A Comparative Study of Temporal DBMS Architectures

Costas Vassilakis, Panagiotis Georgiadis, Anna Sotiropoulou
Department of Informatics, University of Athens

Abstract: *In the past years, a number of implementations of temporal DBMSs has been reported. Most of these implementations share a common feature, which is that they have been built as an extension to a snapshot DBMS. In this paper, we present three alternative design approaches that can be used for extending a snapshot DBMS to support temporal data, and evaluate the suitability of each approach, with respect to a number of design objectives.*

1. Introduction

Temporal DBMSs extend the capabilities of snapshot DBMSs by supporting at least one temporal dimension ([16]), thus temporal DBMSs can be considered to be a superset of snapshot DBMSs. Bearing this in mind, a typical approach to implement a temporal DBMS, is to start with an existing snapshot DBMS and enhance it with data types for time representation (e.g. intervals and temporal elements) and temporal operations (e.g. coalescing). This approach has the advantages of reducing to a certain extent the amount of code which must be written, and allowing the implementor to focus only on the problems concerning the temporal aspects of data. These facts have been adopted by researchers: in [3], nine temporal database system implementations are reported[1], out of which only two (HDBMS and TDBMS) have not been based on a snapshot DBMS (notably, however, HDBMS uses BTrieve/Objectrieve for data storage and retrieval). The remaining seven implementations (GCH-OSQL [5], Calanda [15], Chronolog [2], TempCase [19], TempIS [1], TimeDB [4], and VT-SQL [14]) have been based on snapshot DBMSs.

However, diverse design approaches have been followed in these implementations. In some of them, temporal features are integrated within the DBMS kernel, delivering a new, enhanced system, whereas in other implementations, temporal functionality is offered by pieces of software which act as client applications to the snapshot DBMS. In this paper we discuss three approaches for adding temporal functionality to a snapshot DBMS, and assess each approach with respect to the extent that it meets a number of criteria. This survey intends to aid designers of future temporal database systems to make the most appropriate selection of their DBMS architecture, considering the goals of their implementation.

The remnant of this paper is organised as follows: section 2 presents the design objectives, i.e. the criteria which will be used to evaluate each design approach. Section 3 presents alternative solutions of designing a temporal DBMS, and section 4 evaluates the design approaches, with respect to the criteria stated in section 2. Finally, in section 5, conclusions are drawn.

2. Design Objectives

The suitability of each design approach, for extending snapshot DBMSs to include temporal functionality, will be measured against certain criteria, the *design objectives*. The design objectives which will be considered in this paper are defined in the following paragraphs.

1. *Complete temporal functionality*. Temporal data types (e.g. intervals) and operations (e.g. coalescing, temporal union etc.) should be first class citizens within the extended DBMS, i.e. the temporal DBMS should allow their usage wherever an analogous snapshot DBMS data type or operation can be used.
2. *Full snapshot DBMS compatibility*. The extended DBMS must allow for the creation and manipulation of snapshot *data stores* (relations, in the context of relational DBMSs), in the same way that its snapshot version does.
3. *Implementability*. The proposed solution should be implementable, considering the tools which are at the developer's disposal. In particular, access to the underlying DBMSs source code should not be a prerequisite, since it is a hard requirement to satisfy.

[1] TimeIt and TimeMultiCal are also included in this report but are not considered to be DBMSs since they do not provide persistence.

4. *Exploitation of facilities offered by the snapshot DBMS*. It is desirable that facilities offered by the snapshot DBMS, such as join processing algorithms, indexing mechanisms (for non temporal data) etc. are used in the temporal DBMS, without needing to be recoded or modified in any way.
5. *Simplicity of implementation*. More complex solutions are weaker candidates for implementations. Additionally, complex solutions are prone to be less efficient and their code is not easily maintainable.
6. *Data integrity*. Commercial DBMSs provide a data manipulation language (e.g. SQL) as the sole means for accessing the stored data, using the operating system-level protection mechanisms (file or raw disk partition access rights) to prevent users from modifying directly the disk files (or partitions) on which data are stored. Similarly, the proposed solution should provide a mechanism to prohibit users from using the snapshot data manipulation language to modify temporal persistent objects, since such modifications might result to illegal data store states. Figure 1 presents an example in which modifying a valid time relation through SQL, leads it to an illegal state (assuming that an employee has only one salary at any time point).
7. *Performance*. The proposed solution should deliver high performance, using optimisation techniques, wherever possible, and limiting disk input/output and interprocess communication to the minimum necessary degree. No overheads should be imposed to accesses to snapshot data stores. Finally, operation repetition across different system components should be avoided (e.g. each statement should be parsed only once).
8. *Resource consumption*. The proposed system should consume as little system resources as possible. In this paper, memory and interprocess communication resources will be accounted for.
9. *Portability across hardware/software platforms*. The proposed solution should not be tightly bound to specific features offered by the hardware platform or the underlying DBMS.
10. *Additional features*. It is desirable for the proposed system to support a number of non-temporal features offered by the underlying snapshot DBMS, such as transaction management, concurrency control, crash recovery and language extensions.

UPDATE Salary Set EmpId = 'e0001' WHERE EmpId = 'e0002'

EmpId	Salary	Period
e0001	2000	91-93
e0001	2400	94-95
e0002	2600	92-94

→

EmpId	Salary	Period
e0001	2000	91-93
e0001	2400	94-95
e0001	2600	92-94

Figure 1 - Execution of a statement by a snapshot DBMS leads a valid time table to an illegal state

3. The Design Approaches

Three design approaches are identified in this paper for extending existing DBMSs to include support for temporal data and operations. The design approaches are described in sections 3.1 through 3.3. Section 4 provides a critical evaluation of these approaches.

