CS253:HACD.2 Temporal Data and The Relational Model

Notes keyed to slides

1. Cover slide

The chapter in Date first appeared in his 7th edition, as Chapter 22, but the chapter was quite heavily revised for the 8th edition.

There is an unfortunate typographical error on page 744. In the first bulleted paragraph ("The expanded form ..."), delete the last three words, "defined as follows:".

2. Temporal Data and The Relational Model

In particular, none of the leading SQL vendors (IBM, Oracle, Microsoft, Sybase ...) have implemented SQL extensions to solve the problems we describe.

There was significant interest for a time in the second half of the 1990s, when an incomplete working draft for an international standard for such extensions was produced by the SQL standards committee. However, the project was abandoned when support for XML documents in SQL databases became a higher priority to the industry than temporal extensions.

(Some people question the industry's priorities!)

MighTyD

In the academic year 2005-6 a team of computer science undergraduates at Warwick University, for their final-year project, made the beginnings of a prototype implementation of **Tutorial D** with some of the temporal extensions taught in CS253. They call their product **MighTyD**. Like **Rel,** it is written in Java. It is available for free download at http://sourceforge.net/projects/mightyd.

*If by any chance you might be interested in continuing the work on **MighTyD**, please arrange a meeting with me to discuss ideas.*

3. The Book's Aims

Imagine, for example, a database from which nothing is ever deleted and in which every record is somehow timestamped to show the time at which it arrived and, if its information is no longer current, the time at which it was superseded or deleted. The interval between those two times is an interval throughout which the record was "valid" (i.e., represented a held belief).

Such a database might be called a temporal database, but there is no precise definition of that term, nor do we really need one.

The records in a temporal database don't have to be exclusively about the past and present. They could be about the future, too (e.g., employees' planned vacations, project schedules etc.), though beliefs about the future are usually subject to more uncertainty than those about the past and present.

4. Contents (Parts I and II)

This section of CS253 broadly follows the structure of the book, but we assume you have retained a grasp of relational concepts and the **Tutorial D** subset that you were taught in the second year.

The official definition of **Tutorial D** is in *Databases, Types, and The Relational Model*: *The Third Manifesto*, by C.J. Date and Hugh Darwen (Addision-Wesley, 2006, ISBN: 0-201-70928-7). The relevant chapters are available in softcopy (PDF) at http://www.thethirdmanifesto.com.

5. Contents (Part III)

Regarding database design, constraints, queries and updates, you will be shown the complexity of the problems that need to be solved, and proposed solutions to those problems. You should familiarise yourself with the solutions and be able to describe in broad outline some of the problems addressed by those solutions. You do not need to learn the complicated **Tutorial D** expressions for the longhand expansions of the proposed new shorthands!

The topics of Chapters 15 and 16 are not included in CS253.

6. Appendixes

None of these topics is included in CS253.

7. Part I: Preliminaries

Recall the careful distinction we make between ***values*** (relation values in particular) and ***variables*** (relation variables in particular).

We normally abbreviate "relation variable" to ***relvar***. The SQL counterpart, roughly speaking, is the base table, though strictly speaking this corresponds to what we call ***real (or base) relvars*** in particular. (Our counterpart of the SQL updateable view is the ***virtual relvar***.)

8. Chapter 3: Time and the Database

Quantisation is the key. Although most people intuitively think of time as continuous, we consider a time interval to be a set of discrete, equally spaced points. The "distance" between adjacent points is according to a chosen **scale**. In all our examples the scale is one day unless otherwise stated (explicitly or implicitly).

Quantisation has the huge advantage of making an interval correspond to a ***finite*** set of points. Computers are much better at dealing with finite sets than infinite ones and the Relational Model is explicitly based on finite relations only.

"Valid time" and **"transaction time"** are rather inappropriate and unintuitive terms in widespread use in the temporal database community. The valid time of a record refers to all the times at which the proposition it represents is held to be true. The transaction time of a record refers to all the times at which it was or is "in the database". **We do not pursue these concepts on CS253.**

9. CHAPTER 4: What is The Problem?

We introduce you to the special problems that would be addressed by temporal databases if a DBMS were available to support the concept.

