate for the backend databases that are supporting Web applications, where there are potentially many people accessing the database, all of whom are completely out of the control of the designer. I suppose

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

it's possible that you could use pessimistic locking around user interventions with highly trained staff, but it beggars the imagination that you could get away with that with ordinary people buying stuff through the Web. In addition, the slide points out that optimistic locking becomes even more advantageous if the lock granularity increases. The more resources you are locking at once, the less you want to lock around human intervention.

On the other hand, if you happen to be locking something where there is enormous contention, optimistic locking, with its cycle of going back to read the record as required, runs the risk of repeated occurrences where the resource will have changed by the time the user tries to commit the transaction. One word for that kind of activity is *thrashing*.

54. Okay, so just one final note about locking: back when I was teaching database internals 30 years ago, the 3GL programs in use typically called DBMS routines; for example, in the IMAGE/3000 DBMS running on the HP3000, we called the routines database *intrinsic*s. The programmer would call DBLOCK and pass the appropriate parameters for all of the details that we have been discussing in the past few slides such as the object of the lock (implying its granularity), its conditionality, and any other attributes desired. In contrast, today's 4GL programmers use much more convenient instructions that handle the locking strategy automatically.

55. I don't want to spend a lot of time on this next section about *ACID transactions*, but I do think you should be familiar with the acronym and the meaning of the components.

56. We've already discussed *atomicity*, and this reminds us that transactions are defined as *atomic* changes. We don't want parts of them; it's all or none.

57. The concept of *consistency* helps us define what should constitute an atomic transaction. The database always has to be consistent at the completion of the transaction. For example, to take a trivial case, if we are adding a record to a detail chain – that is, adding a record that shares a key with other records, such as the detail lines of an order – the DBMS naturally has to update information about how many records there are in the chain of order details. Regardless of any explicit intentions of the DBMS programmer, that kind of change is always included in the definition of an ACID transaction. When many different datasets and records have to be changed to keep the database consistent, we often turn to *batch processing* so that we can lock the entire database and keep everyone else out of it to prevent any interference and consequent inconsistency while the changes are being made.

58. The next component of the ACID tetrad is *isolation*, which refers to the degree of protection against having other processes reading *intermediate results* of transactions.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

For example, if Alice is busy making a correction to the address of a customer, it would be a mistake for Bob's process to access the record before Alice's process confirms that it's correct and permanent. That's called a *dirty read*.

Then there's the case where Bob reads the record showing the widget inventory level and gets ready to subtract some widgets while Alice's process cheerfully changes the record behind Bob's back. You will recall that that situation is exactly what we're discussing as the reason for locking and that it can very well occur during optimistic locking.

Then we have the odd situation where Charlie has run a query to calculate the accounts payable while Dave has been adding new outstanding bills at the same time. Moments after Charlie's first query, he runs a related query expecting to use the same data for some other calculation and presto! The results are inconsistent.

59. ANSI SQL provides specific types of access modes defined as isolation levels to help resolve these problems.

60. One of the ways that isolation is implemented in ANSI SQL is the definition of a *cursor*. Instead of forcing access to the entire set of records qualified by a SELECT statement, the cursor allows the program to access one record at a time. In the next few slides, will look at the different types of cursors available. Keep in mind that I am not trying to make you into programmers; however, if you ever encounter questions of data integrity in databases, one of the issues you should raise as you go through your diagnostic procedures is definitely problems of concurrency and mistakes in the definition of cursors.

Not all integrity problems are due to hackers; indeed, while I think of it, I can tell you about an amusing incident which has nothing to do with cursors but which does illustrate the point that sometimes a banal factor involving a database can give the illusion (even if only momentarily) of a more serious problem. The problem came to my notice when an operator reported to me sometime around 1985 that there must be a problem with a disk drive. "Oh, yes?" said I. "And why do you think that?" Well, he answered, there was garbage in some records in one of the databases. I asked if the damage was localized to a specific database; yes, it was a big insurance policy database. You will understand that this answer immediately precluded there being a hardware problem; there is no way that a disk drive would limit damage to a specific database. Was it localized to specific fields? Yes – the garbage was only in the free-format comment field. "Ah," I said, "and what exactly is this garbage?" It turned out to be stuff like "Joclyn, please meet me for lunch." By now, you have probably guessed what was going on: the data entry operators had discovered that if they created fake insurance policies (they were at the top end of the range with policy numbers like 99999 and 99998), they could assign one fake policy per operator, write their messages in the comments field, and use this system as a crude form of e-mail. Perhaps it will interest you that I resolved this problem not by threatening or firing the employees but by actually providing them with a freeware e-mail system and politely instructing them no longer to use their crude tampering with the insurance policies.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

61. Without belaboring the point, this and the following slides briefly summarize some of the attributes of ANSI-compliant SQL *cursors*.

