A BRIEF HISTORY OF TEMPORAL DATA MANAGEMENT

CONTENTS
Excluding Time from Databases: A Ubiquitous Paradigm  12
The 1980s  13
  Historical Databases  13
  History Tables  13
The 1990s  14
  Temporal Extensions to SQL  14
  Data Warehouses and Data Marts Come of Age  14
  The Inmon/Kimball Religious Wars  16
The 2000s  18
  Data Cubes  18
  Slowly Changing Dimensions  19
  Real-Time Data Warehouses  20
The Future of Databases: Seamless Access to Temporal Data  20
  Closing In on Seamless Access  22
Glossary References  24

Temporal data management is not a new development. From the earliest days of business data processing (as it was called back then), transactions were captured in a transaction log and the files and tables those transactions were applied to were periodically backed up. With those backups and logfiles, we could usually recreate what the data looked like at any point in time along either of the temporal dimensions we will be discussing. Indeed, together they contain all the “raw material” needed to support fully bi-temporal data management. As we will see in this chapter, what has changed about temporal data management, over the decades, is accessibility to temporal data. These days, it takes less effort to get to temporal data than it used to, and it takes less time as well. But significant additional progress is possible, and computer scientists are working on it. We are, too. Our work has led to this book, and to the software we are developing to implement its concepts.
We emphasize that the following history is not a history in any strict sense. It is more like our reflections on changes in methods of managing data which we have observed, as IT consultants, over the last quarter-century. It is our attempt to look back on those changes, and impose some order and structure on them. It does not fall short of history, in a strict sense, in attempting to impose order and structure. All historical narrative does that, no matter how purely descriptive it claims to be. Rather, it falls short in its reliance solely on personal reminiscence.

Excluding Time from Databases: A Ubiquitous Paradigm

In one sense, temporal data has been accorded only second-class status since the advent of computers and their use in managing business data. Neither database management systems (DBMSs) and the tables they manage, nor access methods and the files they manage, provide explicit mechanisms and structures to distinguish data about the past, present or future of the things we keep track of. Instead, unless developer-designed data structures and developer-written code is deployed, every object is represented by one and only one row in a table. If the row is there, the corresponding object is represented in our databases; otherwise it is not. If something about a represented object changes, the row is retrieved, updated and rewritten to reflect that change.

This focus on current data is reflected in a basic paradigm that has been used since we began managing data with computers. The paradigm is that of one data structure to represent a type of object or event, containing multiple other data structures, each representing an instance of an object or event of that type. Contained within the latter data structures are additional structures, each representing a property of the instance in question, or a relationship it has to another instance of the same type or (more usually) a different type.

This paradigm has manifested itself in such terminologies as (i) files, records, fields and pointers; (ii) tables, rows, columns and foreign keys; and (iii) classes, objects, slots and links. For the remainder of this book, we will use the table, row, column and foreign key terminology, although the concepts of uni-temporal and bi-temporal data management apply equally well to data managed by directly invoking access methods, to data managed with proprietary software, and to data managed with object-oriented structures and transformations.
The 1980s

Historical Databases

In the 80s, as disk storage costs plummeted, it was inevitable that someone would think to put the most recent backup files onto disk where it would be possible to access them without first restoring them from off-line storage media. After that, the next step was to realize that there was value, not just in having a particular set of backup data remain on-line, but also in having the ability to compare multiple backups of the same data, made at different points in time.

Each backup is a snapshot of a set of data of interest, and just as a movie film makes motion apparent while being itself a series of still images, so too a series of database snapshots can make change apparent while being itself a series of still images. Thus was born the concept of a data warehouse, whose originators were Barry Devlin and Paul Murphy.\(^1\) This concept introduced temporal data management at the database level (as opposed to the table, row or column levels), since data warehouses are entire databases devoted to historical data.

History Tables

On an architecturally smaller scale, IT developers were also beginning to design and implement several other ways of managing temporal data. One of them was the use of history tables, and another the use of version tables. In the former case, temporal data management is implemented at the table level in the sense that individual tables are the objects devoted to historical data, usually populated by triggers based on updates to the corresponding current tables. In the latter case, temporal data management is also implemented at the table level, but in this case historical and current data reside in the same table. In some cases, intrepid developers have even attempted to introduce temporal data management at the level of individual columns.

