Database Systems, Fall 2006
IT University of Copenhagen

Lecture 12, part I: Temporal databases

Based on Elmasri & Navathe, 24.2.0-24.2.2.

December 5, 2006

Lecturer: Rasmus Pagh

Today's lecture

- **Temoral data.** How to handle time in databases.
- **Exam preparation.** What to expect at the exam.

The need for temporal data

Some applications (e.g., banking, insurance, hospitals,. . .) should keep a full record of all data (or nearly all data) that has existed in the system.

Preserving data in an accessible form may be a requirement by law.

Basic principle: *Never delete data*.

Advantages:

- Can query old data (e.g. patient records).
- Safeguard against unwanted change – possible to revert changes.

The need for temporal data, cont.

Storage prices diminishing — feasible for many systems to “store all versions” of the database.

This kind of database is called a **temporal database** (“database with time”).

There has been considerable research into how to support temporality as a “native” DBMS feature. However, in this lecture we focus on how to do this using a standard relational DBMS.

Transaction time temporality

Suppose we want to be able to query the database at any (previous) point in time.

Example: How many students were registered for DBT on December 1?

What is a "point in time"? Different granularity may be chosen according to the kind of application (stock trading may require "by minute", for car sales "by day" is probably enough).

To be able to relate all versions of an entity, it is important to have a primary key that does not change (possibly a surrogate key).

Tuple versioning

General method for allowing temporal (transaction time) queries in a relational database:

- Add two attributes to each relation: `Tst` and `Tet` that indicate the *start time* and *end time* for each tuple.
- When inserting a new tuple, `Tst` is set to the current time, and `Tet` is set to a special value **uc** *(Until Changed)*. Think ∞.
- When "deleting" a tuple, `Tet` is set to the current time. No tuple is ever deleted.
- Modifying a tuple corresponds to "deleting" it, and inserting the modified version.

Implementing tuple versioning

The new way of handling insertions, deletions, and modifications may be implemented using “instead of” triggers.

In some applications, most queries are about “current” data.

Then, efficiency may be gained by keeping tuples with `Tet`=**uc** in a separate relation, and using this for such queries.

Special “temporal” indexes that can be used by the DBMS for queries “at time t”, for any t, exist and have essentially the same efficiency as usual B-tree indexes.

Querying tuple versioned data

To ask a query SELECT ...FROM $R_1, \ldots, R_k$ WHERE C on the data at time t, just add the condition that all tuples considered have t in the interval [Tst,Tet].

```
SELECT ...
FROM $R_1, \ldots, R_k$
WHERE $C$
    AND $R_1$.Tst <= t AND t<=$R_1$.Tet
    ...
    AND $R_k$.Tst <= t AND t<=$R_k$.Tet;
```

Some extensions of SQL allow more convenient ways of writing such queries.

Temporal normal form

As described, a lot of space may be used to store copies of nearly identical tuples: One for each update of an attribute.

Possible solution: Create one relation R_a for each attribute a not in the primary key of R. R_a is equal to the projection of R onto a and the primary key attributes. (Then the schema is in *temporal normal form*.)

Elmasri & Navathe state that this makes an expensive "temporal intersection join" necessary. However, this is not true for queries about a particular point in time.

Valid time

In some cases, the time at which a fact was entered into the database is not important: We are interested in the time period in which the fact is *valid*.

Example: The fact that "student x takes course y" is valid during the semester in which the course runs, but may be entered before or after the first day of the semester in the course registration system.

Valid time temporal data

Valid time may be represented similarly to transaction time:

- Add two attributes to each relation: `Vst` and `Vet` that indicate the *valid start time* and *valid end time* for each tuple.
- When inserting a new tuple, `Vst` is set to the *user supplied* valid start time, and `Tet` is set to a special value **now** *(Until Now)*.
- At any time, if the valid end time becomes different from **now**, we may assign the value to `Tet`.

Note that both *proactive* and *retroactive* updates are possible.

Bitemporal databases

In some applications, both *transaction time* and *valid time* of tuples are needed.

These are seen as two independent time dimensions, and the database is called *bitemporal*.

Example: Suppose that a student registers for a course but decides to drop it, then logically we should delete the fact "x takes course y" from the database. If we are interested in patterns of how students change their course choices, we need transaction time temporality.

Example query: How many students on my course changed from another course?

Implementing bitemporal databases

Can be implemented by using the methods described previously “on top of each other” (first valid time, then transaction time).

Indexing of bitemporal data is challenging. Some of the currently best techniques are based on “R-trees”, which was originally conceived for geometric data.

Most important points in this lecture

As a minimum, you should after this lecture:

- Be able to systematically add valid time or transaction time temporality to a database design.

General methods sometimes more complicated than necessary — use common sense.