62. Cursors can have a performance implication, so programmers should be careful to use the cursors that provide just enough capability for their specific needs. We'll look at the four types of cursors listed here in the following slides.

63. The simplest cursor moves through the record set like the current-record pointer in an ordinary sequential-file access mode. Only changes ahead of the *forward-only cursor* in the record set defined by the SELECT clause for this cursor can be seen by this access method.

64. If one is trying to do a number of calculations or sorts or doing extractions that are okay even if they deal with a static image of the database at a particular time, then the *static cursor* is ideal.

65. A slightly more interactive cursor actually reads records from the database using the key values defined in the initial snapshot that corresponds to the data that a static cursor would have gotten. Instead of using only the old data from the snapshot (as a static cursor would) the *keyset cursor* uses the key value from the next record in the record set to query the actual database at that moment and puts the current value of the record into the record set. It's a kind of self-updating static cursor – maybe a not-so-static cursor.

What happens if one of the records that used to be there at the time of the initial snapshot has been deleted? The keyset cursor regenerates the missing record. This may be good or it may be bad; the programmer has to be careful about choosing the right type of cursor for the right application.

Again, from a security standpoint, the point of bringing this up to you is to alert you to software design problems that may have consequences easily misinterpreted by amateurs as evidence of hacking or other unauthorized tampering with the data.

66. The most interactive cursor of all is the *dynamic cursor*, which as you can see on the slide is basically like reading the file all the time; the exception is that one can set the isolation level to preclude access to changes made by other users until they have been committed. The exception is the *dirty read*, which allows dangerous access to possibly retractable changes.

67. So as you can see, programmers have a choice of cursors and you as security experts should be on guard whenever you are dealing with database problems that look superficially like security violations. Keep these programming issues in your open mind as you do your

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

investigations and, as Sherlock Holmes said, don't hypothesize in advance of the facts. "It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts." [Doyle, A. C. (1892). A Scandal in Bohemia. *The Strand Magazine*. Scanned edition available online. < <http://www.artintheblood.com/scan2/scanintro.htm> >]

68. Finally, returning to our four components of ACID transaction theory, *durability* is simply the requirement that changes we make in the database must either be permanent once they are complete or must be removed if they aren't. And we will come back to durability when we discuss logging and transaction rollback in a bit more detail in just a few slides from now.

69. Hard to believe, isn't it? We are actually reaching the end of this humongous lecture. In this last section, I want to address database security and resource management explicitly.

70. We'll start with database security.

71. At the highest level of access control, the question is who gets to do what to which records? This function is known as *authorization*.

Database administrators, working in conjunction with programmers, might modify the database structure; DBAs usually grant specific rights to particular users or categories of users.

72. I won't belabor the point about *identification and authentication*; it's enough to remind everyone that there are four types of authentication, as shown on the slide.

73. Although individual users should not be sharing their user IDs and authentication markers, it is possible that groups of users may be defined with identical privileges. *Role-based security* allows effective management of such groups.

74. General database security may be adequate for most cases, but there's no guarantee of universality. *Business rules* may require application level programming to reflect special, changing security requirements in different functional areas.

75. What happens if the system does crash in the *middle* of a transaction? In this section, we'll finish our tour of database management and security with a quick look at *recovery* options.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

76. Databases are so completely integrated into today's IT that they play a role in innumerable systems that can affect safety, mission-critical operations, financial and regulatory compliance, and national security. So I think it's safe to say that every critical database must absolutely include effective *recovery mechanisms* as an essential aspect of security.

77. *Application logging* provides the fundamental audit trail that will allow all effective recovery of a database after an unexpected interruption – whether that is an application failure or system crash. But log files serve many other functions. They can be used in forensic investigations, for performance optimization, and for system debugging. On centralized systems, they can even be used to allocate costs as a function of resource utilization – the old *charge-back* systems that users hated. Logging processes vary in what they store; for example,

- All logging processes store an image of every record *added* to a database along with timestamps and user information;
- However, some logging processes keep a copy of the old version of the record as well as a copy of the new version of a *changed* record whereas other logging processes may skip copying the original version;
- Some logging systems will keep an image of a *deleted* record whereas others may keep only a record number for the deleted data along with the timestamp and user information.

Thus it follows that security personnel will want to know in advance of a problem precisely what kinds of records are being kept in the log files and may want to discuss altering the details of what gets logged with the DBAs if necessary.