In addition, developers were also beginning to create on-line transaction tables by bringing collections of transactions back from off-line storage media, transactions that originally had been moved onto that media as soon as their current accounting periods were over. The difference between history tables and version tables, on the one hand, and transaction tables on the

\(^1\)See [1988, Devlin & Murphy]. The full citation may be found in the appendix Bibliographical Essay. The year is the year of publication, and entries in that appendix are organized by year of publication.
other hand, is that history and version tables record the state of objects at different times, whereas transaction tables record the events that change the states of those objects and, in particular, the relationships among them.

The 1990s

Temporal Extensions to SQL

By the early 90s, significant computer science research on bi-temporality had been completed. To the extent that word of these developments made its way into the business IT community, bi-temporality was understood as a distinction between logical time and physical time. Logical time, corresponding to what computer scientists called valid time, was generally referred to by IT professionals as effective time. It was understood to be that period of time, denoted by either a single date or by a pair of dates, during which the object represented by a row conformed to the description that row provided. The term “effective time” derives from the fact that for specific and non-overlapping periods of time, each of these rows is in effect as the representative of an object—as the authorized description of what the object is like during that specific period of time. As for physical time, it was understood to be a single date, the physical date on which the bi-temporal data is created.

This view was, in fact, either a misunderstanding of what the computer scientists were saying, or else an independently developed understanding of two kinds of time that were relevant to data. Either way, it fell short of full bi-temporality. For while it acknowledged that one kind of time is a period of time, it believed that the other kind of time is a point in time. With only one temporal extent represented, this was at best a quasi-bi-temporal model of data.

This misunderstanding aside, the computer science work on bi-temporality resulted in a proposal for bi-temporal extensions to the SQL language. The extensions were formulated originally as TSQL, later superseded by TSQL2. This proposal was submitted to the SQL Standards Committee in 1994 by Dr. Rick Snodgrass, but to this date has still not been ratified. Nonetheless, there is much that can be done to support bi-temporal functionality using today’s technology, and much to be gained from doing so.

Data Warehouses and Data Marts Come of Age

The second major development in the 90s was that the concept of a data warehouse was proselytized and extended by Bill
Inmon [1996, Inmon]. As a result of this work, the IT industry began to take note of data warehousing. In its purest form, a data warehouse records history as a series of snapshots of the non-transactional tables in legacy system databases. This reflects the fact that, from the point of view of data warehousing, what is important are persistent objects and what they are like at different points in time, i.e. what states they are in as they pass through time. If we are interested in the changes rather than the states themselves, we can reconstruct the history of those changes by extracting the deltas between successive states.

At about the same time, Ralph Kimball [1996, Kimball] took a complementary approach, describing a method of recording history by means of a collection of transactions. With transactions, the focus changes from objects to the relationships among them, for example from a company and its customers to the account balances which track the relationship between that company and each of those customers. Starting with a base state of a relationship, such as account balances on the first of each year, the metrics of those relationships can be recreated, at any subsequent point in time, by applying statistical functions to the base states and the subsequent transactions, e.g. by adding a chronological series of all purchases, payments and returns by a customer to the beginning balance of her account, until any desired point in time is reached.

As the 90s progressed, a religious war developed between those IT professionals who followed Inmon’s data warehouse method of managing history, and those who followed Kimball’s data mart method. These disputes generated more heat than light, and they did so because the complementary nature of the warehouse vs. mart approaches was never clearly recognized.

Because this was the major focus of discussions by IT professionals, during the 90s, about how historical data should be recorded and managed, it is worth trying to describe both what the two camps thought was at stake, as well as what was really at stake. We will describe what was really at stake in the next chapter. Here, we will describe what the two camps thought was at stake.

2The first edition of Inmon’s first book was apparently published in 1991, but we can find no reliable references to it. It seems to us that it was only with the second edition, published in 1996, that the IT community began to take notice of data warehousing.
The Inmon/Kimball Religious Wars

The Kimball Perspective

What the Kimball advocates thought was at stake, in the middle to late 90s, was the difference between a cumbersome and a nimble way of providing access to historical data. They thought the issue was an either/or issue, a choice to be made between data warehouses and data marts, with the correct choice being obvious.