3.1 Integration of Temporal Functionality in the Snapshot DBMS

The first design approach is to integrate temporal support into the underlying snapshot DBMS, as illustrated in figure 2. This means that the following extensions/ modifications should be built into the snapshot DBMS:

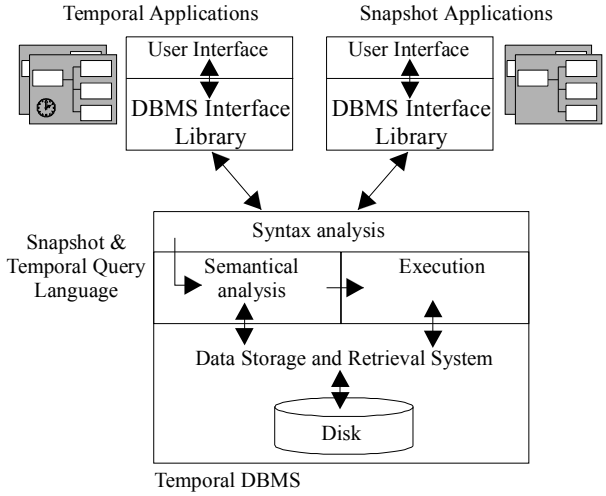

Figure 2 - Temporal functionality is built into the DBMS kernel

1. the appropriate data types (time points, time intervals, temporal elements) must be defined, and their storage structure determined. The kernel should provide conversions between the textual representations, used by applications and users to represent these data types and the internal format, used by the DBMS to store the data to the disks.
2. the semantical analysis module must be extended to determine whether each operation applies to a snapshot data store or a data store with temporal semantics, and schedule the appropriate actions (e.g. coalescing, during data insertion or update, applies

only to valid time data stores -when a coalesced storage schema is used; primary key violation checks are different for temporal and snapshot data stores). For object-oriented DBMSs (OO-DBMSs), this is equivalent to defining temporal persistence classes and providing the appropriate methods for object insertion, deletion and modification.

3. the query execution module must be extended to include additional operations for application of temporal operations which may be applied during query evaluation (e.g. coalescing, temporal aggregation).
4. the snapshot DBMS's DDL and DML parser must be modified to recognise the extended syntax used for temporal operations. For DBMSs, providing temporal functionality without extending the DDL or DML language (e.g. OODAPLEX [22]), this modification is not necessary.

Temporal and snapshot applications use a common interface library to interact with the temporal DBMS. The DBMS interface library provides a collection of host language data types, which are used when data are transferred from the application to the temporal DBMS, or vice versa, together with a set of procedures (or *methods*) which are callable from the host language and trigger actions on behalf of the DBMS. It is possible that the actual procedure calls are inserted into the code by a *preprocessor*, while the programmer describes the actions using a declarative query language (e.g. embedded SQL).

This design approach has been followed in the implementation of TempIS.

3.2 The *Temporal Server* approach

The *temporal server* approach is an elegant solution, which does not modify the snapshot DBMS, but introduces an additional entity, the *temporal server*. The temporal server is logically located between the temporal applications and the snapshot DBMS, as depicted in figure 3. Applications that access temporal data stores use a *temporal DBMS interface library*, which directs requests for data retrieval or modification to the temporal server, who is responsible for the following actions:

1. analyse the requests issued by the applications, determining whether they are conferment either to the temporal query language or to the snapshot query language rules.
2. intercept references to temporal data types (time points, time intervals and temporal elements) which are not supported by the snapshot DBMS kernel, and provide the appropriate conversions between the representations used by the applications and a raw data format (e.g. string) which must be used to store the data into the snapshot DBMS.
3. map the temporal operations to series of snapshot operations, and arrange for the execution of these operations. This involves retrieval, processing and storage of data to the snapshot DBMS, and forwarding of the final results to the application. The temporal server interacts with the snapshot DBMS, using the interface library provided by the snapshot DBMS.

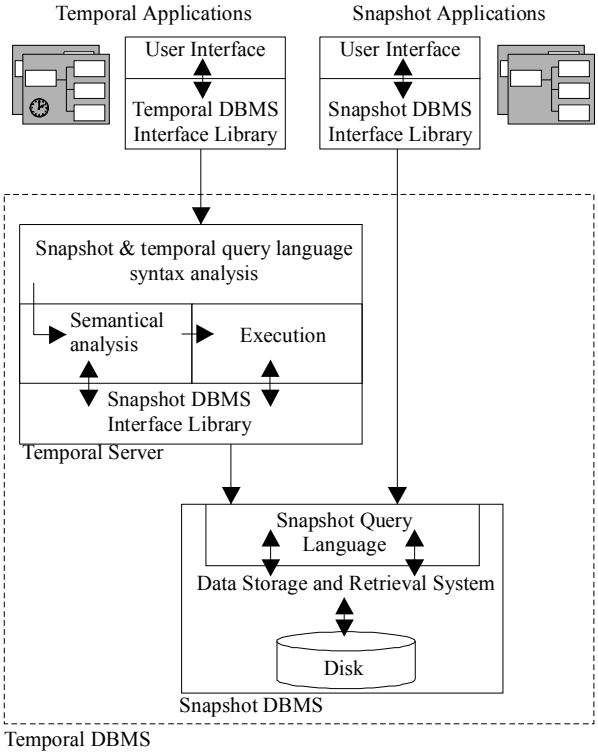

Figure 3 - The temporal server approach

Applications which require data from snapshot data stores only, may connect directly to the snapshot DBMS, using the interface library provided by the snapshot DBMS. If such a connection is established, the application may not use any extensions provided by the temporal server (e.g. query language syntactical or semantical enhancements), since they are not supported by the snapshot DBMS.

Insofar, no implementations using this design approach have been reported.