10. Example: Current State Only

This is what you might call a nontemporal database. We use it as a starting point from which we will develop, in three stages, its fully temporal counterpart.

The queries we can make on this database have temporal counterparts too, and so do the constraints we would like to declare, and so do the update operations we would like to be able to perform—as we shall see as the section unfolds.

11. "Semitemporalising"

"Semitemporalising" because we are doing only half the job, so to speak. Actually, rather less than half.

Although such "since" relvars are inadequate of themselves, we shall see (much later) that they do have part to play in a fully temporal database.

The notation *dnn* for a day number is used for convenience. In real life we would normally expect to see a date, such as 2004-03-01.

For each proposition (represented by a tuple), we have a "since" value indicating the day on which the proposition in question first became true. It is assumed still to be true at the present time. We have no record of similar propositions that used to be true in the past but are no longer true. Thus, this is still a "current state" database.

And of course the existing technology can easily handle such databases. Well, comparatively easily, anyway. But observe that SQL, for example, has no shorthand for expressing the constraint to the effect that the SINCE value in an SP tuple had better not be earlier than the SINCE value in the corresponding S tuple (for a supplier cannot be able to supply anything while not under contract).

Exercise: Write a **Tutorial D** or SQL expression for the constraint just described. Recall that in **Tutorial D** you can use IS_EMPTY (*rel expr*) to express a constraint to the effect that the result of evaluating *rel expr* (an expression in **Tutorial D**'s relational algebra) must at all times be empty.

12. "Fully temporalising" (try 1)

Now we have the times at which true propositions ceased to be true as well as the times at which they started to be true. And that means we have a historical record as well as a record of the current state of affairs (assuming that today is day 10, so every tuple whose TO value is *d10* represents a current state—an interim and inadequate solution to a difficult problem we will return to later).

By "very difficult", we mean so difficult that we won't even show how it might be done! But those queries are not impossible and you are welcome to have a try (in **Tutorial D** or SQL). In each case, the result should not show two or more tuples for the same supplier whose FROM-TO intervals overlap in time or are such that one immediately follows the other in time.

Notice "try 1". Although this representation can be achieved with existing technology, it is not really very suitable. When working with intervals, we sometimes want the end points to be considered as included, sometimes not. For example, how does the system know whether S2 was under contract on day 4, or whether day 4 was actually the first day on which S2 ceased to be under contract? Soon we will introduce "try 2" as a better solution, overcoming this problem.

13. Required Constraints

Again, you are welcome to try to express these constraints in **Tutorial D** or SQL.

14. CHAPTER 5: Intervals

We introduce the concept of ***interval types***. An interval type is a type whose values are intervals. An interval defines a line segment in terms of its end points. For temporal databases we are particularly interested in intervals on the time line, such as the interval from October 2nd, 2006 to December 8th, 2006.

CAVEAT: SQL uses the term *interval* (and the key word INTERVAL) for what should really be called a length or duration, and it uses it exclusively for durations in time, such as 5 minutes, 2 days, 3 nanoseconds. Our use of the term is according to normal mathematical convention.

15. "Fully temporalising" (try 2)

Using an interval type as the declared type of an attribute (DURING), we put both end points together in a single column, so to speak. A square bracket before the begin point or after the end point indicates that that point is included in the interval. But we don't actually store the brackets (or the colons)! The next slide explains.

Of all the values whose type is DATE, there is one for which NEXT_DATE (*d*) is undefined. And that is the value representing the date of the "end of time".

Similarly, there is one value for which PRIOR_DATE (*d*)—the inverse of NEXT_DATE (*d*)—is undefined: the date of the "beginning of time".