It is true that “nimbleness” was a major concern during those years. Data warehouse projects were nearly always long-term, big-budget projects. Like most such projects, they tended to fail at a high rate. Most failures were probably due to the fact that, in general, big complex projects produce big, complex products, and that with increasing complexity comes increasing odds of mistakes which, over time, often result in failure.

But some failures were also due to senior management losing faith in data warehouse projects. As these large projects fell increasingly behind schedule and rose increasingly over budget—something large projects tend to do—the pressure increased to produce results that had recognizable business value. Patience wore thin, and many data warehouse projects that might have been completed successfully were terminated prematurely.

Against the background of failed data warehouse projects, data mart projects promised results, and promised to deliver them quickly. The typical difference in length of project was about three-to-one: something like two to three years for the typical data warehouse project, but only 8 to 12 months for the typical data mart project.

And in fact, the success rate for data mart projects was significantly higher than the success rate for data warehouse projects. For the most part, this was due to more modest objectives: one-room schoolhouses vs. multi-story skyscrapers. It was part of Kimball’s brilliance to find one-room schoolhouses that were worth building.

The Inmon Perspective

What the Inmon advocates thought was at stake was a “one size fits all” approach vs. an approach that provided different solutions for different requirements. Instead of Kimball’s either/or, they took a both/and stance, and advocated the use of operational data stores (ODSs), historical data warehouses and dimensional data marts, with each one serving different needs.
Indeed, the Inmon approach eventually resulted in an architecture with four components, in which (i) OLTP legacy systems feed an updatable consolidated Operational Data Store (ODS), which in turn (ii) feeds a data warehouse of historical snapshots, which in turn (iii) supplies the dimensions for data marts. At the same time, transactions generated/received by the OLTP systems are consolidated into the fact tables of those data marts. This was an “if you can't beat them, join them” response to the challenge posed by Kimball and his data marts. And it was coupled with a criticism of Kimball. How, Inmon advocates asked, can the data mart approach guarantee consistency across dimensions which are used by multiple fact tables? And if that approach could not guarantee consistency, they pointed out, then the consequences could be as devastating as you care to imagine.

For example, suppose costs summarized in a data mart using a sales organization dimension are compared with revenues summarized in a different data mart using purportedly the same sales organization dimension, but a dimension which was actually created a few days later, and which contained several salesperson assignments not present in the first mart's dimension. In that case, the cost and revenue comparisons for any number of given sales organizations will be comparisons of slightly different organizational units, albeit ones with the same names and unique identifiers.

Once problems like these surface, how can management ever have faith in what either data mart tells them? Doesn't Kimball's data mart approach, Inmon's supporters asked, in effect recreate the data stovepipes of legacy systems, stovepipes which everyone recognized need to be eliminated?

Inmon's supporters had their own answer to this question. They pointed out that if the dimensions for all data marts were extracted from a single source, that source being the enterprise data warehouse, then it would be easy to guarantee that the different dimensions were either identical in content or else conformable in content, i.e. mappable one into the other.

Inmon and Kimball: Going Forward

Acknowledging the force of this criticism, Kimball and his supporters developed an approach called conformed dimensions. They agreed that data, in physically distinct dimensions of data marts, needs to be the same when that data is about the same things. That data, they said, needs to be conformed. It needs to be either instance for instance identical, or derivable one from
the other as, for example, when one set of data is a summarization of the other.

A straightforward way to conform data is to derive it from the same source, at the same time. If transformations are applied to the data as that data is being copied, then semantically like transformations, e.g. summaries or averages, should be based on the same mathematical formulas being applied to the same data. If identical data is subjected to different transformations, but those transformations are mathematically mappable one onto the other, then information derived from those two different sets of derived data will be in agreement.

The 2000s

In the first decade of the new millennium, several major developments took place related to the management of temporal data. They were:

i. On-line analytical processing (OLAP) data cubes;
ii. Slowly changing dimensions (SCDs); and
iii. Real-time data warehousing.

