V

Implementation Techniques

Previous parts focus on enhancing our understanding of the temporal aspects of data and on the provision of additional built-in DBMS support for managing temporal data, covering database manipulation as well as database design. The present part takes a next logical step, that of providing new techniques—where existing techniques fall short—for implementing parts of the proposed temporal support.

Common to the implementation techniques, each aims to realize some database functionality. But it is instructive to make an analytical distinction between three more specific, *possible goals of implementation techniques*. A technique may simply aim to constructively demonstrate that some perceived functionality can be implemented. This is a typical goal of a proof-of-concept implementation. Another more ambitious objective of a technique is to offer an efficient implementation. In this situation, other implementations may possibly exist, and performance studies are typically conducted that explore the performance of the implementation and perhaps compare it to the performance of alternative techniques. The last goal of an implementation technique is that of being an effective technique: in this case, it is the ease with which the technique permits us to realize a certain functionality that is the point of focus. An easy means of realizing a certain functionality is clearly preferable to a more difficult one.

A separate distinction useful for characterizing implementation techniques is the *DBMS architecture assumed* by the techniques—we distinguish between two. Techniques that depend on the ability to customize various components of the DBMS in which they are to be embedded, we say assume an *integrated architecture*. For example, an implementation technique may depend on a certain behavior of the data manager component of the DBMS and is thus dependent on the ability to customize the data manager. An integrated architecture may be the one that most readily leans itself towards efficient implementations, but it may also not be an effective architecture. Perhaps most importantly, the dependence on an integrated architecture of an implementation technique makes the techniques relevant only to those with access to the source code of pre-existing DBMSs. The *layered and extensible-system architectures* represent a natural opposite to the integrated architecture. Implementation techniques that assume these types of architectures for their enhancements to the functionality of a DBMS assume that the DBMS is a black box and attempt to reuse the services of the DBMS for their benefit. In this case, the implementation technique may be seen as an advanced application that runs “on-top” of the DBMS or “inside” the DBMS, depending on the extensibility features offered by the DBMS. This general type of architecture may lend itself to more effective implementation of functionality, but its inherently reduced flexibility may also hinder efficient implementation.

The implementation techniques presented in this part provide good coverage of this three-by-two space, with objective and assumed architecture as the dimensions.

The conventional join operation is important because of its frequent use, because of its inherent $\mathcal{O}(n^2)$ worst-case complexity, enforced by the size of the result of the operation, and because the operation can frequently be computed with higher efficiency than that of the brute-force, nested-loop approach. Temporal-join operations not only involve equality predicates, for which existing techniques are optimized, but typically also involve inequality predicates on the time attributes, e.g., to implement an overlap predicate. Inequality predicates typically invalidate existing, advanced non-temporal join techniques, creating a need for special temporal-join techniques.

Chapter 34 introduces a *temporal-join algorithm* based on tuple partitioning. This algorithm avoids the quadratic cost of nested-loop evaluation methods; it also avoids sorting. Performance comparisons between the partition-based algorithm and other evaluation methods are provided. The join algorithm aims to achieve an efficient implementation of an operation for which other implementations exist; and an integrated architecture is assumed.

Chapter 35 proceeds to offer an efficient technique for part of the computation of another fundamental temporal-database operation, namely the *timeslice operation*, which retrieves the state, or timeslice, that was was current at a specific point in time from a relation with transaction-time support. Data is stored as a log of database changes, and timeslices are computed by traversing the log, using previously computed and cached timeslices as outsets. When computing a new timeslice, the cache will contain two candidate outsets: an earlier outset and a later outset. The new timeslice can be computed by either incrementally updating the earlier outset or decrementally "down-dating" the later outset using the log. The chapter provides an efficient algorithm that uses a new data structure, the Pointer-Less Insertion Tree, for determining which outset is the most efficient one to use.

The new algorithm aims to offer a more efficient solution than has so far been available, and in doing so assumes an integrated architecture. Specifically, the algorithm make quite strong assumptions about the data storage layouts, which are difficult to satisfy without the control offered by an integrated architecture. The chapter reports on the performance characteristics of the algorithm and includes relevant comparisons.

