Introduction to ANSI Temporal Tables

Introduction

Introduction to ANSI Temporal Tables is a Web-based training designed for those who need a basic understanding of temporal concepts and Teradata's implementation of ANSI temporal syntax.

This training is designed to be easy to understand. Once you have familiarized yourself with the course content, you may come back to it easily and review the information you need to support your temporal related efforts.

This course is updated for Teradata Database 15.10.

As a result of this training, you should be able to:

- Recognize the value and purpose of temporal tables to a business.
- List and differentiate the different types of temporal tables.
- List the available temporal qualifiers and the types of information they will produce.
- Recognize the benefits and limitations of adding temporal data as an enhancement to traditional data tables.

This course consists of self-study activities and a post-test. You may proceed through the activities at your own pace. The time to complete the course is estimated at 4 hours.

See the help page for additional resources and links.

You must complete the post-test to receive credit for the course.

If you wish, you may download a hard copy for reference.

Temporal Databases

What is a temporal database?

A **temporal database** is a database that includes temporal tables.

Okay, so what are temporal tables?

Temporal tables are tables that provide built-in support for the time dimension of data. They provide special qualifiers for storing, querying, and updating historical, current and future data. For example, they can store the history of an item as it moves through a supply chain or the history of an employee's movement within an organization.

What is the difference between a **temporal database** and a **conventional database**?

A temporal database maintains data with respect to time. It allows time-based reasoning and querying. On the other hand, a conventional database captures only a current snapshot of reality.

Of course, a conventional database may also keep historical data, however the maintenance and querying of this data must be controlled at the application program level.

Teradata's temporal Data Base Management System (DBMS) provides a temporal version of SQL, including enhancements to the data definition language (DDL), constraint specifications, data types, and data manipulation language (DML).

With the release of Teradata Database 15.00, Teradata introduced ANSI temporal syntax.

Teradata had originally introduced its temporal syntax prior to the existence of ANSI standards. While Teradata will continue to support its original temporal syntax, using ANSI-compliant syntax is recommended for new development because third party tools for Business Intelligence and ETL will most likely be aligned with ANSI standards.

There are two major differences between the syntax choices:

1. Whereas Teradata syntax uses the PERIOD data type, ANSI syntax uses derived periods.
2. Whereas Teradata syntax uses TRANSACTIONTIME, ANSI uses SYSTEM_TIME to show when a fact was known to the database.

In this course, we will use ANSI syntax.

Temporal tables are generally used to represent **states** of an object (as distinct from **events**).

State tables define the state of an object.

Example: When an employee changes department, add a new row and update valid and transaction times for the employee to reflect the change.

Teradata temporal tables are often used to implement "state of" processing:

- Identified by a time period
- Identified by slowly changing states

Event tables define the occurrence of an event.

Example: When a stock falls below a given price, log it and trigger a sale of some shares.

Teradata queue tables are used to implement event processing:

- Identified by an instance, such as timestamp.
- Characterized by quickly changing events that may require some action.

Two **events** describe the begin and end of an object **state.** Another way to think about it is that the **state** of an object is delimited by beginning and ending **events**.

Most applications access databases where information changes over time. It may also be a requirement to capture the state of data at specific points in time - past, present and future. For example, we might wish to see what a specific table looked like two years ago.

Doing this without temporal table support might be possible, although it is complex and it presumes that the data from two years ago is still available.

On the next page, we will look at an example of where a temporal approach would be helpful.

Consider an application for an insurance company that uses a ***Policy*** table where the definition looks like this:

```
CREATE TABLE Policy (
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2),
 Policy_Details CHAR(40))
UNIQUE PRIMARY INDEX(Policy_ID);
```

Note that this table has no time-based columns.

Now suppose the application needs to record when rows in the ***Policy*** table are *valid* (i.e., when a policy is in force).

One approach is to add a DATE column to the ***Policy*** table called ***Start_Date*** and another DATE column called ***End_Date***. The new definition of the table looks like this:

```
CREATE TABLE Policy (
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2),
 Policy_Details CHAR(40),
 Start_Date DATE,
 End_Date DATE)
UNIQUE PRIMARY INDEX(Policy_ID);
```

Some complications are now evident.