3.3 Building temporal support into the client application

The third approach is to build directly the support for temporal operations into the client application, as illustrated in figure 4. Temporal applications direct their requests for temporal data store access to a *temporal*

functionality interface library, which is linked into the application's executable code. The temporal functionality interface library is responsible for provision of appropriate conversions for data types not directly supported by the snapshot DBMS, request analysis, mapping of temporal operations to series of snapshot operations and execution of the appropriate actions, in order to complete the request. The temporal functionality interface library interacts with the interface library provided by the snapshot DBMS, in order to fetch data from the snapshot DBMS, or store data in it. Since the snapshot DBMS interface library is built into the application code, the application may forward requests to it directly, bypassing the temporal functionality interface library.

Applications accessing snapshot data stores only, do not need the temporal functionality interface library: only the interface library provided by the snapshot DBMS needs to be linked into the final executable file. In this case, the extensions offered by the temporal functionality interface library are not available to the application.

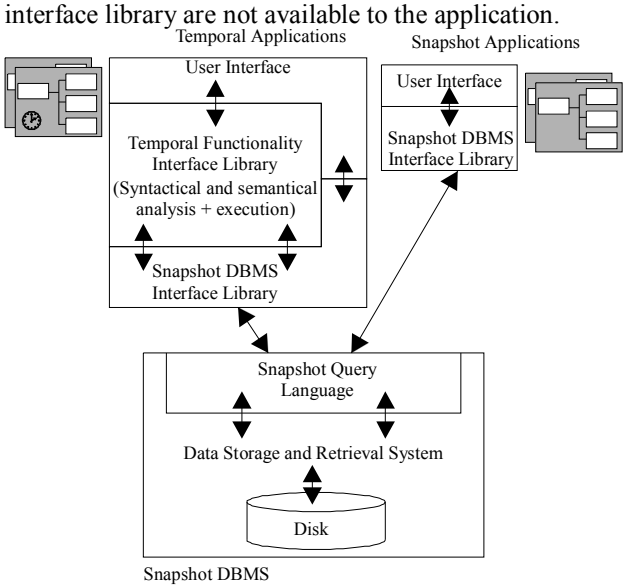

Figure 4 - Temporal functionality built into the client application

This design approach has been followed in the implementation of Arcadia GCH-OSQL, Calanda, ChronoLog, TempCase, TimeDB and VT-SQL[2].

[2] Note that a *monitor application*, which accepts temporal queries, evaluates them and prints out the results, may be used as a *temporal server* by some other application (e.g. TempCase operates on top of VT-SQL), thus the distinction between a temporal server and a client application is not clear-cut. We classify under *client application* all pieces of software that can be used interactively by the end user.

4. Evaluation of the Design Approaches

In this section we evaluate the degree to which each design approach fulfils the objectives defined in section 2.

4.1 Complete temporal functionality

It may be argued that the level of temporal functionality offered by the extended DBMS is independent of the design approach that will be taken. However, implementation considerations may lead designers to limit the amount of temporal functionality offered by the extended DBMS. For example, consider the following TSQL2 query, which retrieves the names of employees who have worked for two years under the management of either *e1* or *e2*.

```
SELECT EmpId, EmpName FROM EmployeeData
WHERE EmpId IN (SELECT SNAPSHOT E.EmpId
      FROM EmpMgr(EmpId) AS E, E(MgrId) AS F
      WHERE F.MgrId = 'e1' OR F.MgrId = 'e2' AND
            CAST (E) AS INTERVAL YEAR >
            INTERVAL '2' YEAR)
```

In order to evaluate this query, a temporal DBMS will formulate a query execution plan (QEP), similar to the one illustrated in figure 5.

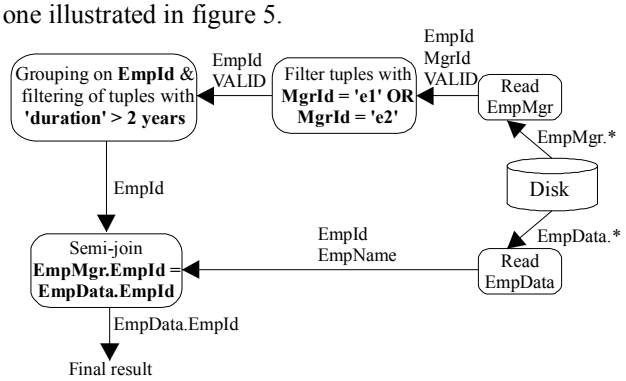

Figure 5 - Query execution plan

The ability to follow the QEP may depend on the design approach used for the temporal DBMS, as follows: if temporal support is built directly into the snapshot DBMS kernel, then all operations (filtering, grouping/filtering and semi-join) are available to the execution module, so the query may be evaluated. However, if temporal support is implemented outside the snapshot DBMS kernel, then, it is clear that data must be moved to the module implementing the temporal extensions (either the temporal server or the temporal functionality interface library in the client application), in order to execute the grouping/filtering operation. The key question is how the semi-join will be evaluated, with two possible alternatives:

1. the module implementing the temporal extensions is able to execute the semi-join, so the two final operations are performed within that module, as illustrated in figure 6. However, other queries may require sorting, aggregation, anti-join, or any other snapshot operation to be performed after some temporal operation; but if we build algorithms for every snapshot operation within the module implementing the temporal extensions, we have effectively implemented a complete temporal DBMS, apart from the physical storage module. This is clearly undesirable, since the reason for extending an existing DBMS rather than building one anew, was to avoid this piece of work.