For CS253, as in most of the book, we concentrate on point types like this, in which there is a first value and a last value. Note, however, that such types cannot be used for intervals over, for example, days of the week or times of day. These require so-called ***cyclic*** point types, which have some rather interesting properties and are described in Chapter 16. **You are not required to study cyclic point types.**

16. CHAPTER 6: Operators on Intervals

Just as there are some operators that are defined for all relation types (e.g., JOIN, projection, IS_EMPTY), there are also some that are defined for all interval types. It is the possession of these operators that characterises an interval type.

Our main interest is in the (read-only) operators that are defined to operate on interval values. As well as these, we need expressions to denote such values, for which purpose we use special read-only operators called *selectors*.

17. Interval Selectors

A **selector** *S* for type *t* is an operator that, when invoked, returns a value of type *t*. None of the arguments to the invocation can be of type *t*. For every value *v* of type *t* there is some invocation of *S* that returns *v*.

A **literal** is a special kind of selector invocation. You can think of the literal 12 as being an invocation of an operator that operates on a given sequence of decimal digits and yields the integer indicated by that sequence. The invocations of INTERVAL_INTEGER shown in this slide are literals because in each case both arguments to the invocation are themselves literals.

Notice how there are four different ways of selecting the interval that runs from 1 to 10 inclusive.

Exercise: How many ways are there of selecting the interval that runs from the beginning of time to the end of time?

18. Monadic Operators on Intervals

Exercise: What is the result (TRUE or FALSE) of the following **Tutorial D** expression?

```
WITH INTERVAL_INTEGER ( [3:7) ) AS i1 :
COUNT ( INTERVAL_INTEGER ( ( PRE ( i1 ) : POST ( i1 ) ] ) =
COUNT ( i1 )
```

19. Comparisons of Two Intervals

Our decision to use mathematical symbols like ⊆ in **Tutorial D** has turned out to be questionable. Implementations such as **Rel** and **MighTyD** support alternatives that are available on a normal keyboard; <= for ⊆, >= for ⊇, and so on.

20. Some Pictorial Definitions

This slide uses colour, so the monochrome rendering of it in the handouts is useless. But meanings of these operators should obvious to you from their names.

Points to note:

- MEETS is commutative (*i1* MEETS *i2* ≡ *i2* MEETS *i1*).
- If *i1* MEETS *i2,* then either *i1* SUCCEEDS *i2* or *i1* PRECEDES *i2.*
- OVERLAPS is commutative. *i1* OVERLAPS *i2* if and only if the operands have at least one point in common.
- MERGES, not defined by Allen, turns out to be especially important, as we shall see.

21. More Dyadic Operators

The UNION, INTERSECT and MINUS operators are not defined for all pairs of intervals. They are defined only for pairs of intervals such that the union, intersection or difference of their sets of contained points constitutes a single interval.

So the operands *i1* and *i2* of UNION must be such that *i1* MERGES *i2* is true.

Exercise: What constraint must the operands of INTERSECT satisfy? And those of MINUS?

How could these operators be defined if we didn't have interval types? They would have to return 2-tuples!

22. CHAPTER 7: The COLLAPSE and EXPAND Operators

These are a pair of operators each of which operates on a given set of intervals (all of the same type) and returns an equivalent set of intervals (of that same type).

The COLLAPSE and EXPAND operators are not very useful in themselves, but they help with the definition of the more important operators to come.

23. Sets of Intervals

If several different forms are deemed to represent the same thing under some **equivalence relationship**, then certain of those forms might be the preferred ones in certain circumstances. A form that is the preferred one for some purpose is a **canonical form.**

24. Collapsed Form

A set of intervals in **collapsed form** is the smallest of all the sets of intervals that are equivalent to it under the given equivalence relationship (ultimately constituting the same set of points) …

25. Expanded Form

… and a set of intervals in **expanded form** is the biggest of such sets.

26. COLLAPSE and EXPAND

27. CHAPTER 8: The PACK and UNPACK Operators

We introduce two new *relational* operators. There is no suggestion that relational algebra is incomplete without them. Each is a shorthand that can be defined in terms of the operators you are already familiar with.