What if, for example, a customer makes a change to his/her policy during the life of the policy?

If there is a change during the life of the policy, a new row will need to be created to store the new policy conditions that are in effect from that time until the end of the policy.

But the policy conditions prior to the change are also likely to be important to retain for historical reasons.

The original row represents the conditions that were in effect for the beginning portion of the policy, but the ***End_Date*** will need to be updated to reflect when the policy conditions were changed. The new row will reflect the 'new' version of the policy.

We can see that because of these changes, it is necessary that more than one row will have the same value for ***Policy_ID***. So the primary index for the table might need to change to be non-unique.

All modifications to the table must now consider changing the ***Start_Date*** and ***End_Date*** columns.

Queries will also become more complicated, because they need to filter based on the ***Start_Date*** and ***End_Date*** columns.

The presence of a DATE column in a table does not make the table a temporal table, nor does it make the database a temporal database. A temporal database must **automatically record and manage the time-varying nature of the information** as used by the enterprise.

Rather than using approaches such as adding DATE columns to traditional tables, the Teradata Database provides support to effectively create, query, and modify time-varying tables in a completely different manner using a combination of internal temporal logic and temporal extensions to SQL.

Temporal databases increase your depth of business intelligence activities. With temporal tables, you can:

- Conduct 'chain of events' analytics
- Conduct 'point in time' analytics
- Reconstruct historical transaction details, including a complete replication of a table's state at a past point in time

Temporal databases provide technical benefits by simplifying query coding. This can:

- Reduce the cost of query development
- Reduce the cost of data maintenance

Teradata provides temporal table support at the column level using **derived period** columns.

A **period** is an anchored duration that represents a set of contiguous time granules within the duration. It has a beginning and ending bound, defined by two DateTime type columns in the table.

Here is an example from a valid-time table. The derived period column is named ***Duration***:

```
...
Period_Begin TIMESTAMP(6) WITH ZONE NOT NULL,
Period_End TIMESTAMP(6) WITH ZONE NOT NULL,
PERIOD FOR Duration(Period_Begin,Period_End) AS VALIDTIME
...
```

In a temporal table, the values from the beginning and ending bound columns are combined in a third, derived column that represents the period of time, or duration they delimit.

The derived period column is maintained dynamically by the database.

The elements of a derived period column must have matching data types. In other words, you can use two DATE columns, or two TIME columns, or two TIMESTAMP columns. You may not use a DATE column for the beginning bound and a TIME column for the ending bound.

The period that a derived period column represents **starts at the beginning bound** and **extends up to, but does not include, the ending bound** value.

ANSI temporal tables can have one or two derived period columns to represent different kinds of time. These derived period columns combine the values from two regular DateTime type columns, and represent the period or duration bounded by the values in the two component columns.

Derived period columns function much like native Teradata Database Period data type columns. Period data types are not ANSI compatible, they are Teradata extensions to the ANSI SQL:2011 standard.

However, many of Teradata Database's functions, operators, and predicates for Period data types can be used on derived temporal columns.

Now the application for the insurance company can create the ***Policy*** table with a **derived period** column to record the period during which each policy (row) is valid.

```
CREATE TABLE Policy (
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2),
 Policy_Details CHAR(40),
 Policy_Begin DATE NOT NULL,
 Policy_End DATE NOT NULL,
 PERIOD FOR Policy_Duration
  (Policy_Begin,Policy_End) AS VALIDTIME)
PRIMARY INDEX(Policy_ID);
```

The derived period column ***Policy_Duration*** is used internally. The value for each row is created and maintained automatically by Teradata Database based on the table definition. Data is never inserted explicitly for the derived period column, and that column is never itself returned or queried.

Temporal Time

Temporal tables include one or two derived period columns, which represent time information:

A **system-time column** represents the time period for which the Teradata Database was aware of the information in the row. It reflects the database reality. For system-time tables, the database automatically records when rows have been added, modified, and deleted. The user does not need to do anything to maintain this.