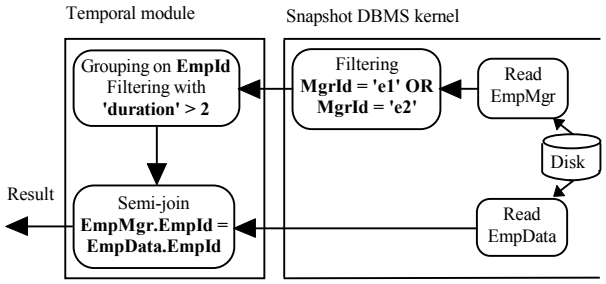

Figure 6 - QEP execution, when the temporal module implements snapshot operations

2. the module implementing the temporal extensions is not able to evaluate the semi-join or any other snapshot operation; rather, after performing the grouping/filtering operation, data are stored back in some temporary data store within the snapshot DBMS, and the snapshot DBMS's kernel evaluates the semi-join, in order to produce the final result, as illustrated in figure 7. This alternative is more easily implementable than the previous one, but presents serious performance problems, since data transfers between the snapshot DBMS kernel and the module implementing the temporal extensions are necessary, and a *holding point* is introduced: the evaluation of the semi-join may not start before all the results of the grouping/filtering operation have been stored into the temporal data store, whereas in "normal" execution conditions, the evaluation may start as soon as the first tuple (or block of tuples) of the grouping/filtering operation's output is available. Additional storage space is also needed, to accommodate the temporary data store.

Considering the problems in both alternatives, the designers of the temporal extension may opt for some limitation to the set of temporal queries that can be answered by the temporal DBMS; a likely choice may be that the temporal DBMS will be able to evaluate only queries which satisfy the following two conditions:

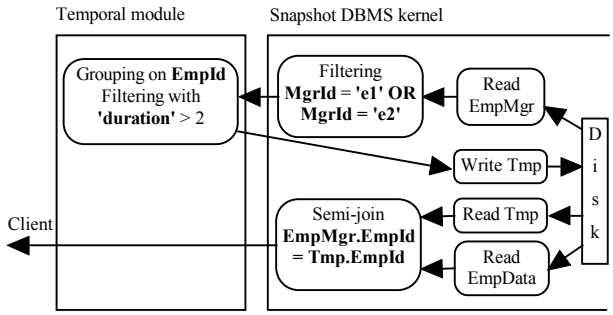

Figure 7 - QEP execution, when the temporal module does not implement snapshot operations

1. the QEP can be partitioned in two sets, the first one including the snapshot operations and the second one comprising of the temporal operations.
2. there exists a valid rearrangement of the QEP, equivalent to the initial one, such that no operation in the second set is followed by an operation belonging to the first set.

4.2 Full snapshot DBMS compatibility

Full snapshot DBMS compatibility is an easy requirement to satisfy, since all the algorithms needed to provide the snapshot operations already exist within the snapshot DBMS. Full snapshot DBMS compatibility is guaranteed if the following conditions are met:

1. all extensions to the query language come either as new statements or as optional clauses within existing statements. This guarantees that every statement that is syntactically valid in the snapshot query language remains valid in the temporal query language. Effectively, the temporal query language parser must accept all the snapshot query language statements.
2. the temporal DBMS interface library (or temporal functionality interface library, in the 3rd design approach) supports all the calls which are provided by the snapshot DBMS interface library. This ensures that applications can interact with the temporal DBMS in the same way they interacted with the snapshot DBMS.
3. if a snapshot data store is accessed through a snapshot DBMS interface library procedure call, or via execution of a non-extended statement, then the results of the access (data store modifications and/or returned data) are the same with the results that the snapshot DBMS would produce, if it processed the same request. This guarantees that the semantics of snapshot operations applied on snapshot data stores are not modified.

   Practically, this means that the temporal DBMS checks both the query syntax and whether the accessed data store is a snapshot or a temporal one,

before determining the operation semantics and formulating the query execution plan. In order to distinguish between snapshot and temporal data stores, it is required that adequate information is stored within a *data dictionary*, which is either the standard data dictionary maintained by the snapshot DBMS (extended with the appropriate columns), or a new data store.

4.3 Implementability.

In general, when extending a DBMS to include temporal functionality, the following actions are required:

1. implement the appropriate data types to represent time (time points, time intervals and, possibly, temporal elements).
2. define functions that will operate on the data types which are used to represent time (e.g. a function accepting an argument of type interval and returning its starting point).
3. implement operations on temporal data stores (e.g. a special version of *delete* for transaction time data stores, coalescing, temporal union, etc.).
4. define syntactical and extensions to the snapshot data definition and manipulation language, which invoke the temporal operations.

When extensions are built outside the DBMS kernel, it is possible to support all the features listed above, by adding suitable code either to the client application or to the temporal server. In the following paragraphs, we will elaborate on the ability to incorporate support for these features directly into the DBMS kernel. Our review will cover two classes of database systems, namely relational and object-oriented.

Data types for time representation. All OO-DBMSs provide means of extending the built-in object lattice, and support *lists*, facilitating the implementation of the *temporal element* data type. The new data types are *first class objects*, i.e. they have the same functionality with the built-in data types. Moreover, object-oriented DBMSs allow the programmer to differentiate the *internal representation* (i.e. the actual bytes that are stored on the disk) from the *external representation* of the data type (i.e. the form in which the user inputs and sees data), permitting thus the usage of a space- and operation-efficient internal representation, and a customisable, user-friendly external representation.

