39

Stratum Approaches to Temporal DBMS Implementation

Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass

Previous approaches to implementing temporal DBMSs have assumed that a temporal DBMS must be built from scratch, employing an integrated architecture and using new temporal implementation techniques such as temporal indexes and join algorithms. However, this is a very large and time-consuming task. This paper explores approaches to implementing a temporal DBMS as a stratum on top of an existing non-temporal DBMS, rendering implementation more feasible by reusing much of the functionality of the underlying conventional DBMS. More specifically, the paper introduces three 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 paper 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.

Keywords: temporal databases, database systems architectures, database interfaces, legacy systems

1 Introduction

Most database application manage temporal data [9, 16], such as time and date of withdrawal of money from an ATM machine, closing values of stocks on the stock exchange, or the periods over which employees are associated with projects.

Temporal data management is currently being (re-)implemented in each individual application in an ad-hoc manner, with little support from the DBMS. Writing temporal queries in SQL-92 can be very tedious, and it has been shown that a temporal SQL can significantly reduce the amount and difficult of code needed to express temporal queries [18, Ch. 1]. Temporal data management applications could thus benefit substantially from built-in support.

Temporal databases extend conventional databases by associating timestamps with facts. Implementing a temporal database management system (temporal DBMS) on top of a conventional DBMS has generally not been pursued because it cannot take advantage of well-known temporal implementations techniques such as temporal indexes (e.g., [12]), temporal storage structures (e.g., [1]), and temporal join (e.g., [21]) and coalescing algorithms [5]. Further, it seems that there has been an implicit assumption (e.g., in [17]) that the performance of temporal DBMSs should be similar to that of conventional DBMSs, even when a temporal DBMS manages multiple versions of data and a conventional DBMS manages only one version. However, building a complete DBMS from bottom up is a very large task that may only be accomplished by the major DBMS vendors.

With the general goal of providing built-in support for time-varying data without having to construct a temporal DBMS from scratch, we explore in this paper how a temporal DBMS can be implemented in a stratum on top of an existing, conventional DBMS. The idea is to reuse the functionality of existing DBMS technology. The limitation of building on top of an existing DBMS is that it is not possible to modify existing core DBMS functionality, e.g., the data manager, the query processor component, and the transaction manager.

While the stratum approach may bring built-in temporal support in the DBMS to application programmers, the approach also provides a means of experimenting with new temporal database technologies. The approach makes it feasible for research teams to implement and experiment with temporal query languages, and it also allows some experimentation with parts of the back end of a database, e.g., query evaluation and special temporal operator implementations [5]. The experiences gained from using the stratum approach can be helpful when realizing the long-term goal of building temporal functionality directly into the DBMS.

We list eight criteria that a stratum should satisfy. Among others, the criteria include these: no changes to the underlying DBMS, retention of all desired properties of the DBMS, minimal impact on middleware. We then define three meta-architectures to building a stratum, namely (a) imposing a stratum directly,

(b) using middleware as the stratum (e.g., ODBC [13]), and (c) using a preprocessor. Each overall architecture captures several specific architectures, which are discussed in turn. We classify existing systems according to their architecture, including the temporal DBMSs listed in a recent survey [4]. The specific architectures are evaluated against the eight criteria.

The paper concludes that a stratum approach makes it possible to implement a temporal DBMS with reasonable resources. It will take years before an integrated architecture will become available. In the meantime, a stratum approach can be used. In addition, a stratum is not necessarily a unintelligent converter—new temporal functionally can be implemented in a stratum.

The paper is organized as follows. Section 2 discusses the general idea of a stratum and lists our evaluation criteria for stratum implementations of a temporal DBMS. A total of 15 specific stratum architectures, partitioned into three meta architectures, and their current use are explored in Sections 3 and 4, respectively. In Section 5, we compare the specific architectures to the criteria. Related work is the topic of Section 6, and Section 7 summarizes the paper.

2 The Stratum Approach

This section describes the general idea of a stratum approach, it considers how the approach applies to temporal databases, and it lists our design criteria for a temporal stratum.

2.1 The Stratum Architecture

The general idea of a stratum architecture is illustrated in Figure 1, where the downward arrows denote a flow of queries, and the upward arrows denote a flow of data. All boxes denote software components. The round boxes denote components that we can alter, and the square boxes denote components we cannot alter, i.e., black-boxes. There are three levels in the stratum approach. The application level consists of the applications that access the DBMS. At the stratum level, the stratum is implemented as an interface to the DBMS. Finally, at the representational level, we have the DBMS where the data is actually stored.

In the stratum approach, the database applications are not directly connected to the DBMS. All communication between the applications and the DBMS is interposed by a stratum. There are two important potential advantages of using a stratum. First, it is possible to provide applications with a different data model than what is actually implemented by the DBMS. Second, a new data model implemented in a stratum does not have to be supplied by the DBMS vendor.

When the stratum approach is applied to temporal databases, the idea is to convert the conventional DBMS, which supports SQL-92, to a temporal DBMS, which supports some temporal SQL. The applications send temporal queries to the

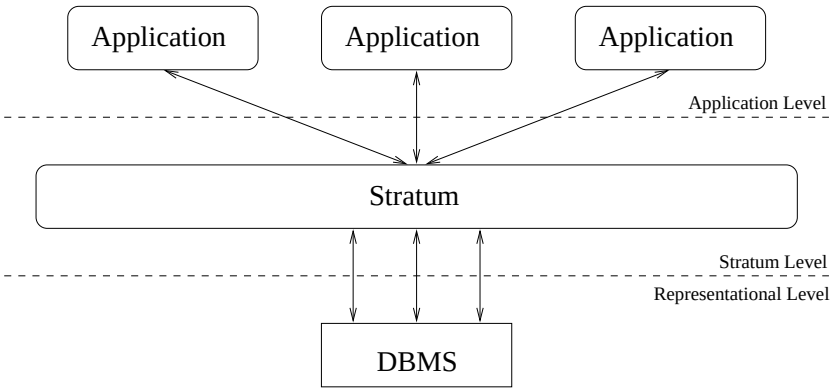

Figure 1: The Stratum Approach

temporal DBMS. The queries are received by the stratum, are converted into SQL-92 queries, which, in turn, are sent to the DBMS (in [6] it has been shown that all temporal queries can be converted to equivalent SQL-92 queries).