28. Packed Form and Unpacked Form

A relation that is not in packed or unpacked form exhibits redundancy (and suffers from problems similar to those of SQL tables that contain duplicate rows).

29. Packed Form

The syntax for PACK is:

```
PACK rel exp ON ( attribute-name-commalist )
```

In CS253 we confine our attention (mostly) to packing on a single attribute.

30. Unpacked Form

The notes on the previous slide apply here too, *mutatis mutandis*.

In unpacked form every interval is a unit interval, therefore containing just a single point.

The intervals of an unpacked form could easily be replaced by their single point values, but for definitional purposes it is more convenient to keep the intervals. We expect unpacked forms to exist mostly only conceptually; we do not expect actually to "see" them very often.

If all relations were in unpacked form, queries, constraints, and updates would be as easy to express as they are without the shorthands that we propose. But relations in unpacked form are difficult to interpret and might be huge—especially when the scale of the point type is very small (say, 1 microsecond).

As we shall see, the proposed shorthands allow us to think we are operating on unpacked forms even though we usually "see" the packed forms.

31. Properties of PACK and UNPACK

Packing and unpacking on several attributes are really beyond the scope of CS253, but you might like to note in passing some possibly surprising properties.

Note that packing on two attributes is not simply packing on one and then packing the result on the other. In general, you have to unpack on both first. Even then, the result of packing on both depends on the order in which you do the packing. That is why **Tutorial D** uses rounded parentheses for enclosing the attribute name list, as opposed to the curly braces, {...}, that we normally prefer for lists in which the order of elements is immaterial. In the case of UNPACK, the order is immaterial, but we thought it would be confusing if UNPACK and PACK used different notations.

32. CHAPTER 9: Generalizing the Relational Operators

At last we are able to introduce something really useful for temporal database purposes. We define syntax for an extension that is applicable in the same way to every relational operator.

33. Tutorial D's Relational Operators

The solid triangles, ◂ and ▸, used as parentheses in our proposed extension, are not available on conventional keyboards. For that reason **MighTyD** uses << and >> instead.

It is not important for you to remember all of **Tutorial D**'s relational algebra operators in detail. It is much more important for you to understand the single principle underlying each of the new "U_" operators (U for USING).

Each "old" operator has a U_ counterpart. What's more, each U_ operator degenerates to its "old" counterpart when it is invoked with no USING attributes.

Later we shall see how this same USING construct applies to constraints and updating operations too.

34. USING Example 1

Notice that the U_project example shown on this slide solves the first of our two "very difficult" queries ...

35. Example 2: U_NOT MATCHING

… and the U_NOT MATCHING example solves the other one.

36. Example 3: U_SUMMARIZE

You might need to refresh your memory on SUMMARIZE. Note the PER clause. This is **Tutorial D**'s counterpart of (and improvement on) SQL's GROUP BY. It allows the "grouping columns" to be taken from another relation—S_DURING in the example. Thus we can calculate aggregates for suppliers who supply no parts at all as well as for those who do supply something. **Tutorial D** also allows BY to be used, with exactly the same meaning as SQL's GROUP BY. Here, BY(S#, DURING) would be shorthand for PER (SP_DURING {S#, DURING}).

U_SUMMARIZE turns out to be especially interesting …

37. U_SUMMARIZE is Interesting (1)

The example on this slide is the temporal counterpart of

```
SUMMARIZE SP PER ( S{ } ) ADD COUNT AS NO_OF_PARTS
```

which gives a unary relation consisting of a single tuple showing the number of distinct cases of a supplier being able to supply a part. This is not necessarily the same as the total number of parts available from any supplier! To obtain that you would write this:

```
SUMMARIZE SP{ P# } PER ( TABLE_DEE ) ADD COUNT AS NO_OF_PARTS
```

We summarize that projection of SP to make sure each supplied part is counted only once, and we summarize per TABLE_DEE to be sure of getting a result showing zero, instead of an empty relation, in the case where there are no suppliers at all.