Relational DBMSs are less flexible with supporting new data types. Since lists do not fit in the first normal form model, temporal elements cannot be stored as an atomic data type: normalisation rules ([6]) suggest that an additional relation must be used for storing each column of type *temporal element*. For incorporating atomic data types, such as *interval*, various DBMSs[3] provide different levels of extensibility (in the following we assume that the programmer has no access to the DBMS's source code):

- Ingres offers the *Object Manager* tool ([8]), which allows for the definition of new data types. The user-defined data types are first level objects and may have different external and internal representations.
- Sybase provides the *sp_addatatype* stored procedure to define a new data type, but only as an alias of a built-in data type ([17]). However, it allows a *validation rule* to be bound to the new data type, so it can be verified that the data stored to columns of the user-defined data type conform to some rules. For example, we could define the data type *interval* with granularity of *date*, represented as *(date1, date2)* using the following command batch:

```
sp_addatatype interval, char(24)
sp_createrule interval_rule as
    substr(@value, 1, 1) = '(' AND
    substr(@value, 24, 1) = ')' AND
    substr(@value, 12, 2) = ', ' AND
    convert(substr(@value, 2, 10), date) <
                convert(substr(@value, 14, 10), date)
sp_bindrule interval_rule, interval
```

  Notice that, using this technique, the external representation is the same as the internal representation and quite inflexible, since it requires the user to start the string with a left parenthesis, end it with a right one, and separate dates using a comma and a space. Additionally, each date must be entered using a 10-character string.
- Oracle does not provide a tool for defining a new data type, but allows for a *CHECK* clause to be included in a column definition ([12]). This can be exploited by translating the *interval* data type, when found in a *CREATE TABLE* statement to a *CHAR(N)* data type, with the appropriate validation check. Although this approach works, it is as inflexible as the previous one and, additionally, system catalogues will report that the column's type is *CHAR(N)*, instead of *INTERVAL*.

Functions on the data types representing time. In OO-DBMSs, each data type is coupled with its *behaviour*, i.e. a set of procedures and functions (*methods*, in object-oriented terminology), thus the programmer can

[3] The list of DBMSs is not exhaustive; it is only intended to outline the extensibility capabilities of the relational DBMSs, with respect to adding new data types.

define all the functions operating on data types representing time.

Different relational DBMSs, provide different levels of ability to define new functions:

- Ingres offers the *Object Manager* tool ([8]), which allows for the definition of new functions. These functions may include type checking and can be used in any place that a built-in function is allowed.
- Oracle supports the *package* concept ([12]) for defining new functions and procedures. Once a package is created, the functions and procedures defined within it are accessible to all users and applications.
- Sybase does not support any kind of user-defined functions.

Operations on temporal data stores. Temporal data stores require special handling, since data insertion and update may need to be followed by coalescing (depending on whether the data storage model is coalesced or not), special forms of statements must be supported (eg. a DELETE command containing a VALID PERIOD clause, in TSQL2 [20]) and temporal flavours of primary keys may need to be considered (e.g. the *time point* keys described in [13]). Additionally, when temporal data are retrieved, temporal operations (e.g. coalescing, temporal union etc.) may be applied on them.

None of the relational DBMSs allows for definition or customisation of the code which handles data modifications. However, most relational DBMSs provide *triggers* that are *fired* when data stores are modified, and some relational DBMSs provide facilities for storing code within the database (e.g. Oracle and Sybase allow for definitions of *procedures* within *packages* ([12]) and *stored procedures* ([17]), respectively). Triggers may be used for supporting temporal keys (by attaching them to the INSERT and UPDATE events and associating appropriate pieces of code) and ensuring a coalesced storage schema (although the latter may prove to be very tricky, due to chained trigger firing). Procedures may be used to implement the special forms of statements, as long as the language in which they are coded is flexible enough to allow a dynamic WHERE clause. Figure 8 illustrates examples of using triggers to support primary keys and coalesced storage schema (the scheme of table *Employee* is considered to contain columns *EmpId* and *Period*; no two rows of *Employee* are allowed to have identical values for *EmpId* and overlapping values for *Period*).

```
CREATE TRIGGER Insert_Guard_Employee
BEFORE INSERT ON Employee FOR EACH ROW
WHEN (EXISTS (SELECT * FROM Employee
              WHERE   new.EmpId =
              Employee.EmpId
                 AND  overlap(new.Period,
                              Employee.Period))
BEGIN
raise_application_error(-5000, "Primary key
               violation on table Employee");
END;

CREATE TRIGGER Coalesce_Employee
AFTER INSERT OR UPDATE ON Employee
BEGIN
/* Avoid recursive firing */
ALTER TRIGGER Coalesce_Employee DISABLE;
/* Code for coalescing table Employee */
/* Reenable trigger. */
ALTER TRIGGER Coalesce_Employee ENABLE;
END;
```

Figure 8 - Using triggers to support temporal operations

Whenever a temporal table is created, these triggers and procedures must be automatically generated; this can be handled either by an external temporal module (the temporal server or the temporal functionality interface library) or by a special procedure, which is specifically used for creating temporal tables. Unfortunately, this approach presents the following problems:

1. triggers have been designed to tackle a different set of problems, such as enforcing referential integrity or appending a record to a log, when some condition is met. Using triggers to support a coalesced storage scheme is not a normal choice and may introduce conflicts with other user-defined triggers. Additionally, the temporal module must provide security mechanisms, to ensure that the triggers enforcing the temporal semantics are not modified by the user.
2. both triggers and procedures are of static nature (i.e. they operate on tables with a specific schema), thus different pieces of code must be assigned to triggers/procedures operating on different tables. Consequently, the temporal module must be able to generate automatically the pieces of code which will be associated with the triggers/procedures of each table.
3. as each temporal table requires its own set of procedures and triggers, the amount of code stored in the database increases significantly. This is undesirable both in terms of space and administrative overhead, on behalf of the DBMS, which may lead to degraded performance.
4. since the code is static, it is difficult to use optimisation techniques.
5. if no external temporal module is used, then users should be aware of the special procedures which are

used for temporal table creation and invocation of special forms of statements. This is undesirable since it leads to loss of uniformity (some operations are performed via *statements*, while others are initiated through procedure calls).

Retrieval queries involving temporal operations, such as coalescing, cannot be supported using these mechanisms. Temporal operations are applied on arbitrary relation schemata and thus cannot be handled by static code. Additionally, procedures are not generally allowed to return set-type results (i.e. relations), while triggers do not return data and may not be associated with data retrievals.

For these reasons, when building temporal extensions to a relational DBMS (having no access to the kernel source code) it is preferable to implement the operations on temporal data stores outside the DBMS kernel.