The result from the DBMS is returned to the stratum, which may do some processing of the data before is is passed to the applications. The purpose of the stratum is to make the conventional DBMS look like a DBMS supporting a temporal data model from the applications' point of view, as done, e.g., in [2, 7, 22, 27].

We restrict our attention to considering only new applications that may exploit the built-in temporal support. We do not consider the (orthogonal) problem of converting legacy applications with built in ad-hoc temporal support to applications using the temporal support implemented in the stratum.

2.2 Design Criteria for the Stratum Approach

In evaluating a stratum-implemented temporal DBMS, we stress the set of eight design criteria introduced next. The criteria are used in Section 5 to evaluate the different stratum architectures.

No modifications to the underlying DBMS are required The DBMS is used entirely as a black-box by the stratum. From the DBMS's point of view, the stratum is an application. The stratum uses only the DBMS's, or a middleware's, call level interface (CLI) and does not rely on the DBMS being extended with any temporal functionality. Because the stratum encapsulates the DBMS entirely, it is the only application that uses the DBMS directly. It is important that the stratum does not require the DBMS to be modified because we do not have the source code for the DBMS available.

Minimal impact on middleware The stratum may not use the DBMS's native CLI, but may instead use a generic API, e.g., ODBC [13]. We allow changes to this middleware, which can be used in the implementation of the stratum (to be discussed in Section 3.2) because generic APIs are open standards with their source

code available. An example can be to change the middleware to initiate a temporal SQL-to-SQL-92 conversion. The criterion on middleware is more flexible than the criterion on the DBMS because we do not assume we have the specification or the source code for the DBMS. Minimal impact on middleware is important to avoid side effects on existing applications.

Independence of applications The stratum implementation should encapsulate the DBMS for all applications. Applications implemented using the DBMS directly, e.g., via its native CLI, and applications using the DBMS indirectly, e.g., via a library, should all see the data model exposed by the stratum. If applications do not see the same data model, several versions of new applications must be implemented, and existing applications may be affected by the addition of time attributes to tables they use.

Maximum reuse of existing technology We want a thin stratum and therefore want to reuse as much of the functionality of the underlying DBMS as possible. We do not want to implement functionality already in the DBMS, e.g., the log and the transaction managers. Only functionality not found in the DBMS should be implemented in the stratum. The motivation for maximum reuse and a thin stratum is that limited resources are available for implementing the stratum.

Gradual availability of temporal functionality Again, because we assume limited resources and because an early return on the resources invested in the development of the temporal DBMS is desirable, it should be possible to make new temporal functionality available in a stepwise fashion. This provides a foundation for early availability of a working temporal DBMS with functionality that may increase gradually. Gradual availability is important to be able to demonstrate and evaluate temporal functionality.

Retention of desired properties of the underlying DBMS The underlying DBMS satisfies core database properties, e.g., the ACID properties of transactions. We want to retain these properties in the stratum, so that applications are not adversely affected by a stratum being interposed. The criterion ensures that the functionality provided by the stratum is an extension of the functionality provided by the underlying DBMS. However, it also means that if the underlying DBMS does not ensure a certain database property, the stratum will not support it either.

Adequate Performance We define adequate performance as follows. First, legacy applications should have the same performance as before a stratum is interposed. Performance is essential to the acceptance of temporal functionality. We cannot require existing (legacy) applications to be rewritten because new applications are built that use temporal support. Second, temporal queries on temporal databases should be as fast as the corresponding SQL-92 queries on the corresponding "snapshot" databases with temporal data. Put differently, SQL-92 code, for temporal-data

access, generated by the stratum's temporal SQL-to-SQL conversion should be as fast as hand-optimized SQL-92 code for the same purpose. Otherwise, application programmers may not want to use the automatic converter.

DBMS independence The stratum should be independent of the underlying DBMS. This may be achieved by using standards, such as SQL-92. It is also desirable that the techniques used in the implementation of the stratum be generic. As an example, we want to avoid that the temporal SQL-to-SQL conversion uses recursive SQL as found in IBM's DB2, but not in most other DBMSs.

The criteria are somewhat conflicting. As examples, the "independence of applications" criterion may conflict with the "adequate performance" criterion, and the "maximum reuse of existing technology" criterion may conflict with the "DBMS independence" criterion. The stratum implementor must consider these trade-offs.

Several observations are in order for a stratum that fulfills all the criteria. First, no legacy application that now uses the stratum was affected when the stratum was introduced (this assumes that the temporal SQL is upward compatible with SQL-92). They work as before and have the same performance. However, legacy applications not using the stratum will be affected if table they use are altered to support time.

Second, it is not possible to encapsulate the DBMS from the DBA's point of view. The DBA must be aware that, e.g., tables have been extended with time attributes to implement the built-in support for time offered by the stratum. Third, all update statements on temporal tables must be performed via the stratum if integrity constraints specified in the stratum are to be enforced. Alternatively, the stratum must rely on the integrity constraint mechanisms of the DBMS to implement new temporal constraints. Otherwise, it may be possible to update a temporal table to an inconsistent state, by circumventing the stratum. Finally, to make it possible for the stratum to do semantic checking of temporal SQL queries, all DDL statements altering tables to support time dimensions must be executed via the stratum.

3 Stratum Implementation Approaches

The next step is to explore how a stratum may be implemented. The outset is the general architecture from Figure 1. We assume that we have a set of applications that use temporal SQL, but that we do not have a temporal DBMS. Therefore, we simulate a temporal DBMS by using a conventional DBMS and interposing a stratum between the applications and the conventional DBMS.

The stratum can be implemented in different positions, leading to the following three overall architectures, each of which is explored in more detail in the sequel.

- Interposing a stratum directly between the applications and the conventional DBMS.
- Interposing a stratum in middleware (e.g., ODBC) between the applications and the conventional DBMS.
- Interposing a stratum using a preprocessor software component.