Data cubes are a software structure, manipulated by business intelligence software, that provides rapid and easy access to very large collections of dimensional data, based on often terabyte-sized fact tables. Slowly changing dimensions are a family of uni-temporal structures which provide limited support for historical data in data mart dimension tables. Real-time data warehousing is an evolution from monthly, weekly and then nightly batch updating of warehouses, to real-time transactional updating.

Data Cubes

The first of these three developments is of minimal interest because it is pure technology, involving no advance in the semantics of temporal concepts. We briefly discuss it because it is a convenient way to focus attention on the “cube explosion problem”, the fact that even with terabyte-sized fact tables, a full materialization of sums, counts, and other statistical summarizations of all the instances of all the permutations and combinations of all of the hierarchical levels of all of the dimensions of the typical data mart would dwarf, by many orders of magnitude, the amount of data contained in those terabyte-sized fact tables themselves. Since including uni-temporal or bi-temporal data in the dimensions of a data mart would increase the
number of summarization instances by any number of orders of magnitude, any discussion about introducing temporality into data mart dimensions will have to pay close attention to this potentially overwhelming issue.

**Slowly Changing Dimensions**

The second development, slowly changing dimensions (SCDs), provides a limited solution to part of the problem that bi-temporality addresses, but it is a solution that we believe has confused and continues to confuse the majority of IT practitioners about what bi-temporal data management really is. All too often, when we begin to present bi-temporal concepts, we hear “But Ralph Kimball solved all that with his slowly changing dimensions.”

A new idea is unlikely to attract much attention if its audience believes it addresses a problem that has already been solved. Some clarification is needed, and we hope that, by the end of this book, we will have provided enough clarification to disabuse most IT professionals of that mistaken notion.

But here, we can make two brief points. First of all, SCDs are uni-temporal, not bi-temporal. They do not distinguish changes from corrections.

Moreover, SCDs put much of the semantic burden of accessing temporal data directly on the end user. She is the one who has to know where to look for a specific temporal version of an instance of a dimension. Is it in a different row in the dimension table? If so, which one? If a date column distinguishes them, which date column is it? Or perhaps the temporally distinct instances are to be found in different columns in the same row. If there are several of them, which column is the one she wants?

Of course, there may exist developer-written code which translates the user’s request into physical access to physical rows and columns within the dimension. Perhaps, as should be the case, all the user has to specify, for example, is that she wants, in the product dimension, a specific sku and, for that sku, the description in effect on a specific date.

This is an improvement. But all it really does is move the semantic burden from the end user to the developer and her hand-written code. It is like the days long ago when assembler programmers, about to update a set of records in memory, had to load a starting address in one register, the row length of the records in another register, the number of records in the set in a third register, and then code a loop through the set of records by updating each record, adding the record length to the current memory location and thus moving on to the next record.
Whenever we can specify the semantics of what we need, without having to specify the steps required to fulfill our requests, those requests are satisfied at lower cost, in less time, and more reliably. SCDs stand on the wrong side of that what vs. how divide.

Some IT professionals refer to a type 1.5 SCD. Others describe types 0, 4, 5 and 6. Suffice it to say that none of these variations overcome these two fundamental limitations of SCDs. SCDs do have their place, of course. They are one tool in the data manager's toolkit. Our point here is, first of all, that they are not bi-temporal. In addition, even for accessing uni-temporal data, SCDs are cumbersome and costly. They can, and should, be replaced by a declarative way of requesting what data is needed without having to provide explicit directions to that data.

Real-Time Data Warehouses

As for the third of these developments, it muddles the data warehousing paradigm by blurring the line between regular, periodic snapshots of tables or entire databases, and irregular as-needed before-images of rows about to be changed. There is value in the regularity of periodic snapshots, just as there is value in the regular mileposts along interstate highways. Before-images of individual rows, taken just before they are updated, violate this regular snapshot paradigm, and while not destroying, certainly erode the milepost value of regular snapshots.

On the other hand, periodic snapshots fail to capture changes that are overwritten by later changes, and also fail to capture inserts that are cancelled by deletes, and vice versa, when these actions all take place between one snapshot and the next. As-needed row-level warehousing (real-time warehousing) will capture all of these database modifications.