A **valid-time column** represents the time period when a 'fact' was valid. It reflects the real-world reality. ANSI calls this 'application time.' Examples might be the time an insurance policy or product warranty is valid or the length of employment of an employee. Often this data comes from a source system, and is not maintained by the Teradata Database itself (the exception to this is when you make a time-bounded change to a row in a valid-time table).

The system-time and valid-time period values can both exist as separate columns in the same row and are often different.

A fact recorded in the database may take place before or after the real world change occurs.

A salary increase which is scheduled to start next month may be input, and therefore known, to the database today.

A temporal table has either a Valid-Time dimension, a System-Time dimension, or both.

Temporal tables can be classified into 3 groups:

- **Valid-Time table** - A table that has a derived period column with a VALIDTIME attribute
- **System-Time table** - A table that has a derived period column with a SYSTEM_TIME attribute, with system versioning enabled
- **Bitemporal table** - A table that is both valid-time and system-time

Temporal databases may have both temporal and nontemporal tables.

- **Temporal table** - A table that is either valid-time, system-time, or bitemporal
- **Nontemporal table** - A table that has neither valid-time nor system-time

When working with temporal tables, it is important to recall that **periods** (including derived periods) are **inclusive-exclusive**. This means that the **start time is included in the period**, but the **stop time is not**.

Consider the following two examples:

beginning bound: **`DATE '2017-02-01'`**
ending bound: **`DATE '2017-02-10'`**

This period includes the entire day of February 1, but does not include the day February 10. The period begins with the midnight starting point of February 1 and ends with the midnight ending point of February 9.

beginning bound: **`TIME '08:00:00'`**
ending bound: **`TIME '15:40:00'`**

This period begins with the starting point of 08:00:00 and ends with the last second preceding 15:40:00 (i.e. 15:39:59).

Notice that **the level of granularity dictates exactly when the period ends**. In the first case, the granularity is days, so the entire day of February 10 is excluded. In the second case, the granularity is seconds, so the exact second represented by 15:40:00 is excluded.

Try It!

empnum	deptnum	begin_dttm	end_dttm
1001	100	2017-05-01 09:47:22.260000+00:00	2017-11-23 09:49:44.520000+00:00

Given the row above, which of these choices are included in the valid time?

2017-05-01 00:00:00.000000+00:00

2017-06-01 00:00:00.000000+00:00

2017-10-01

2017-11-23 09:49:44.520000+00:00

2017-11-23 00:00:00.000000+00:00

To work with ANSI temporal tables, your session temporal qualifier must be set to ANSIQUALIFIER. This is the default.

You can verify this with a **HELP SESSION** command.

Example

```
HELP SESSION;
```

...	Temporal Qualifier	Calendar	...
...	**ANSIQUALIFIER**	TERADATA	...

System-Time Tables

System-Time denotes the time period when a fact (row) is **known (or recorded in) the database.** The beginning bound represents when a fact is entered in the database. The ending bound represents when a new fact supersedes the existing fact or when a fact is deleted.

For a table to be system-time, it needs to have:

- The beginning and ending bound columns for the system-time period, each with a time of TIMESTAMP(6) WITH TIME ZONE
- The SYSTEM_TIME derived period column
- System versioning enabled on the table

A table that has system-time (but not a valid-time) is called a **system-time table**.

The system-time values are maintained entirely by the system, not by the user.

```
CREATE MULTISET TABLE employee
  (ename VARCHAR(12),
   eno INTEGER,
   deptno INTEGER,
   sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS
ROW START,
   sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS ROW
END,
   PERIOD FOR SYSTEM_TIME  (sys_start, sys_end))
PRIMARY INDEX ( eno ) WITH SYSTEM VERSIONING
;
```

In a system-time table, rows are either **open** or **closed**.

A row is considered **open** if the ending bound is open-ended (represented by TIMESTAMP '9999-12-31 23:59:59.999999+00:00' or UNTIL_CLOSED). In the example below, the row for ***empno*** 1001 in ***dept*** 300 is **open**.