In the subsequent discussions of the three architectures, we will only consider applications where an API is used to communicate with the DBMS. This is a general approach to accessing a DBMS.

The discussions and figures use sample specific APIs, e.g., the DBMS-specific APIs for the DB2, Oracle, and Sybase DBMSs. Different specific DBMSs are used simply to make the discussion easier to follow, and we do not investigate the differences between, e.g., between the DB2 and Sybase APIs—from our point of view, they are simply representatives of DBMS-specific APIs. Similarly, we use the ODBC API [13] simply as a representative of any generic API (because it is the best documented such one). We could have used other generic APIs such as the JDBC API [10] or the Perl DBI API [3].

3.1 Interposing a Stratum Directly

Interposing a stratum directly between the applications and the DBMS is illustrated in Figure 2. As for Figure 1, upward and downward arrows denote the flow of queries and data, respectively. The round boxes and the square boxes are software components that we can and cannot alter, respectively. The dashed lines show the input interface and the output interface of the stratum.

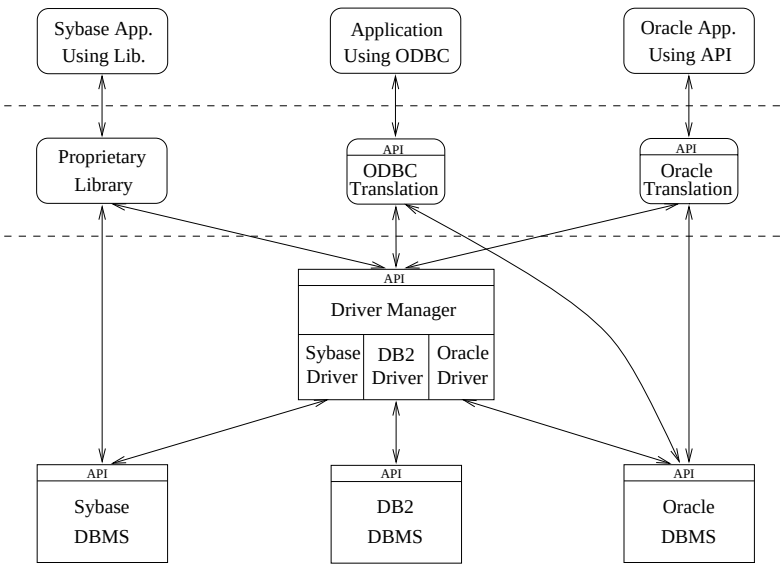

Figure 2: Interposing a Stratum Directly

Before the stratum was interposed in Figure 2, the ODBC Translation and Oracle Translation components did not exist. Further, the proprietary library was

not temporally enhanced. The applications were linked with the Proprietary Library, the ODBC driver manager, or the Oracle API.

After the stratum is interposed, the API calls made by the applications are intercepted (the ODBC and Oracle examples in Figure 2). The temporal-SQL code in the call is translated to SQL-92 code, and the stratum calls a DBMS or the driver manager at the representational level with this code.

When a stratum is interposed in a proprietary library, as shown in the Sybase example, we will assume that no temporal-SQL code is passed as a parameter, but that the library implements high-level functions specific to the database being managed. For example, if an employee table is present, the library may implement a function *Create_Employee*(<*parameters*>) that creates a new employee, specified by the *parameters*, by inserting a tuple into the employee table in the underlying DBMS. Note that in the proprietary-library approach, no SQL-92 code is passed as a parameter. This is in contrast to the API approach, and it gives the two approaches different properties.

To implement a stratum by interposing it directly, the stratum must support an API (or library interface) that is a superset of the API (or library) the applications used before the stratum was interposed. We next turn to discussing the examples in Figure 2 in greater detail.

The Sybase application to the left in Figure 2 is an example of an application that uses a proprietary library. Before the stratum was interposed, the Sybase application used the proprietary library, which, in turn, used the Sybase DBMS. After interposing the stratum in the library, we do not want to alter the possibly many applications that use this library. Instead we change the implementation of the library. We retain, or strictly extend, the library's interface to the applications. We have the flexibility in the stratum to either make it use a DBMS-specific API or a generic API. This flexibility is indicated in the figure by the arrows from the proprietary library at the stratum level to the Sybase API and to the Driver Manager API at the representational level.

In the middle of Figure 2, we have an example of an ODBC application which, before the stratum was interposed, was linked to the ODBC driver manager. After the stratum is interposed, the application is connected to a stratum ODBC driver manager component. This component must comply fully with the ODBC API specification. When the ODBC application connects to a DBMS (now via the stratum), the stratum converts the arguments passed, if necessary. Again, we have the flexibility in the stratum to either map the input API calls to a generic API or a DBMS-specific API.

The example to the right in Figure 2 shows an Oracle application that used the Oracle-specific call-level interface before the stratum was interposed. After the stratum is interposed the application uses the component at the stratum level that complies with the Oracle call-level interface. The Oracle call-level interface

component in the stratum has the same functionality as the stratum ODBC driver manager, converting temporal SQL to SQL-92 and forwarding the function calls.

Studying the input and output APIs of the stratum components, it can be seen that the six combinations shown in Figure 3 exhaust the possibilities. Interposing a stratum directly between the applications and the DBMSs or driver manager thus yields a total of six specific architectures for implementing a stratum.

$$\left\{\begin{array}{c}\textbf{Input Interface}\\ \text{Proprietary Library}\\ \text{Specific API}\\ \text{Generic API}\end{array}\right\} \times \left\{\begin{array}{c}\textbf{Output Interface}\\ \text{Specific API}\\ \text{Generic API}\end{array}\right\}$$

Figure 3: Interposed Stratum Interfaces

3.2 Using Middleware as the Stratum

Next, we turn to the use of middleware for implementing a stratum. Again note that we use ODBC as our prototypical middleware only because it is a mature and well-documented interface. Other types of middleware such as JDBC and DBI are based on ODBC and resemble it. The idea of using ODBC as the stratum is shown in Figure 4. The dashed arrows inside the driver manager indicate different paths that can be taken and are explained further shortly.