If an OO-DBMS is used, temporal operations in retrieval queries can be mapped to calls to methods which accept set-type arguments and yield set-type results. The code handling data store modifications can be also be stored within the DBMS kernel, in the form of general-purpose procedures ([7]) or methods of specific classes. The Multimedia Information Manager (MIM) used in ORION ([9]) is a good example of the capabilities of OO-DBMSs in handling data with specific data storage requirements. Admittedly, not all OO-DBMSs are as flexible as ORION: there is no obvious way to implement an analogous mechanism in IRIS, while in O2 the O2Engine layer ([11]) has to be extended (or replaced), which is a more difficult task than adding a few classes.

Extending the data definition and manipulation language. This feature is not supported by any DBMS; in order to extend the DDL or the DML, the programmer must either have access to the DBMS source or the extensions must be built outside the DBMS kernel. However, with OO-DBMSs, a significant part of the temporal language functionality may be supported by methods, thus reducing the need to extend the language itself.

4.4 Exploitation of facilities offered by the snapshot DBMS.

If the temporal extensions are built into the snapshot DBMS kernel, then the temporal features will coexist with the snapshot mechanisms that are provided by the snapshot DBMS, and the execution procedure may select the most appropriate algorithm for each operation. If, however, temporal support is implemented outside of the DBMS kernel, some operations may need to be recoded, in order to provide complete functionality as shown in section 4.1.

4.5 Simplicity of implementation.

Simplicity of implementation is a very important factor, not only because less coding effort is required, but additionally the final product will be more stable, error-free and maintainable. Simplicity of implementation is affected by a number of factors:

1. *level of programming*: writing low-level, kernel-specific code is generally a more complex task than programming in a high-level language, say C, using embedded SQL to interact with the DBMS.
2. *availability of debugging tools*. If the extensions are built outside the snapshot DBMS kernel (either using the *temporal server* approach or building temporal support into the client application), standard debugging tools may be used in order to trace the erroneous spots, while the snapshot DBMS can be used to monitor changes to the database state. If, however, the extensions are integrated into the DBMS kernel, debugging is more cumbersome, and coding errors may be more disastrous, since they may result to corruption of data stores and/or "hanging" of the DBMS.
3. *amount of code that must be written*. As already stated in section 4.4, implementing the extensions outside of the DBMS kernel may require recoding of a number of snapshot operations, increasing the bulk of code that must be written.
4. *special techniques that must be employed.* The *temporal server* approach introduces another level of complexity, because *fairness* and minimisation of the average and weighted request turnaround time ([10]) must be pursued. For example, consider that two temporal applications TA_1 and TA_2 are simultaneously connected to the temporal server depicted in figure 3, issuing the following queries, respectively:

```
Q1: SELECT *
      FROM BigDataStore(Period), BigDataStore2(Period)
Q2: SELECT * FROM SmallDataStore
```

Assuming that Q1 is issued right before Q2, if the temporal server evaluates queries sequentially, then the request issued by TA_2 (Q2) cannot be processed before Q1 is evaluated. This will lead to an unacceptable delay for TA_2, indicating that sequential processing is not an appropriate approach for the temporal server. A more suitable approach is to interleave the execution of the queries, by using some preemptive scheduling technique with time quotas

(e.g. round robin) or by using *lightweight processes* (or *threads*-[18]) within the temporal server, and assigning each incoming connection to a different thread. Both of these approaches increase the complexity of the temporal server. An alternative solution, would be to use one temporal server per application connection, and leave the process switching and fairness aspects to the operating system.

Note that these techniques need not be employed if temporal support is integrated either in the snapshot DBMS kernel or in the client application: in the former case, the mechanisms for query execution interleaving which are built in the snapshot DBMS can be exploited, while in the latter case we can identify two phases in query execution, the *snapshot phase* (executed within the snapshot DBMS kernel) and the *temporal phase* (executed within the temporal application). The snapshot DBMS will arrange for execution interleaving for the first phase of the queries, while the operating system is responsible for allocating the CPU to the temporal applications, during the execution of the second phase of the queries.

4.6 Data integrity.

Data integrity is jeopardised if the pieces of code which handle modifications to temporal data stores can be bypassed, and data stores are modified using the snapshot DBMS data manipulation language. When this code has been stored within the DBMS, three cases can be identified:

- the source code is available, thus the extensions are hard-coded within the kernel and cannot be bypassed; in this case, data integrity is guaranteed.
- modifications are monitored by triggers that cannot be disabled or performed through class methods which cannot be overridden; no illegal modifications are possible in this case.
- modifications are performed through general-purpose procedures; in this case it cannot be guaranteed that the user will invoke the appropriate procedure, instead of directly modifying the data store using the snapshot DBMS's DML, and additional security measures are called for.

If the code handling modifications to temporal data stores is stored outside of the DBMS kernel (within the temporal server or the temporal functionality library), then users must be prevented from connecting directly to the snapshot DBMS and using it to modify data stores with temporal semantics, as such modifications may lead the data stores to inconsistent states (see figure 1). The temporal server approach can incorporate a protection scheme, outlined below:

1. for each user of the temporal DBMS, two user ids are created at the snapshot DBMS level: an *external user id* and a *shadow user id*. The mappings between the *external user ids* and the *shadow user ids* are maintained either in a table within the snapshot DBMS or in an operating system file; in both cases, the repository is only accessible to the snapshot DBMS administrator. Only the external user id, along with the password, is disclosed to the user.
2. if a user must access a snapshot database (i.e. a database consisting only of snapshot data stores), then access to that database is granted to the external user id; if, however, access to a temporal database is required (i.e. a database containing *at least* one temporal table), the database administrator grants access to the shadow user id.
3. when the temporal server receives a request for connection to a temporal database, it maps the external user id that the user provided to the shadow user id, using the repository (the temporal server must run under the database administrator's user id, in order to be able to perform this mapping), and opens a connection to the database using the shadow user id.