38. U_SUMMARIZE is Interesting (2)

This is something of an exception to the general rule that both operand relations are conceptually unpacked on the USING attribute(s). Here, the single USING attribute doesn't even exist in the PER relation, but that's okay because the operation is still well defined (and possibly useful).

U_JOIN could be subject to similar treatment, not insisting on both operands having all of the specified USING attributes.

Perhaps the general rule should be revised and relaxed, such that each relation operand is conceptually unpacked on every attribute whose name is included in the USING list.

39. CHAPTER 10: Database Design

40. Contents

41. Current Relvars Only

Note that SSSC is in 5NF (fifth normal form) and yet is decomposable. For example, we could split it into three binary relvars with attributes {S#, SNAME}, (S#, STATUS} and {S#, CITY}. As it happens, each of those three would be in 6NF, whereas SSSC, by virtue of being decomposable, is not in 6NF.

Exercise: Why do we normally not decompose relvars such as SSSC, with our existing technology? What constraints would need to be declared if we did decompose it as suggested?

We shall see that 6NF, while possibly a bad idea here, becomes a positively good idea when we "temporalize" the database.

42. Semitemporalizing SSSC (try 1)

The predicate for SSSC (try 1) is something like this:

> Ever since day SINCE, supplier S# has been under contract, has been named SNAME, has had status STATUS and has been located in city CITY.

The SINCE attribute value in each tuple gives the date since a certain proposition has held true. The proposition in question is actually a compound one formed by the conjunction ("and-ing") of four "atomic" propositions. We cannot tell since when any of those atomic propositions has held true. If we wish to do that, we need a slightly more complicated design.

Note that SSSC_SINCE, like SSSC, is in 5NF but not in 6NF, and 5NF is still sufficient.

To overcome the problem mentioned on the slide, we really need a separate "since" attribute for each attribute of SSSC, so to speak, as shown on the next slide.

43. Semitemporalizing SSSC (try 2)

Again we are in 5NF but not in 6NF, and again 5NF is sufficient.

Notice how even the key (S#) has a corresponding "since" attribute, S#_SINCE. It indicates the date on which supplier S# was placed under contract. Since that date it is possible that S#'s name, status and city have all changed, so we do need this date to be separately recorded, if we need it at all. (If the key is composite, we do not need a separate "since" attribute for each component of the key.)

Exercise: What constraints should be declared for S_SINCE?

44. Fully Temporalizing SSSC

For each "since" attribute of SSSC_SINCE, we have a corresponding "during" relvar. The others, not shown on the slide, would be S_STATUS_DURING and S_CITY_DURING.

We call this vertical decomposition because it is "column-wise", according to the normal tabular representation of relations.

45. Sixth Normal Form (6NF)

Recall that according to the normal definition of 5NF, a **join dependency** is said to hold in relvar *R* if there exist *n* distinct projections of *R*, *p1, p2, ... pn,* such that $n > 1$ and at all times *R* is equal to *p1* JOIN *p2* JOIN ... *pn*. (Cases where $n = 2$ are by far the most common.)

A join dependency is "implied by a candidate key of *R*" if each of the projections includes each attribute of that candidate key.

For the purposes of 6NF, the definition is altered slightly such that the projections become U_projections and the JOINs become U_JOINs. Recall that the U_ operators degenerate to their "non-U_" counterparts when the USING list is empty, so the definitions based on U_ operators are backwards compatible with the old ones.

46. "Circumlocution" and 6NF

The example on this slide illustrates a problem we will come back to when we study the kinds of constraints needed for temporal databases. We use the term *circumlocution*—saying something in a needlessly roundabout way—because in the top relation on the slide the fact that S1 was named Smith from day 1 to day 9 is expressed via two tuples when one should be sufficient for that purpose.

The slide also illustrates the importance of using U_projection to achieve the split. Regular projection would have produced two relations of cardinality 2. As one of these would not be in packed form, we would not have solved the problem.