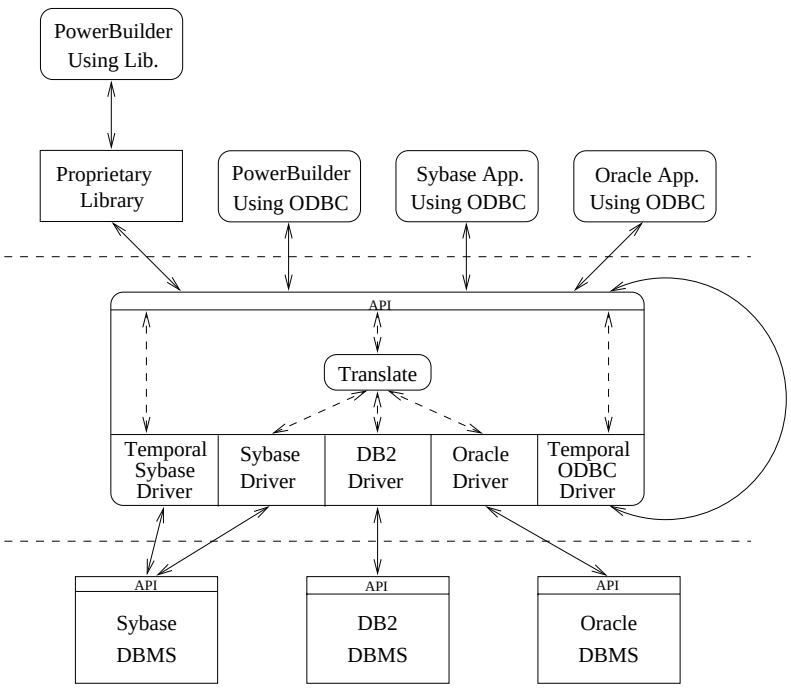

Figure 4: Using ODBC as the Stratum

Both before and after the stratum is interposed in Figure 4, the applications communicate with the ODBC driver manager.

The stratum can be implemented in two places using a generic API as ODBC. First, the stratum can be implemented within the driver manager. This is indicated with the component "Translate" in the figure. Second, the stratum can be implemented entirely in an ODBC driver. This is indicated with the components "Temporal Sybase Driver" and "Temporal ODBC Driver."

When implementing the stratum within the driver manager, the driver manager itself is extended by a component that translates temporal SQL to SQL-92. When an application makes an ODBC call, the driver manager normally just forwards the call (assuming a connection has been established). With the extra temporal SQL-to-SQL-92 translation component added, the driver manager checks whether the arguments in the call contain temporal SQL that must be translated, performs the translation if necessary, and then forwards the call and translated arguments to the appropriate "plain" ODBC-driver. By "plain" we mean an off-the-shelf ODBC driver. In Figure 4 the three ODBC drivers in the middle, i.e., the Sybase, DB2, and Oracle drivers, are the "plain" ODBC-drivers. With this approach, the paths taken within the driver manager are from the API through "Translate" to a "plain" driver.

The other alternative when using ODBC is to implement the stratum entirely in an ODBC-driver. The driver manager is then not altered. Instead, the translation is done in "temporal" ODBC drivers. In Figure 4, we show two types of such a "temporal" driver. To the left, there is a "Temporal Sybase Driver," and to the right, there is a "Temporal ODBC Driver." We discuss each in turn.

Using a DBMS-specific "temporal" ODBC driver, as exemplified by the "Temporal Sybase Driver," when an application makes an ODBC call, the driver manager performs the same actions as for a "plain" ODBC driver: it simply forwards the call and arguments. In the "temporal" driver, temporal SQL is converted to SQL-92, and the DBMS is queried.

When using a generic "temporal" ODBC driver (i.e., the "Temporal ODBC Driver"), the driver manager forwards the call and the arguments to the driver. The generic "temporal" driver converts temporal SQL to SQL-92. It does not forward the call directly to a specific DBMS, but instead reconnects to the ODBC driver manager. This second connection uses the "plain" driver for the appropriate specific DBMS. The reconnection to the driver manager is possible because an ODBC driver can function as an application.

The combinations of input and output from the stratum components using the ODBC driver architecture as the stratum are shown in Figure 5. The architecture provides a total of three specific stratum architectures: (1) A generic API/specific API architecture obtained by implementing a DBMS specific "temporal" ODBC driver; (2) a generic API/generic API architecture realized by implementing a generic

"temporal" ODBC driver; and (3) a generic API/specific API achieved by adding a translation component to the driver manager. Note that the first and third architectures, while different, have identical input and output interface.

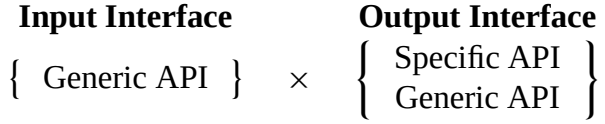

Figure 5: ODBC Stratum Interfaces

3.3 Preprocessing

The third overall architecture for implementing a stratum is to use a preprocessor. The idea is shown in Figure 6, where the dashed arrows show the flow of program code. A stratum implemented in a preprocessor does the conversion at compile time, as opposed to the two overall architectures discussed previously, where the stratum does the conversion at runtime. The preprocessor architecture is therefore only possible for applications that do not generate temporal SQL code at runtime, e.g., it cannot be used for applications handling ad-hoc queries against a temporal DBMS. The preprocessor idea is widely used to embed SQL code into a host language such as C or COBOL.

There is no difference between the architectures before and after the preprocessor stratum is interposed. The source code of the "temporal" applications is converted using a preprocessor, being compiled into an executable. The only difference is that the preprocessors are extended. First, a preprocessor converts temporal-SQL code to SQL-92 code. Next, the SQL-92 code is run through the preprocessor supplied by the DBMS vendor. We do not show the DBMS vendors' preprocessors in Figure 6; rather, the two preprocessing steps are both done in the preprocessor components at the stratum level.

As an example consider the Sybase application using the Sybase API. Before the temporally-enhanced application code is used, it is run through the "temporal" Sybase preprocessor at the stratum level. This converts the temporal SQL in queries to SQL-92 and may convert the API used to being either the Sybase-specific API or the generic ODBC API.