With today's inexpensive disk storage, the volume of logs is not usually an issue; however, sometimes there are performance questions about data transfers through networks for large log files. All discussions of logging also have to remember to discuss how long to store (archive) the logs.

From a security standpoint, remember to discuss how the log files will be safeguarded against tampering; common methods include adding hash totals to each record, chaining the hash totals by including the hash from the previous record in the computation of the hash for the current record, and using digital signatures on individual records and for whole log files.

78. A log file that simply showed all of the changes in the database wouldn't be much good for recovery, however. The log file needs to mark the beginning of a transaction and then to mark the end of that particular transaction. That way, a recovery program reading the log file can easily distinguish a completed transaction – one with a start and its corresponding end – from a damaged, interrupted transaction which is missing its end marker.

79. For recovery to work, there have to be *backups*. Backups can cover the entire system, parts of the system or only a specific application. Regardless of the target, a backup can everything in existence for that set, or it can look at everything that has changed since the last full backup

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

(that's a differential backup), or it can copy everything that has changed since the last incremental backup. A delta backup is even more specific: instead of copying files that have changes in them, it copies only the changed data from those files. Delta backups are always application-specific. Log files are really variable: depending on the specific application, they may include only metadata whereas in global database log files, they can include copies of specific records as well, as discussed in previous slides.

80. This chart shows what might be included in different types of backup; the red rectangles represent days when specific files have been changed. The prime markers in the *delta* row are there to suggest how the delta backup includes only changed *records* from the files indicated.

81. So now here's an exercise for you. Go back to the chart on slide 80 and imagine that you have started to work on Wednesday morning when the system suffers a catastrophic failure. You have to put the system back to where it was Tuesday night at the time the last backup was taken. What do you have to do if you are using different types of backups? You think about this and interrupt the playback using the *escape* key. Then you can start the show over again at this slide by clicking on the little *slideshow* icon at the bottom of the screen and pick up from where you interrupted the commentary.

Okay so now let's go through it step by step.

- If you are taking a *full backup* every night, you just restore Tuesday night's full backup and you're done.
- If you are depending on a *daily differential* backup, then you have to restore Sunday night's full backup and Tuesday night's differential backup. Then you have to delete files that were deleted between Sunday and Tuesday night.
- If you are using *incremental* backups, then you have to restore Sunday night's full backup, restore Monday night's incremental backup, and restore Tuesday night's incremental backup. Then you have to delete files that were deleted between Sunday and Tuesday night.
- If you are using *delta* backups, you have to restore Sunday night's full backup and run a special application program to read the Monday night and Tuesday night deltas. Then you have to delete files that were deleted between Sunday and Tuesday night.

82. But what about using log files if it's only the *application* that has crashed? In that case you can do a *rollback* from the current state by reading the log file to find all of the interrupted transactions and getting rid of them. Alternatively, you can restore a known good backup copy of the database that has been synchronized with the start of your log file or log files and then run through the log file reapplying every completed transaction. In either case you end up with a consistent database. That latter is a *roll-forward* recovery.

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

Unfortunately it's not possible in theory to state that one type of recovery is necessarily faster than the other; as my notes on the slide point out that would depend on how many operations there are in each type of recovery. However in practice, rollback recovery is almost always faster than roll-forward recovery because there are usually relatively few transactions that have been broken because they were in progress when the application system crashed. Roll-forward recovery is used primarily when a system crash is so severe that there is question about the integrity of the entire database and you don't want to risk simply removing incomplete transactions.

83. I have only a few more points to make in this lecture. Remembering that the *Parkerian Hexad* includes *utility*, I've chosen to include some database management issues that are not usually thought of as security concerns but which I think ought to be. In particular, you should be aware that the bias towards assuming that all problems are likely to be due to malicious software may lead you down the wrong branch of the problem-solving tree. Poor resource management can cause performance problems and system breakdowns in the absence of malice.

84. I've already mentioned that log files have many functions; here, we see a few more.

One application of log file analysis is to identify users who have to make a lot of corrections to their inputs. I'm not suggesting that this be part of a punitive management style; on the contrary, it seems to me that an appropriate response is to provide additional training for the users.

An alternative explanation for high error rates is that the user interface (UI) may be flawed. UI problems and other design flaws and errors may account for high error rates even if different users adapt to the problems differently. Yes, perhaps Martha knows how to get around the problems and therefore has fewer errors than Bob – but that doesn't mean we shouldn't be analyzing the root causes of the problems and fixing them if possible.