47. "The Moving Point NOW"

Some authorities have advocated use of a special marker, NOW. The problems with this approach are described in detail on pages 177-180 of the book.

For example, consider the interval [NOW:*d14*]. What happens if that interval is recorded somewhere in the database and the clock reaches day 15? For another example, what is the effect, on day 14, of assigning the interval [*d01*, NOW] to variable I1? Does I1 have the value [*d01*:*d14*] or the "value" [*d01*:NOW]? Does I1 compare equal to [*d01*, NOW] on the day the assignment is performed? And on the next day?

Matters get even worse if NULL is used. Think of what effects that might have on all the operators defined for intervals and for PACK, UNPACK,etc.

Using "the end of time" can be confusing if lengths of intervals are taken as measures of how long something has been true for.

48. Horizontal Decomposition

In other words, if we take as our starting point a proposed relvar with attributes S#, SNAME, STATUS, CITY, DURING, we first horizontally decompose to give S_SINCE (with the SINCE attribute replacing DURING) and a "during" relvar for the historical information. Then we vertically decompose the "during" relvar, using U_projection.

49. CHAPTER 11: Integrity Constraints I

Our study of constraints divides naturally into two parts. The first part (here) deals with temporal counterparts of keys and foreign keys. The second part deals with altogether new kinds of constraints occasioned by horizontal decomposition.

50. Candidate Keys and Related Constraints

Having explained vertical decomposition, we can now dispense with S_NAME_DURING and S_CITY_DURING because their treatment will be the same as that of S_STATUS_DURING.

Redundancy: saying the same thing more than once.

Circumlocution: saying something in a roundabout way.

Contradiction: saying something that cannot be true, such as "S1's status is 20 and S1's status is 30".

51. The Redundancy Problem

52. The Circumlocution Problem

53. Solving The Redundancy and Circumlocution Problems

54. The Contradiction Problem

55. Solving The Contradiction Problem

We call this shorthand a WHEN/THEN constraint.

56. WHEN/THEN without PACKED ON

The question arises as to whether a WHEN/THEN constraint is ever needed without a PACKED ON constraint. This example is our best attempt to find such a case. It is not a very compelling example. (Can you think of a better one?)

Omitting the PACKED ON constraint would permit the tuple ([1994:1995], Clinton) to appear in addition to those shown in the example, though the WHEN/THEN constraint prohibits ([1994:1995], Lincoln) from being inserted.

Notice in passing that there appear to be a couple of gaps in the historical record here, for the intervals 1989-1992 and 2001 to the present day. Did the person responsible for the example accidentally miss something out or is there mischief afoot? In case you think another kind of constraint is needed to prevent such "accidents", you are absolutely right! We call this kind of constraint a **denseness** constraint. We shall come across the need for these in our suppliers-and-shipments database very shortly.

It seems that mostly both PACKED ON and WHEN/THEN are required, so a further shorthand is justified.

57. Neither WHEN/THEN nor PACKED ON

Can you think of a compelling example of a "during" relvar where neither a PACKED ON constraint nor a WHEN/THEN constraint is needed? We can't, as this feeble example—our best attempt to find one—illustrates.

58. WHEN / THEN and PACKED ON both required

Here is the promised shorthand to cover the normal case, where both PACKED ON (DURING) and WHEN UNPACKED ON (DURING) THEN KEY ... are both required. Yet another application of the USING construct. (And there are more to come, when we deal with updating.)

59. CHAPTER 12: Integrity Constraints II

60. General Constraints

That the requirements fall neatly into three groups of three is partly an accident of our chosen example. However, the method of grouping follows a general pattern that can be applied in any database design.

61. Requirement Group 1

These are the three that relate to a supplier being under contract.

If we were recording suppliers' names and cities as well as their statuses, then Requirement R3 would be accompanied by two more similar requirements relating to name and city.

This sets the theme, which recurs with some subtle variations, as we are about to see ...