Under this scheme, the user can use a direct connection to the snapshot DBMS, in order to access snapshot databases. However, the user cannot access directly temporal databases, since the external user id, is not authorised to use any of these databases. Temporal databases may be accessed through the temporal server only.

Note that this approach cannot be used when temporal support has been built into the client application for two reasons:

1. the temporal server runs under the database administrator's user id, in order to be able to access the repository in which shadow-to-external user id mappings are maintained. Allowing any client application to run under this user id is extremely dangerous for the security and integrity of the databases.
2. since a connection between the client application and the snapshot DBMS will be open, the user can bypass the temporal functionality interface library and forward directly DML statements to the snapshot DBMS, through the snapshot DBMS interface library.

4.7 Performance.

Building the temporal extensions directly into the snapshot DBMS kernel is expected to deliver the highest

performance among the three approaches. Statements are parsed only once, by the DBMS parser, and all operations are performed within the DBMS kernel, so no extraneous process switching between the DBMS kernel and the application is introduced. No additional operations are performed when a snapshot database is accessed, whereas for temporal databases, the system dictionary must be queried to determine whether the accessed tables incorporate temporal semantics or not. Data are moved to the client application only when the application has requested for them, minimising the time spent for interprocess communication.

When temporal support is moved outside the DBMS kernel, performance is anticipated to degrade. Statements must be parsed *twice*, since the temporal server (or the temporal functionality interface library) must analyse the statement, and map it to a series of snapshot query language statements. Each snapshot query language statement will be subsequently processed by the snapshot DBMS parser. During query execution it is possible that intermediate results are moved between the temporal server (or the temporal functionality interface library) and the snapshot DBMS, so process switching and interprocess communication overheads are introduced. In these cases, the query execution time is also penalised with the cost of creating and dropping temporary tables. If creation of intermediate tables is required, *holding points* are introduced (i.e. no operation may proceed until the temporary table has been created and data insertion into it has been completed), thus advantages of pipeline execution are lost.

When snapshot databases are accessed, direct connections to the snapshot DBMS may be established, thus performance is not penalised. If, however, a temporal database is accessed, system dictionary must be queried to determine whether the tables involved in some operations have temporal semantics or not.

Finally, if the temporal server approach is followed, then data must be moved from the snapshot DBMS kernel to the temporal server and subsequently forwarded to the client application, increasing the time needed for interprocess communication and introducing an additional process switch.

4.8 Resource consumption.

If temporal support is integrated within the snapshot DBMS kernel, then resource consumption increment is minimal: only the size of the DBMS kernel changes, to accommodate the extra code implementing the temporal operations, and the size of the DBMS interface library will increase, if extra calls are provided. Only one connection needs to be established per application, and only final results need to be communicated between the extended DBMS kernel and the client application.

If the *temporal server* approach is used, then the size of the DBMS kernel is not increased, but an additional process is introduced, which includes the code implementing the temporal operations plus the snapshot DBMS interface library. At the side of the client application, the snapshot DBMS interface library is replaced by the TDBMS interface library. If it is chosen to limit the complexity of the temporal server by starting one new temporal server process per application connection (see section 4.5), then more system memory is used up and more administrative overhead will be imposed to the operating system, in order to manage the new set of processes. In all cases, however, two connections must be established per application, the first one between the application and the temporal server and the second one linking the temporal server to the snapshot DBMS. Multiplexing connections between the client applications and the temporal server to a smaller number of connections between the temporal server and the snapshot DBMS is not a good idea, since this would render useless the transaction support and locking features of the snapshot DBMS. Consider the case that two applications A_1 and A_2 connect to the temporal server, and the temporal server chooses to multiplex incoming requests through a single connection *C*. If both applications issue the same request, e.g.

```
UPDATE Employee SET Salary = Salary * 2
WHERE EmpId = 'e1'
```

it is clear that one of them should wait until the other commits or aborts. However, since the connections are multiplexed, as far as the snapshot DBMS is concerned both requests originate from the same source (i.e. the connection *C*), so the first update will not block the second. Furthermore, if at a later stage one of the applications commits or aborts its transactions, then both statements will be committed or aborted, respectively. Thus, multiplexing will lead to the need for implementing locking and transaction support features into the temporal server, increasing its complexity even further. Additionally, multiplexing different connections between the client applications and the temporal server to one connection to the snapshot DBMS makes the *fairness* goal hard to achieve (if some request is pending on a connection to the snapshot DBMS then it must be completed before another request can be issued through the same connection).

Finally, if temporal support is incorporated within the client application, then the size of each client application is increased, since the executable will include both the temporal functionality interface library and the snapshot

Design objective \ Design approach	Full Integration with the Snapshot DBMS	Introduction of a Temporal Server	Extensions built into the client application
Full temporal functionality	Yes	Affects performance	Affects performance
Snapshot DBMS compatibility	Full with no extra work	Requires extra work	Requires extra work
Implementability	Depends on DBMS and source code availability	High	High
Snapshot DBMS facilities	Available	May need to be recoded	May need to be recoded
Implementation simplicity	Medium	Low	High
Data integrity	Guaranteed	Ensured by techniques	Jeopardised
Performance	High	Low (many overheads)	Medium
Resource consumption	Low	High (memory, processes and connections)	Medium (memory)
Portability	Non-portable	Portable	Portable
Additional features	Readily available	Requires extra work	Requires extra work

Figure 9 - Evaluation of the design approaches

DBMS interface library, leading thus to increased memory consumption. Only one connection needs to be established between the client application and the snapshot DBMS, and the number of processes within the system is not affected.