The different type of input and output from the stratum components are the same as for interposing a stratum directly as shown in Figure 3, leaving six specific architectures for building a temporal DBMS in a preprocessor.

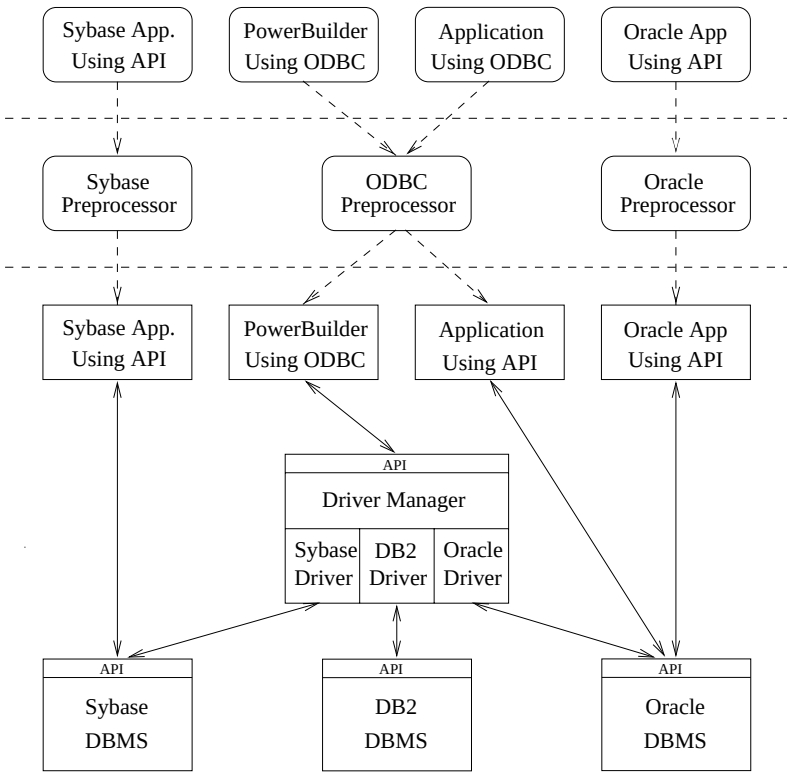

Figure 6: The Preprocessor Architecture

4 Applications of the Different Architectures

In this section we discuss the utility of the different stratum architectures and, when possible, provide concrete examples of their use. Specifically, we have tried to categorize all the existing temporal DBMS implementations found in a recent survey [4] that use the stratum approach. Where we have not been able to find an example relating to temporal DBMS implementation, we discuss non-temporal examples.

4.1 Interposing a Stratum Directly

As shown in Figure 3, there are six combinations of input and output from the stratum components. The resulting six different architectures will be discussed in turn.

The *proprietary library/specific API* architecture can be used if a site has a large number of applications using a single DBMS and wants to change the underlying DBMS to a temporal DBMS. The applications are targeted towards a specific DBMS that is considered a strategic component. There is no reason for porting the library to support different DBMSs.

The advantage of using a single DBMS is that it is possible to use all the features of the DBMS. It may have "that one essential feature," providing the reason

why this specific DBMS is used. The feature can be a hardware feature, e.g., the DBMS runs on an IBM mainframe, or a software feature, e.g., it supports data blades.

We assume this architecture can be used, e.g, for companies that are extensively using one DBMS in their applications, e.g., banks and insurance and telephone companies. The DBMS may be a part of a high-performance mission-critical transaction processing system. This architecture has been used by the Swiss Regional Banks to implement a bitemporal DBMS library on top of Oracle 7.3 [2].

The *proprietary library/generic API* architecture can be used if a company has an existing library targeted towards a specific DBMS which is used by a large set of applications. However, the company now wants to add temporal support to the DBMS. Further, the company gradually wants to move from a closed environment to an open one. Instead of changing all the applications, the proprietary library is reimplemented to support the mapping from temporal SQL to SQL-92. To make the library open, the reimplementation makes connections to DBMSs via a generic API, e.g., ODBC instead of via a DBMS-specific API.

The Perl 5 ODBC module [14] is an example of this architecture. The module makes it possible to access the C-language ODBC API from Perl programs. Note that the Perl ODBC module is an example of a library that is schema independent. The module is not built to support a specific set of applications, but targets a generic API, making it applicable to any database. In contrast, the Swiss bank proprietary library/specific API example mentioned above is a database-specific, or schema-dependent, library where the library implementor is aware of the underlying schema of the DBMS targeted.

The *specific API/specific API* architecture can be used where a large set of applications use only one DBMS. The architecture is more general than using the proprietary library/specific API architecture because the specific API/specific API architecture is schema independent. The architecture converts the DBMS-specific API calls, and not only the calls to the proprietary library. It is likely to be used for the same reasons as the proprietary library/specific API architecture: a specific DBMS is a strategic product, and all the features of the specific DBMS can be utilized in the mapping, possibly leading to better performance. The architecture is also useful for custom-built applications where the DBMS to be used is known at design time, and where this DBMS is used throughout the lifetime of the applications.

The architecture can be used by the major DBMS vendors to extend their database products with temporal support. Different research prototypes have added temporal support to existing DBMSs by using this architecture, e.g., Chronolog, HDBMS, TimeDB, and T-Square DBMS [4]. These are all examples of temporal extensions of a specific conventional DBMS. The prototypes are not implemented as an API conversion. Instead, they convert a temporal SQL dialect to SQL-92

(in fact, to vendor-specific SQL-92 dialects) and then query the SQL–92 database. However, they all adopt the the overall idea of the specific API/specific API architecture.

The *specific API/generic API* architecture can be used if the source code from an application generator tool contains DBMS-specific API-calls and the user prefers the application to access another DBMS, e.g., via ODBC.

The *generic API/specific API* architecture can be used if a set of ODBC applications have a performance problem and the applications are only connected to one specific DBMS. By interposing a stratum that connects directly to the DBMS instead of using the ODBC driver manager, it may be possible to enhance the performance of the applications by moving temporal functionality from the stratum into the DBMS, e.g., as stored procedures.