Both kinds of historical data have value when collected and managed properly. But what we actually have, in all too many historical data warehouses today, is an ill-understood and thus poorly managed mish-mash of the two kinds of historical data. As result, these warehouses provide the best of neither world.

The Future of Databases: Seamless Access to Temporal Data

Let's say that this brief history has shown a progression in making temporal data “readily available”. But what does “readily available” really mean, with respect to temporal data?
One thing it might mean is “more available than by using backups and logfiles”. And the most salient feature of the advance from backups and logfiles to these other methods of managing historical data is that backups and logfiles require the intervention of IT Operations to restore desired data from off-line media, while history tables, warehouses and data marts do not. When IT Operations has to get involved, emails and phone calls fly back and forth. The Operations manager complains that his personnel are already overloaded with the work of keeping production systems running, and don’t have time for these one-off requests, especially as those requests are being made more and more frequently.

What is going on is that the job of Operations, as its management sees it, is to run the IT production schedule and to complete that scheduled work on time. Anything else is extra. Anything else is outside what their annual reviews, salary increases and bonuses are based on.

And so it is frequently necessary to bump the issue up a level, and for Directors or even VPs within IT to talk to one another. Finally, when Operations at last agrees to restore a backup and apply a logfile (and do the clean-up work afterwards, the manager is sure to point out), it is often a few days or a few weeks after the business use for that data led to the request being made in the first place. Soon enough, data consumers learn what a headache it is to get access to backed-up historical data. They learn how long it takes to get the data, and so learn to do a quick mental calculation to figure out whether or not the answer they need is likely to be available quickly enough to check out a hunch about next year’s optimum product mix before production schedules are finalized, or support a position they took in a meeting which someone else has challenged. They learn, in short, to do without a lot of the data they need, to not even bother asking for it.

But instead of the comparative objective of making temporal data “more available” than it is, given some other way of managing it, let’s formulate the absolute objective for availability of temporal data. It is, simply, for temporal data to be as quickly and easily accessible as it needs to be. We will call this the requirement to have seamless, real-time access to what we once believed, currently believe, or may come to believe is true about what things of interest to us were like, are like, or may come to be like in the future.

This requirement has two parts. First, it means access to non-current states of persistent objects which is just as available to the data consumer as is access to current states. The temporal
data must be available on-line, just as current data is. Transactions to maintain temporal data must be as easy to write as are transactions to maintain current data. Queries to retrieve temporal data, or a combination of temporal and current data, must be as easy to write as are queries to retrieve current data only. This is the \textit{usability aspect} of seamless access.

Second, it means that queries which return temporal data, or a mix of temporal and current data, must return equivalent-sized results in an equivalent amount of elapsed time. This is the \textit{performance aspect} of seamless access.

\section*{Closing In on Seamless Access}

Throughout the history of computerized data management, file access methods (e.g. VSAM) and database management systems (DBMSs) have been designed and deployed to manage current data. All of them have a structure for representing types of objects, a structure for representing instances of those types, and a structure for representing properties and relationships of those instances. But none of them have structures for representing objects as they exist within periods of time, let alone structures for representing objects as they exist within two periods of time.

The earliest DBMSs supported sequential (one-to-one) and hierarchical (one-to-many) relationships among types and instances, and the main example was IBM's IMS. Later systems more directly supported network (many-to-many) relationships than did IMS. Important examples were Cincom's TOTAL, ADR's DataCom, and Cullinet's IDMS (the latter two now Computer Associates' products).

Later, beginning with IBM's System R, and Dr. Michael Stonebreaker's Ingres, Dr. Ted Codd's relational paradigm for data management began to be deployed. Relational DBMSs could do everything that network DBMSs could do, but less well understood is the fact that they could also do nothing more than network DBMSs could do. Relational DBMSs prevailed over CODASYL network DBMSs because they simplified the work required to maintain and access data by supporting declaratively specified set-at-a-time operations rather than procedurally specified record-at-a-time operations.