62. Requirement Group 2

Requirement R4 is obviously of the same kind as Requirement R1, but here it addresses contradiction as well as redundancy. We don't want more than one tuple indicating that S1 has status 20 on day 1, for example, nor do we want one tuple showing S1 as having status 20 on day 1 and another showing S1 as having status 30 on day 1.

Requirement R6 is not only similar in kind to Requirement R3: it is the inverse of R3.

63. Requirement Group 3

The variation here is that Requirement R9, unlike R3 and R6, is not accompanied by its inverse: a supplier who is unable to supply anything on a certain day is permitted to be under contract on that day.

64. Meeting the Nine Requirements (a): current relvars only

This slide shows **Tutorial D** constraint declarations that are needed to meet the nine requirements in the "semitemporal" counterpart of our database. Perhaps there is no compelling need for any new shorthands yet.

Recall that IS_EMPTY (*rel expr*) is **Tutorial D**'s shorthand hand for *rel expr* { } = TABLE_DUM (the relation with no attributes and no tuples).

65. Meeting the Nine Requirements (b): historical relvars only

And here are the **Tutorial D** constraint declarations, using shorthands we have already seen, to handle the case where all the relvars are "during" ones—in other words, where horizontal decomposition has not been needed.

66. Meeting the Nine Requirements (c): current and historical relvars

You can study these constraints if you really want to satisfy yourself that they are correct and do the required job, but the whole point of this slide is to show what a compelling case there is for some much more powerful shorthand than any we have yet introduced.

67. Special Treatment for Current and Historical Relvars

And here are the proposed shorthands. SINCE_FOR associates an attribute of a point type (such as DATE, as here) with another attribute in an intuitive way, and HISTORY_IN associates a "during" relvar with that "since" attribute in an equally intuitive way.

All the constraints we have described under "the nine requirements" are implicitly declared by these uses of SINCE_FOR and HISTORY_IN.

68. CHAPTER 13: Database Queries

69. Database Queries

Recall that a virtual relvar is Tutorial D's counterpart of SQL's "updatable view".

The one illustrated here provides a much more convenient target for the familiar database updating operations than the "since" and "during" relvars, and also factors out a subexpression that is likely to be required in very many queries.

Its form is common to all horizontal decompositions, which makes it possible to conceive of a shorthand for generating it, as we shall eventually see in the next chapter.

70. Query Example

Note that the USING construct is used only once in this example, very near the end. This is a fairly commonly observed phenomenon.

71. CHAPTER 14: Database Updates

72. The Example Database

73. What Are The Problems?

74. U_ update operators

Recall that ":=" is **Tutorial D**'s **assignment** operator.

Even with these U_ update operators, correctly applying required updates to a horizontally decomposed database can be excruciatingly difficult. We really need to be able to use that virtual relvar in which current state and history are combined, as a target of updates, so that the system can take care of special needs such as, for example, data deleted from the "since" relvar being appropriately added to the corresponding "during" relvar.

75. The PORTION Clause

This topic is not included in CS253.

76. Updating the Combination View

77. CHAPTER 15: Stated Times and Logged Times

The topic of this chapter is not included in CS253.

78. Proposed Terminology

79. Special Treatment for Logged Times

80. Chapter 16: Point Types Revisited

This topic is not included in CS253.

81. Appendixes

These topics are not included in CS253.

82. Beware of Wikipedia!

The Wikipedia entry (as on 26 January, 2006) appears to present one particular view of temporal databases that was first formulated in the early 1980s.

Note: "A ... database is a database management system". The use of "database" to mean "database management system" is an unfortunate gaffe that pervades the industrial literature. It looks particularly nonsensical here!

Lorentzos's original work was also done in the 1980s and presents a starkly contrasting, relational view on which this material in CS253 has been based.

83. Beware of Wikipedia!

Note:

- The "two extra fields" assumption—no hint of the advantages of interval types.
- The apparent assumption that "the end of time" is the only method that has been considered to address the problem of the missing upper bound for current information.
- The lack of any caveats regarding the use of this approach.

End of Notes