The *generic API/generic API* architecture can be used where a set of ODBC-enabled applications are connected to several DBMSs, each of which is updated to support temporal data. When the temporal SQL-to-SQL-92 conversion occurs before the driver manager, all DBMSs previously accessed can still be accessed without building a converter for each specific DBMS.

4.2 Using Middleware as a Stratum

For this type of architecture, the combinations of input and output to the stratum level are shown in Figure 5.

The *generic API/specific API* architecture is the normal way of using ODBC. A set of applications are using a DBMS which is enhanced to support temporal data management. To enable the existing applications to use the enhanced DBMS, all the conversion from temporal SQL to SQL-92 is done in the DBMS-specific driver.

An example is the NNODBC driver [11], which allows users to query an NNTP news server with a subset of SQL-92 via ODBC. The NNODBC driver encapsulates the news server with a relational interface, i.e., makes it look like a table from the driver manager's point of view. Another similar example is the flat-file ODBC driver [13] that allows users to query ASCII files via SQL.

The *generic API/generic API* architecture is useful when applications are connected to different DBMSs via a generic API, but there are no DBMS-specific drivers available for the DBMS to be used. However, there is a "temporal" driver, which bridges to a generic API for which a DBMS-specific driver exists.

An example of this architecture is the JDBC-ODBC bridge [10], which allows Java applications, using the generic JDBC API, to access databases via ODBC. As a difference from the example shown in Figure 4, not one but two different driver managers are used. The applications using the JDBC-ODBC bridge connect to the JDBC driver manager. The JDBC-ODBC driver then connects to the ODBC driver manager, which establishes a connection to a specific DBMS.

The *extended driver manager* architecture is an alternative to the generic API/generic API architecture. Extending the driver manager has the advantage that only a single software components has to altered to provide temporal support in multiple underlying DBMSs.

4.3 Preprocessor Stratum

The preprocessor approach is a simple one that is currently in wide use for permitting the embedding of SQL code in host language code, e.g., C/C++, Pascal, and COBOL code. Such host language code is run through a preprocessor before being compiled. The preprocessor converts the embedded SQL code into, e.g., function calls using a DBMS-specific API. The converted source code is then compiled. In the stratum approach, this scenario must be extended with a temporal SQL-to-SQL-92 conversion.

The combinations of input and output to the stratum level are shown in Figure 3. The main difference between interposing a stratum directly and using a preprocessor architecture is that the former does the conversion of temporal SQL to SQL (and possiblely between APIs) at runtime, whereas the latter does the conversion at compile time. For this reason, we omit the discussion of all six specific architectures and instead refer the reader to Section 4.1. However, we have the following comments on two of the specific architectures.

The *specific API/specific API* preprocessor architecture is highly relevant for DBMS vendors. As already mentioned, preprocessors are widely used; and a temporal preprocessor does not necessitate any changes to the underlying DBMS. However, it does require the DBMS vendor to define a temporal SQL. The *specific API/generic API* and the *generic API/generic API* architectures are of relevance to independent software houses that support more than one DBMS and are interested in a single product that is relevant to as many customers as possible. Again, a prerequisite is the specification of a temporal SQL.

5 Comparison of the Architectures

The following three subsections compare the 15 specific stratum architectures identified in Section 3 against the criteria introduced in Section 2.2. We use the following notation for evaluating the architectures. A table field is empty if a criterion is not fulfilled. A check-mark ($\surd$) indicates that a criterion is fulfilled, and a check-mark-plus ($\surd^{+}$) indicates that a criterion is fulfilled to a higher degree than required. We use NA if a criterion is not applicable to the specific architecture.

5.1 Interposing a Stratum Directly

The six specific architectures for interposing a stratum directly are compared in Table 1. The criteria are listed as rows in the table in the order they were discussed in Section 2.2.

Input Interface	*Prop. Lib.*		*Specific*		*Generic*	
Output Interface	*Spec.*	*Gen.*	*Spec.*	*Gen.*	*Spec.*	*Gen.*
No DBMS Mods.	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Minimal Impact	NA	$\surd^+$	NA	$\surd^+$	NA	$\surd^+$
Indep. of Apps.			$\surd$	$\surd$	$\surd$	$\surd$
Reuse of Tech.	$\surd$	$\surd$	$\surd^+$	$\surd^+$	$\surd^+$	$\surd^+$
Gradual Avail.	$\surd^+$	$\surd^+$	$\surd$	$\surd$	$\surd$	$\surd$
Retention Props.	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Adequate Perf.	$\surd^+$	$\surd$	$\surd^+$	$\surd$	$\surd^+$	$\surd$
Indep. of DBMS		$\surd$		$\surd$		$\surd^+$

Table 1: Interposed Architectures

None of the architectures require modifications to the underlying DBMS. The stratum is an application that uses the DBMS; specifically, the stratum uses the public interface to a specific DBMS or a generic API. To implement the architectures that use a generic API as either the input or output interface, no modifications are required to the middleware. Because "no modifications" is the absolute minimum impact on the middleware, we give these architectures a check-mark-plus.

The two architectures that use a proprietary library as their input interface are not independent of applications. The applications have to call the proprietary library to use the new temporal functionality. Even if some some applications use the library, this does not rule out that other applications access the DBMS directly. And as mentioned in Section 2.2, exposing different data models to same database may cause problems. The remaining four architectures are independent of the applications because all calls to the input interface (an API) are interposed.

With respect to reuse of existing technology, all architectures are in compliance. However, the two architectures using a proprietary library as input interface require the library to be reimplemented. For this reason, we find that the architectures that use an API as input interface may reuse existing technology better. On the other hand, using a proprietary library as input interface may provide the best possible way of ensuring gradual availability of temporal functionality. Temporal functionality can be provided on a per-table basis. As time dimensions are added to tables, all the functions using tables must be updated. Using an API as the input interface requires more coding before application programmers can start using the temporal functionality, because these architectures are more general than the proprietary library architectures.