4.9 Portability across hardware/software platforms.

Portability of the extensions across hardware and software platforms is hard to achieve if the extensions are built within the DBMS kernel, since kernel-specific (and possibly hardware-specific) features will be used (e.g. data formats, inter-module communication mechanisms). If the extensions are implemented outside the DBMS kernel using industry standards (e.g. embedded SQL for communicating with the snapshot DBMS), then porting to other hardware/software platforms will require minimal changes to the source code, thus the *temporal server* and *client integration* approaches appear to be more portable.

4.10 Additional features.

If the temporal extensions are implemented within the snapshot DBMS kernel, then the extended DBMS will support all the language extensions offered by the snapshot DBMS, since both the parser and the execution module are already capable of handling them. The locking, concurrency control and recovery mechanisms offered by the snapshot DBMS will be available to the extended DBMS as well.

In the case that extensions are implemented outside the DBMS kernel, the temporal module parser must recognise the DBMS's syntactical extensions and take the appropriate actions. The locking, concurrency control and recovery mechanisms offered by the snapshot DBMS may be also used, although special techniques may need to be employed; in [21] such techniques are described. According to these techniques, two connections to the snapshot DBMS are required per client application, thus the overall resource consumption increases even more.

5. Conclusions

In this paper we presented three design approaches to the implementation of temporal systems on top of existing snapshot DBMSs. The degree to which each design approach fulfils a number of design objectives was examined. Figure 9 summarises the results of our survey.

The main advantages of the first approach (integrating temporal support into the DBMS kernel) are high performance, complete temporal functionality and exploitation of all the features offered by the snapshot DBMS. However, it is not easily implementable without access to the source code, and cannot be ported to different platforms without major changes. The *temporal server* approach, although elegant, presents some major problems, concerning implementation simplicity, performance and resource consumption. Finally, providing temporal support within the client application may deliver acceptable performance, but it is possible that some parts of the snapshot DBMS will need to be recoded (or complete temporal functionality sacrificed) and data integrity is jeopardised.

6. References

[1] I. Ahn and R. Snodgrass, *Performance Evaluation of a Temporal Database Management System*, Proceedings of the International Conference on Management of Data, 1986.

[2] M. Böhlen, *ChronoLog 4.0 Reference Manual*, Institute for Electronic Systems, Aalborg University, 1995.

[3] M. Böhlen, *Temporal Database System Implementations*, Department of Mathematics and Computer Science, Aalborg University, 1995.

[4] M. Böhlen, *TimeDB Software Documentation*, Department of Mathematics and Computer Science, Aalborg University, 1995.

[5] C.Combi, Francesco Pincrioli, M. Cavallaro and G. Cucci, *Querying Temporal Clinical Databases with Different Time Granularities: The GCH-OSQL Language*, Ninteenth Annual Symposium on Computer Applications in Medical Care, Philadelphia, 1995.

[6] C. J. Date, *An introduction to database systems*, Vol. II. Addison-Wesley Publishing Company, 1985.

[7] D.H. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J.W. Davis, W, Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngbeak, B. Mahbod, M. A. Neimat, T. Risch, M. C. Shan and W. K. Wilkinson, *Overview of the IRIS DBMS*, in "Object Oriented Concepts, Databases and Applications", W. Kim and F. Lochovsky (eds), ACM Press, 1989.

[8] Ingres Corporation, Ingres Object Management Extension User Guide for the UNIX and VMS Operating System Release 0.3, November 1989.

[9] W. Kim, N. Ballou, H. T. Chou, J. F. Garza, D. Woelk, *Features of the ORION Object-Oriented Database System*, in "Object Oriented Concepts, Databases and Applications", W. Kim and F. Lochovsky (eds), ACM Press, 1989.

[10] A. M. Lister and R. D. Eager, *Fundamentals of Operating Systems*, McMillan Education Ltd., 1988.

[11] O2 Technology, *O_2C Beginner's Guide*, March 1995.

[12] ORACLE Corporation, *SQL Language Reference Manual (for version 7.0)*, 1993.

[13] ORES Project (ESPRIT III P7224), *Deliverable D2: Specification of Valid Time SQL*, edited by 01 Pliroforiki, Agricultural University of Athens and University of Athens, 1993.

[14] ORES Project (ESPRIT III P7224), *Deliverable D4: Implementation of Valid Time SQL*, edited by 01 Pliroforiki, University of Athens, Information Dynamics and Agricultural University of Athens, April 1994.

[15] D. Schmidt, A. K. Dittrich, W. Dreyer and R. Marti, *Time Series, a Neglected Issue in Temporal Database Research?*, Proceedings of the International Workshop on Temporal Databases, Zurich, September 1995.

[16] R. Snodgrass, *The Temporal Query Language TQUEL* in *ACM Transactions on Database Systems*, vol. 12, no. 2, July 1987, pp. 247-298.

[17] Sybase Inc., *Transact SQL user's guide (for release 10)*, 1994.

[18] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall Inc., 1992.

[19] B. Theodoulidis, A. Ait-Braham, G. Karvelis, *The ORES Temporal DBMS and the ERT-SQL Query Language*, Proceedings of the 5th International Conference on Database and Expert System Applications, Athens 1994.

[20] The TSQL2 Language Design Committee (R. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. Dyreson, R. Elmasri, F. Grandi, C. Jensen, W. Kafer, N. Kline, K. Kulkarni, T. Leung, N. Lorentzos, J.Roddick, A. Segev, M. Soo, S. Sripada), *The TSQL2 Temporal Query Language*, ed. R. Snodgrass, pub. Klower, 1995.

[21] C. Vassilakis, N. Lorentzos and P. Georgiadis, *Transaction Support in a Temporal DBMS*, Proceedings of the International Workshop on Temporal Databases, Zurich, September 1995.

[22] G. Wuu and U. Dayal, *A Uniform Model for Temporal Object-Oriented Databases*, Proceedings of the International Conference on Data Engineering, Tempe, Arizona.