The two previous chapters presented techniques intended for valid-time and transaction-time databases, respectively. The following two chapters occurs in the context of bitemporal databases, where both the valid time and transaction time of the data are recorded. The two chapters offer different techniques for indexing the bitemporal extents of data, thus offering competing foundations for the efficient evaluation of predicates that involve the valid and transaction time of data.

Like spatial data, bitemporal data thus has associated two-dimensional regions. Such data is in part naturally now-relative: some data is currently true in the mini-world or is part of the current database state. So, unlike for spatial data, the regions of now-relative bitemporal data grow continuously. Existing indices, including commercially available indices such as B^+- and R-trees, do not contend well with even small amounts of now-relative bitemporal data.

Chapter 36 proposes two extended R^*-trees that are capable of indexing the possibly growing two-dimensional regions associated with bitemporal data, by also letting the internal bounding regions grow. Internal bounding regions may be triangular as well as rectangular. New heuristics for the algorithms that govern the index structure are provided. As a result, dead space and overlap, now also functions of time, are reduced. This chapter's focus is on efficiency, and performance studies indicate that the best extended index is typically significantly faster than the existing R-tree based indices. Although an integrated architecture is assumed, a continuation of this work has resulted in a prototype implementation applicable to an extensible DBMS architecture. However, further improvements of the existing extensible architectures are necessary before this research is practically applicable beyond integrated architectures.

Chapter 37 proposes a new indexing technique that eliminates the different kinds of growing data regions by means of transformations and then indexes the resulting stationary data regions with four R*-trees; and queries on the original data are correspondingly transformed to queries on the transformed data. Extensive performance studies are reported that provide insight into the characteristics and behavior of the four trees storing differently-shaped regions, and they indicate that the new technique yields a performance that is competitive with the best existing index from the previous chapter; and unlike this existing index, the new technique is readily applicable in a layered architecture.

The next three chapters differ from the previous ones in several respects. They do not consider individual, isolated operations, but rather adopt a holistic view and consider the general challenges that occur when attempting to implement a temporal SQL. The traditional approach has been to assume an integrated architecture for implementing a temporal SQL: it has been assumed that a temporal DBMS must be built from scratch, utilizing new technologies for storage, indexing, query optimization, concurrency control, and recovery. (This was also assumed in Chapters 20 and 21.) In contrast these three chapters assume a layered architecture, and their main emphasis is on effective implementation rather than on efficient implementation, although the latter remains a concern. In addition, the last chapter addresses the problems of simply ensuring a correct implementation under the restrictions imposed by a layered architecture.

Chapter 38 introduces and explores the concepts and techniques involved in implementing a temporally enhanced SQL while maximally reusing the facilities of an existing SQL implementation. The topics covered span the choice of an adequate timestamp domain that includes the variable *now*, a query processing architectures, and transaction processing.

Chapter 39 proceeds to identify three layered, or stratum, meta-architectures, each with several specific architectures. Based on a new set of evaluation criteria, advantages and disadvantages of the specific architectures are identified. The chapter also classifies all existing temporal DBMS implementations according to the specific architectures they employ. It is concluded that a stratum architecture is the best short, medium, and perhaps even long-term approach to implementing a temporal DBMS.

Chapter 40 delves into a challenge faced by all proposals for associating timestamp values, denoting valid and transaction time, with data. With few exceptions, the assignment of timestamp values has been considered only in the context of individual modification statements. This chapter goes further and considers timestamping in the context of transactions, and in a layered architecture. The chapter initially identifies and analyzes several problems with straightforward timestamping, then proceeds to propose techniques aimed at solving these problems. Timestamping the results of a transaction with the commit time of the transaction is a promising approach, and a spectrum of techniques for how to do this are explored. Next, although many database facts are valid until the current time, *now*, this value is absent from the existing time data types. Techniques using different substitute values are explored, and the performance of the different proposed techniques are studied. The result is a comprehensive approach that provides application programmers with simple, consistent, and efficient support for modifying bitemporal databases in the context of user transactions.