We assume that the architectures where the output interface is a specific API can achieve better performance than the architectures where the output interface is a generic API. The justification is that the former can be tuned to a specific DBMS, e.g., rely on stored procedures. The cost of better performance is that they become dependent on the DBMS, as shown in the last row in Table 1.

5.2 Using Middleware as a Stratum

The three specific architectures that use middleware as the stratum are compared in Table 2. The leftmost generic API/specific API architecture is the DBMS-specific "temporal" driver architecture. The rightmost generic API/specific API architecture is the architecture that alters the driver manager.

Input Interface	*Generic*		
Output Interface	*Spec.*	*Gen.*	*Spec.*
No DBMS Mods.	$\surd$	$\surd$	$\surd$
Minimal Impact	$\surd^{+}$	$\surd^{+}$	$\surd$
Indep. of Apps.	$\surd$	$\surd$	$\surd$
Reuse of Tech.	$\surd^{+}$	$\surd^{+}$	$\surd$
Gradual Avail.	$\surd$	$\surd$	$\surd$
Retention Props.	$\surd$	$\surd$	$\surd$
Adequate Perf.	$\surd^{+}$	$\surd$	$\surd$
Indep. of DBMS		$\surd^{+}$	$\surd$

Table 2: Middleware Architectures

As can been seen from Table 2, all architectures are DBMS independent—they only rely on additions to the middleware. Regarding their impact on the middleware, the two "temporal" driver approaches require no changes to the driver manager. The drivers are added to the driver manager as "plain" drivers. Altering the driver manager requires addition of software components to the middleware. The changes are likely to be isolated and do not require reimplementing the entire driver manager. Having to change the middleware, we find that this is a minimal impact.

All the architectures are independent of applications (the input interface is a generic API), can provide temporal functionality gradually, and retain the desired properties of the underlying DBMS. Regarding performance, the first architecture can be tuned to a specific DBMS. Again, the better performance is at the cost of DBMS independence. The tuning is not possible for the third architecture, even though it also uses a specific API as output interface. The DBMSs are accessed via "plain" ODBC drivers, which cannot be altered. However, the architecture becomes independent of the DBMS because multiple specific APIs can be used.

5.3 Preprocessor Stratum

The six specific architectures for the overall preprocessor architecture are compared in Table 3.

Input Interface	*Prop. Lib.*		*Specific*		*Generic*	
Output Interface	*Spec.*	*Gen.*	*Spec.*	*Gen.*	*Spec.*	*Gen.*
No DBMS Mods.	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Minimal Impact	NA	$\surd^+$	NA	$\surd^+$	NA	$\surd^+$
Indep. of Apps.			$\surd$	$\surd$	$\surd$	$\surd$
Reuse of Tech.	$\surd^+$	$\surd^+$	$\surd^+$	$\surd^+$	$\surd^+$	$\surd^+$
Gradual Avail.	$\surd^+$	$\surd^+$	$\surd$	$\surd$	$\surd$	$\surd$
Retention Props.	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Adequate Perf.	$\surd^+$	$\surd$	$\surd^+$	$\surd$	$\surd^+$	$\surd$
Indep. of DBMS		$\surd$		$\surd$		$\surd^+$

Table 3: Preprocessor Architectures

With respect to modifications to the DBMS, impact on middleware, and independence of applications, the preprocessor architectures are similar to their equivalent architectures (based on input and output interface) for imposing a stratum directly, as discussed in Section 5.1.

All the preprocessor architectures are very good for reusing existing technology. The preprocessor approach is widely used, so we assume DBMS vendors and software houses have experience with implementing preprocessors in general. Further, the preprocessor architectures make the coupling between the stratum and the DBMSs lower because there is no run-time interaction between the stratum and the DBMSs. The strata (preprocessors) are only used at compile-time, not at run-time. We also assume that because of their widespread use, many applications programmers are familiar with the concept of a DBMS preprocessor.

Regarding performance, we have rated the preprocessor architectures similar to the performance of the architectures when the stratum is interposed directly. However, we believe that the performance of the preprocessor architectures will be better because queries are optimized at compile time instead of at runtime. As before, we assume that performance and DBMS independence are inversely related for the architectures.

6 Related Work

The use of strata, or layers, is a general software design technique useful for decreasing the complexity of systems. The use of a layer can be found in several design patterns. The *Facade* design pattern [8] can be used to provide a high-level interface to subsystems. The Facade pattern is useful for layering the system and

can do work on its own, e.g., if the interface to the subsystems does not apply directly to the interface provided by the Facade. In the context of this paper, the Facade would then be the stratum and a specific DBMS would be a subsystem. Other types of layers, also called wrappers, can be found in the *Decorator* and the *Adaptor* design patterns [8].

An alternative to a stratum approach to building a temporal DBMS is the integrated architecture where a DBMS is built from scratch and the implementation incorporates temporal support. The Postgres DBMS [24, 25] is the most well-known example of such an architecture. It supports transaction time and so-called *time travel* in the query language PostQuel. The TempIS Temporal DBMS supports both valid and transaction time [15] and extends academic Ingres [23]. This system implements the TQuel temporal query language [17]. (The implementation of the TempIS Temporal DBMS is discontinued.) The TimeMultiCal is another temporally enhanced DBMS built from scratch [19]. It supports multiple calendars, but neither valid time nor transaction time. The T-Requiem system has an integrated architecture (for contact information, see [4]). This system extends a public domain DBMS (Requiem) with valid and transaction time support. The prototype is not publicly available.

The stratum approach has recently be used for implementing a temporal DBMS prototype, called TimeDB, which supports both valid time and transaction time [22]. It is built on top of the Oracle DBMS and supports the ATSQL2 temporal query language [20], a descendent of the TSQL2 [18] temporal query language. The Tiger prototype [7] is a close relative of TimeDB. It implements ATSQL [6] and can be tested online.

A mixture of an integrated and a stratum architecture is documented in [27]. Here, a temporal DBMS prototype supporting valid time is implemented partly on top of the Ingres DBMS and partly as an extension of the Ingres DBMS. The Ingres kernel is extended with support for an interval data type. The rest of the temporal functionality is built on top of the extended kernel.