A row is considered **closed** if the period has a 'hard' end date that is not UNTIL_CLOSED or its literal equivalent. In the example below, the row for ***empno*** 1001 in ***dept*** 100 is **closed**.

empname	empno	dept	sys_start	sys_end
Abigail	**1001**	**100**	**2015-06-01 09:47:22.260000+00:00**	**2015-11-26 09:49:44.520000+00:00**
Abigail	**1001**	**300**	**2015-11-26 09:49:44.520000+00:00**	**9999-12-31 23:59:59.999999+00:00**
Bob	1002	200	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
Chris	1003	100	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00
Chris	1003	300	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00

Try It!

Assuming the temporal_date is June 30, 2016, determine if the below rows are **open** or **closed**.

...	sys_start	sys_end
...	**2015-05-01 09:47:22.260000+00:00**	**2016-06-01 09:49:44.520000+00:00**

Is this row open or closed?

...	sys_start	sys_end
...	**2015-05-01 09:47:22.260000+00:00**	**9999-12-31 23:59:59.999999+00:00**

Is this row open or closed?

If no temporal qualifiers are used in the FROM clause of a SELECT query, the default for system-time tables is to qualify all rows that are currently (at the time of the query) **open**.

If, however, you wish to filter based on the system-time, Teradata Database provides special syntax to be used in the FROM clause. These temporal qualifiers take precedence over other criteria that may be specified in a WHERE clause, which can be used to further restrict the rows returned.

You have four options: **AS OF** a point in time, **BETWEEN ... AND**, **FROM ... TO**, and **CONTAINED IN**.

We will look at examples of these on the following pages.

For the examples on the following pages, consider the below system-time table ***salary***.

```
CREATE MULTISET TABLE salary
   (empno INTEGER,
    salary INTEGER,
    sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS
ROW START,
    sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS
ROW END,
    PERIOD FOR SYSTEM_TIME  (sys_start, sys_end))
PRIMARY INDEX ( empno ) WITH SYSTEM VERSIONING;
```

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

A simple SELECT, with no other temporal qualifier, will only retrieve **open** rows.

```
SELECT * FROM salary;
```

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

Using **AS OF** returns rows that were (or will be) open at a specific point in time.

```
SELECT * FROM salary
FOR system_time AS OF TIMESTAMP'2015-11-26 09:48:00.000000+00:00';
```

empno	salary	sys_start	sys_end
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

Using **BETWEEN .. AND** returns rows that were (or will be) open during a specified period of time, or *immediately after*. Be careful! It is not a typical usage of the word "between" to also include immediately after. You may instead want to use **FROM ... TO** or **CONTAINED IN**.

```
SELECT * FROM salary
FOR system_time BETWEEN TIMESTAMP'2015-11-26 09:00:00.000000+00:00'
                AND     TIMESTAMP'2015-11-26 09:49:28.720000+00:00';
```

empno	salary	sys_start	sys_end
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

Both rows for employee 1003 are returned, even though the row for the higher salary doesn't begin until the ending timestamp of the period defined in the query.

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

Using **FROM .. TO** returns rows that overlap a specified period of time.

```
SELECT * FROM salary
FOR system_time FROM    TIMESTAMP'2015-11-26 09:00:00.000000+00:00'
                TO      TIMESTAMP'2015-11-26 09:49:28.720000+00:00';
```

empno	salary	sys_start	sys_end
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	**40000**	**2015-06-01 09:47:22.550000+00:00**	**2015-11-26 09:49:28.720000+00:00**

empno	salary	sys_start	sys_end
1001	50000	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1002	35000	2015-06-01 09:47:22.520000+00:00	9999-12-31 23:59:59.999999+00:00
1003	45000	2015-11-26 09:49:28.720000+00:00	9999-12-31 23:59:59.999999+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

Using **CONTAINED IN** qualifies rows with system-time periods that are entirely contained between two points. The system-time period must start at or after the first point, and must end before or at the second point.

Note: CONTAINED IN is a Teradata Database extension to ANSI.

```
SELECT * FROM salary
FOR system_time CONTAINED IN (DATE'2015-06-01', DATE'2015-12-01');
```

empno	salary	sys_start	sys_end
1001	40000	2015-06-01 09:47:22.260000+00:00	2015-11-26 09:49:44.520000+00:00
1003	40000	2015-06-01 09:47:22.550000+00:00	2015-11-26 09:49:28.720000+00:00

Only the rows **opened *and* closed** between June 1, 2015 and December 1, 2015 are returned.