Those record-at-a-time operations work like this. Network DBMSs require us to retrieve or update multiple rows in tables by coding a \textit{loop}. In doing so, we are writing a procedure; we are telling the computer \textit{how} to retrieve the rows we are interested in. So we wrote these loops, and retrieved (or updated) one row at a time. Sometimes we wrote code that produced
infinite loops when confronted with unanticipated combinations of data. Sometimes we wrote code that contained “off by one” errors. But SQL, issued against relational databases, allows us to simply specify what results we want, e.g. to say that we want all rows where the customer status is XYZ. Using SQL, there are no infinite loops, and there are no off-by-one errors.

For the most part, today’s databases are still specialized for managing current data, data that tells us what we currently believe things are currently like. Everything else is an exception. Nonetheless, we can make historical data accessible to queries by organizing it into specialized databases, or into specialized tables within databases, or even into specialized rows within tables that also contain current data.

But each of these ways of accommodating historical data requires extra work on the part of IT personnel. Each of these ways of accommodating historical data goes beyond the basic paradigm of one table for every type of object, and one row for every instance of a type. And so DBMSs don’t come with built-in support for these structures that contain historical data. We developers have to design, deploy and manage these structures ourselves. In addition, we must design, deploy and manage the code that maintains historical data, because this code goes beyond the basic paradigm of inserting a row for a new object, retrieving, updating and rewriting a row for an object that has changed, and deleting a row for an object no longer of interest to us.

We developers must also design, deploy and maintain code to simplify the retrieval of instances of historical data. SQL, and the various reporting and querying tools that generate it, supports the basic paradigm used to access current data. This is the paradigm of choosing one or more rows from a target table by specifying selection criteria, projecting one or more columns by listing the columns to be included in the query’s result set, and joining from one table to another by specifying match or other qualifying criteria from selected rows to other rows.

When different rows represent objects at different periods of time, transactions to insert, update and delete data must specify not just the object, but also the period of time of interest. When different rows represent different statements about what was true about the same object at a specified period of time, those transactions must specify two periods of time in addition to the object.

Queries also become more complex. When different rows represent objects at different points in time, queries must specify not just the object, but also the point in time of interest. When different rows represent different statements about what was
true about the same object at the same point in time, queries must specify two points in time in addition to the criteria which designate the object or objects of interest.

We believe that the relational model, with its supporting theory and technology, is now in much the same position that the CODASYL network model, with its supporting theory and technology, was three decades ago. It is in the same position, in the following sense.

Relational DBMSs were never able to do anything with data that network DBMSs could not do. Both supported sequential, hierarchical and network relationships among instances of types of data. The difference was in how much work was required on the part of IT personnel and end users to maintain and access the managed data.

And now we have the relational model, a model invented by Dr. E. F. Codd. An underlying assumption of the relational model is that it deals with current data only. But temporal data can be managed with relational technology. Dr. Snodgrass's book describes how current relational technology can be adapted to handle temporal data, and indeed to handle data along two orthogonal temporal dimensions. But in the process of doing so, it also shows how difficult it is to do.

In today's world, the assumption is that DBMSs manage current data. But we are moving into a world in which DBMSs will be called on to manage data which describes the past, present or future states of objects, and the past, present or future assertions made about those states. Of this two-dimensional temporalization of data describing what we believe about how things are in the world, currently true and currently asserted data will always be the default state of data managed in a database and retrieved from it. But overrides to those defaults should be specifiable declaratively, simply by specifying points in time other than right now for versions of objects and also for assertions about those versions.

Asserted Versioning provides seamless, real-time access to bi-temporal data, and provides mechanisms which support the declarative specification of bi-temporal parameters on both maintenance transactions and on queries against bi-temporal data.

**Glossary References**

Glossary entries whose definitions form strong inter-dependencies are grouped together in the following list. The same glossary entries may be grouped together in different ways.
at the end of different chapters, each grouping reflecting the semantic perspective of each chapter. There will usually be several other, and often many other, glossary entries that are not included in the list, and we recommend that the Glossary be consulted whenever an unfamiliar term is encountered.

- effective time
- valid time
- event
- state
- external pipeline dataset, history table
- transaction table
- version table
- instance
- type
- object
- persistent object
- thing
- seamless access
- seamless access, performance aspect
- seamless access, usability aspect