Vassilakis et al. [28] have provided a survey of temporal DBMS architectures that complements the study provided by this paper. While both papers present surveys, there are fundamental differences. They describe and evaluate three architectures that provide built-in temporal support in a client-server environment; in contrast, we have explored 15 stratum architectures. More specifically, Vassilakis et al. do not assume that the underlying DBMS is a black-box, as is assumed here. Next, they assume that temporal SQL is not, and cannot be, translated to regular SQL. Not performing this translation leads to very different types of architectures. For example, query optimization must be partly done in the DBMS and partly in the stratum. Further, they assume that application may connect directly to the underlying DBMS, In contrast, we disallow direct access from applications to the

underlying DBMS because this may cause problems with respect to data integrity, as discussed in Section 2.2.

Finally, in [26] it is discussed how a temporal DBMS can be implemented on top of an existing system with a minimal effort. Several implementation techniques are covered.

7 Summary

Building a temporal DBMS from scratch is a daunting task, which may only be successfully taken on by the major DBMS vendors. To enable the efficient implementation of applications that may benefit from built-in support for time in the DBMS and to enable experimentation with a temporal DBMS, we have investigated how the task of building a temporal DBMS can be reduced by building on top of an existing conventional DBMS, maximally reusing its functionality.

A set of criteria for evaluating a stratum architecture is proposed. Three overall architectures to building a stratum are identified and fifteen specific architectures are discussed. We categorize the existing temporal DBMS implementations that we are aware of according to the specific architectures.

The specific architectures are then compared against our criteria. There is no best architecture. Which architecture is preferred depends on the situation where the stratum is to be used. Those who want temporal functionality available quickly can use a temporally enhanced library to provided temporal support. A library can also be tailored to a specific DBMS for maximum performance. The DBMS vendors can extend their products by, e.g., providing a temporally enhanced preprocessor or a stratum on top of the specific DBMS. DBMS vendors should make the temporal extension general, requiring more work compared with only extending a single library with temporal support.

We believe that the best short and medium term approach to building a temporal DBMS is to build on top of an existing conventional DBMS. This way, resources can be focussed on implementing new temporal functionality without having to reimplement existing functionality.

Acknowledgements

This research was supported in part by the Danish Technical Research Council through grant 9700780, by the National Science Foundation through grants IRI-9632569 and IRI-9202244, and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, contract no. FMRX-CT96-0056.

References

[1] I. Ahn and R. T. Snodgrass. *Partitioned Storage for Temporal Databases*. Information Systems, 13(4):369–391, 1988.

[2] R. Barnert and G. Schmutz *Die zeitbezogene Datenhaltung bei den Schweizer Regionalbanken*. Wirtschaftsinformatik, 39(1):45–53, 1997.

[3] T. Bunce et al. *Perl DBI API*. `www.hermetic.com/technologia/-DBI/`, Dec. 1997.

[4] M. H. Böhlen. *Temporal Database System Implementations.* SIGMOD Record, 24(4):53–60, 1995.

[5] M. H. Böhlen, M. D. Soo, and R. T. Snodgrass. *Coalescing in Temporal Databases*. VLDB Proceedings, pp. 180–191, 1996.

[6] M. H. Böhlen and C. S. Jensen. *A Seamless Integration of Time into SQL*. TR R-96–2049, Aalborg University, 1996.

[7] M. H. Böhlen. *The Tiger Bitemporal Database Prototype*. `www.cs.auc.dk/~tigeradm/`, Dec. 1997.

[8] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[10] G. Hamilton and R. Cattell. *JDBC: A Java SQL API version 1.20*. JavaSoft, 1997.

[11] K. Jin. *NNTP ODBC Driver*. `ftp.uu.net/pub/database/perl-interfaces/other/`, Dec. 1997.

[12] D. Lomet and B. Salzberg. *Access Methods for Multiversion Data*. ACM SIGMOD, pp. 315–324, 1989.

[13] Microsoft Corp. *Microsoft ODBC Software Development Kit Version 2.0*. Microsoft Press, 1994.

[14] D. Roth. *The Win32::ODBC Module*. `www.roth.net/odbc/`, Dec. 1997.

[15] K. Ryu. *A Temporal Database Management Main Memory Prototype*. TempIs TR 26. University of Arizona, 1991.

[16] A. R. Simon. *Strategic Database Technology: Management for the Year 2000*. Morgan Kaufmann Publishers, 1995.

[17] R. T. Snodgrass. *The Temporal Query Language TQuel*. ACM TODS, 12(2):247–298, 1987.

[18] R. T. Snodgrass (ed.). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[19] R. T. Snodgrass et al. *The MultiCal System*. `ftp.cs.arizona.edu/-tsql/multical/`, 1997.

[20] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. *Adding Valid Time to SQL/Temporal*. ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2, 1996.

[21] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. *Efficient Evaluation of the Valid-time Natural Join*. ICDE Proceedings, pp. 282–292, 1994.

[22] A. Steiner et al. *TimeDB*. `www.cs.auc.dk/general/DBS/tdb/-TimeCenter/Software/`, Dec. 1997.

[23] M. Stonebraker, E. Wong, and P. Kreps *The Design and Implementation of INGRES*. ACM TODS, 1(3):189–222, 1976.

[24] M. Stonebraker. *The Design of the Postgres Storage System*. VLDB Proceedings, pp. 289–300, 1987.

[25] M. Stonebraker, M. Hirohama, and L. A. Rowe. *The Implementation of Postgres*. IEEE TKDE, 2(1):125–142, 1990.

[26] K. Torp, C. S. Jensen, and M. H. Böhlen. *Layered Implementation of Temporal DBMSs—Concepts and Techniques*. TR R-96-2037, Aalborg University, 1996.

[27] C. Vassilakis, P. Georgiadis, and N. Lorentzos. *Transaction Support in a Temporal DBMS*. In Recent Advances in Temporal Databases, Springer-Verlag, pp. 255–271, 1995.

[28] C. Vassilakes, P. Geogiadis, and A. Sotiropoulou. *Comparative Study of Temporal DBMS Architectures*. 7th Intl. Workshop on Database and Expert Systems Applications Proceedings, pp. 153–164, 1996.