When you modify rows in system-versioned system-time tables, **Teradata Database automatically handles the system-time period**.

DELETE: Deleted rows are "closed" in system time by having the end of their system-time period set to the time of the deletion. The row is not physically deleted.

UPDATE: When you modify a row, the old row is "closed" in system time, and a new row is automatically created to reflect the new information. The new row will have a system-time beginning at the time of the modification. The old row is not physically deleted.

The only way to physically delete rows from a system-time table is to explicitly drop system versioning from the table. When you drop system versioning, Teradata Database automatically deletes all closed rows.

Since modification cannot overwrite past records, the results of past queries can be easily reconstructed. It also means that errors, both intentional (i.e. fraud) and non-intentional, can be easily detected and recovered. The impact of those errors can be evaluated.

For the examples on the following pages, consider the below system-time table ***salary***.

```
CREATE MULTISET TABLE salary
   (empno INTEGER,
    salary INTEGER,
    sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS
ROW START,
    sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS
ROW END,
    PERIOD FOR SYSTEM_TIME  (sys_start, sys_end))
PRIMARY INDEX ( empno ) WITH SYSTEM VERSIONING;
```

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	9999-12-31 23:59:59.999999+00:00
1005	40000	2015-12-01 09:31:00.000000+00:00	9999-12-31 23:59:59.999999+00:00

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	9999-12-31 23:59:59.999999+00:00
1005	40000	2015-12-01 09:31:00.000000+00:00	9999-12-31 23:59:59.999999+00:00

A DELETE will close the row, but not physically delete it.

```
DELETE FROM salary
WHERE empno = 1004;
```

If today's date is December 15, 2015, the resulting table will look like this:

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	**2015-12-15 11:15:34.360000+00:00**
1005	40000	2015-12-01 09:31:00.000000+00:00	9999-12-31 23:59:59.999999+00:00

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	2015-12-15 11:15:34.360000+00:00
1005	40000	2015-12-01 09:31:00.000000+00:00	9999-12-31 23:59:59.999999+00:00

When you UPDATE, the old row is closed and a new (open) row is inserted.

```
UPDATE salary
SET salary = 50000
WHERE empno = 1005;
```

If today's date is December 15, 2015, the resulting table will look like this:

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	2015-12-15 11:15:34.360000+00:00
1005	40000	2015-12-01 09:31:00.000000+00:00	**2015-12-15 11:20:17.420000+00:00**
1005	50000	2015-12-15 11:20:17.420000+00:00	9999-12-31 23:59:59.999999+00:00

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	2015-12-15 11:15:34.360000+00:00
1005	40000	2015-12-01 09:31:00.000000+00:00	2015-12-15 11:20:17.420000+00:00
1005	50000	2015-12-15 11:20:17.420000+00:00	9999-12-31 23:59:59.999999+00:00

When you INSERT a row, you need to provide the values for the beginning and ending bounds of the system time (since they are NOT NULL). However, the Teradata Database will automatically replace those values with the appropriate system timestamp.

```
INSERT INTO salary VALUES (1006, 450000, TIMESTAMP'2000-01-01
00:00:00.000000+00:00', TIMESTAMP'2000-01-01 00:00:00.000000+00:00');
```

If today's date is December 15, 2015, the resulting table will look like this:

empno	salary	sys_start	sys_end
1004	35000	2015-12-01 09:30:00.000000+00:00	2015-12-15 11:15:34.360000+00:00
1005	40000	2015-12-01 09:31:00.000000+00:00	2015-12-15 11:20:17.420000+00:00
1005	50000	2015-12-15 11:20:17.420000+00:00	9999-12-31 23:59:59.999999+00:00
1006	45000	**2015-12-15 13:48:12.110000+00:00**	**9999-12-31 23:59:59.999999+00:00**

System Time Use Case

Case Study - A regulation to protect tax payers' privacy

A tax agency had a requirement to be able to replicate any query run against the database on a past date with the same results as seen on that date.

One approach considered was to store each issued query along with the resulting answer set. This solution was very time, storage, and labor intensive.