Another application is performance analysis. We can study transaction volumes to adjust staffing levels; we can look at response times from a number of different angles and take appropriate action based on what we learn.

In the next slide, will look at the importance of spotting dramatic changes in slope, called inflection points.

85. I'm sure that you have either created or seen graphs of particular resources, transaction rates, or response times and noticed that some of the curves occasionally show sharp changes. For example, the blue line shows some phenomenon that's been going along at one level for a while and then at time A suddenly rises to a new plateau. Why? Similarly the thick green line has been rising at a modest rate for a while and then suddenly at point B the growth rate takes off. Why?

One of the specific cases from my own experience that comes to mind concerns disk space utilization at the data center I was running in the mid-1980s. I was doing my usual monitoring of resource utilization and turned to disk space for the month. One of our clients showed an unusual

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

spike in the rate of growth of disk space utilization starting about two weeks earlier. It was quite striking, and at the rate they were going, they would use up an entire BFD (big fixed disk) – a \$75,000 investment in today's money – within a couple of months. What was going on? A quick investigation of the files in their directory showed thousands of files with the name TMPnnnn (e.g., TMP1234) – temporary files generated during their batch jobs that would normally be deleted at the end of each job. Investigation of their JCL (Job Control Language) showed that a programmer had REMmed (disabled by converting into a comment) the PURGE (deletion) command for the temp files while running debugging operations two weeks before. He had forgotten to remove the REM. Had we not had a regular procedure for monitoring resource utilization and looking for exceptions, we might have run out of disk space for everybody on the system without warning because of that glitch.

86. It's not just inflection points that should be of concern. Even normal growth in resource utilization must be monitored and projected. For databases, disk space is a critical resource. In particular, database files must not fill up; unless your DBMS can allocate disk space dynamically, the DBA must monitor disk space utilization within all datasets to ensure that no file will ever reach saturation. Not only will a full dataset stop the processing in the DBMS, even a dataset *approaching* saturation can severely degrade performance for all changes including the addition of new records and modification of existing data.

87. Use statistical methodology when projecting saturation of critical resources, regardless of what the details are. Whether it's disk space, processor capacity, memory requirements, personnel needs – take advantage of well-established linear and nonlinear regression methods to compute upper and lower confidence bounds for your projections. Don't give your managers a point estimate (T in our graph on the slide) for the time you will reach saturation (S): include worst-case ( $T_U$ ) and best-case ( $T_L$ ) estimates with a known probability (e.g., 95% confidence bounds) as well based on the variability of the data and the reliability of the projections.

88. One final note on performance: you should realize that the way records are physically stored in a dataset can profoundly influence the speed with which they are retrieved. I've mentioned before that database designers have to think about which keys to include in detail datasets. A typical example would be how we store the order details (each line of an order) in an order-detail dataset. Typically, the primary key for such a dataset would be the order number because it's so frequent that we need all of the lines for a specific order.

Now, normally, nobody really cares about where the records are in a dataset; after all, that's what the DBMS handles for us. We just ask for records and get 'em on demand. Who cares where they are?

But over time, in our order-processing example, as orders are deleted or modified, new records are added to the dataset in the first available free position in the physical dataset, putting order lines out of sequence in the dataset. The colored rectangles on the left of the slide symbolize a

## MSIA GI512 SEMINAR 1 WEEK 4 LECTURE 2

### *Introduction to DBMS Admin & Security*

---

mixed up dataset where each color represents a single order and the little rectangles represent the individual lines in each order. The thick black rectangles represent the unit of physical I/O called a block. The DBMS always reads an entire block at a time. You can see that if you try to read all the records for a specific order, you may end up having to read the same blocks over and over because the individual records are scattered across multiple blocks. In contrast, on the right-hand side, the orderly progression of records represents a packed dataset. You would need only two I/Os to read all the records in this particular collection of order details.

So once again, be aware that reduction in availability need not necessarily imply malicious software or theft of resources by hackers: it could be the consequence of normal wear-and-tear on a database and it may respond to database maintenance and optimization.

89. Finally, in our last substantive slide of this lecture, I just want to point out that databases cannot be static. Inevitably, DBAs have to participate in constant evolution of the information infrastructure to meet the strategic goals of the organization. Whether that means restructuring the existing database or participating in the reengineering process using new tools, DBAs and security personnel need to keep track of the changing environment in which they work for the good of their colleagues.

90. Congratulations: you have made it to the end of this introduction to the issues of security and database management. I hope that you have been stimulated into thinking about your own case study, whether it is organizational or industry based, and that you are taking away some practical ideas that will help you in your current work and in your career. All the best to you. Bye for now.