Another approach considered was to temporalize the appropriate tables by adding system versioning. With this approach, when query results needed to be recreated, a simple AS OF with the appropriate timestamp could be added to the query to replicate past results.

This second approach was much easier to implement and less resource intensive. Extra space was still needed to carry the additional closed rows, however this was far less than storing each query and its result set. The ease of replication of any query made the tradeoff acceptable.

The following are facts and rules related to system-time tables:

- The system time specifies when the row was known (inserted) to the database.
- The system time indicates whether the row is current (i.e., open) or history (closed).
- The user has no control over system time – it is managed completely by the system.
- The temporal database only appends rows to a system-time table; it never deletes them.
- Old rows are assigned a system end time, which effectively closes the row but does not delete it.
- New rows are assigned a system begin time and an UNTIL_CLOSED end time.
- A row with UNTIL_CLOSED as its end system time is considered **open**.
- A row is **closed** when it is superseded by a new open row and then end time is modified by the system to be the current timestamp.

This complete, maintained history of all changes to the table guarantees the integrity of past reconstructions and audits.

Valid-Time Tables

Valid-Time is used to model the real world. It denotes the time period during which **a fact (i.e. row) is true (i.e. valid)**.

For a table to be valid-time, it needs to have:

- The beginning and ending bound columns for the valid-time period, each with a DATE or TIMESTAMP data type, defined as NOT NULL
- The VALIDTIME derived period column

A table that has valid-time (but not a system-time) is called a **valid-time table**.

The valid-time values are often supplied by source systems.

```
CREATE MULTISET TABLE employee
  (ename VARCHAR(12),
   eno INTEGER,
   deptno INTEGER,
   start_date DATE NOT NULL,
   end_date DATE NOT NULL,
   PERIOD FOR emp_duration (start_date, end_date) AS VALIDTIME)
PRIMARY INDEX ( eno )
;
```

A common problem in data warehousing is supporting slowly changing dimensions without built-in temporal support.

What is a **slowly changing dimension** (**SCD**)? Examples are such things as customer change of address, change of last or first name, or change of marital status. These are changes that occur infrequently over lengthy periods of time in some cases. There are two ways generally used to handle this type of processing without temporal support:

- Type 1: New record replaces the old. In this case, the old information is not kept. It can be difficult or impossible to recreate historical situations.
- Type 2: New record is added to the table for a change of address. Two (or more) rows now exist for the same customer – one for each address -with a date/time showing when the new row was added. You can recreate history, but maintenance can be difficult.

Using a Valid-Time table is a good way to manage a slowly changing dimension. With temporal aggregation and join support, analysis of dimension table information is easy.

A row is considered **valid** if the TEMPORAL_DATE or TEMPORAL_TIMESTAMP (which typically are the same as Current_Date and Current_Timestamp) falls within the period of its valid time.

In a valid-time table, rows are described as **current**, **history**, or **future**, in relation to the Temporal_Date or Temporal_Timestamp value.

Current - The temporal date falls **within** the valid time period.
History - The temporal date falls **after** the valid time period.
Future - The temporal date falls **before** the valid time period.

If today is January 30, 2016, then in the example below:

The row for ***empno*** 1001 in ***dept*** 100 is **history**.
The row for ***empno*** 1001 in ***dept*** 300 is **current**.
The row for ***empno*** 1003 in ***dept*** 300 is **future**.

ename	eno	deptno	start_date	end_date
Darcy	**1001**	**100**	**2015-01-01**	**2016-01-01**
Darcy	**1001**	**300**	**2016-01-01**	**9999-12-31**
Edgar	1002	200	2015-06-01	9999-12-31
Frances	1003	100	2015-06-01	2016-05-01
Frances	**1003**	**300**	**2016-05-01**	**9999-12-31**

The valid time period of a row is called its Period of Validity (PV).

In the example below, the Period of Validity of the first row is twelve months long. It begins on January 1, 2015. It ends at January 1, 2016.

Recall that period ending bounds are exclusive. In other words, December 31, 2015 is included in the PV, but January 1, 2016 is not included in the PV.

ename	eno	deptno	start_date	end_date
Darcy	1001	100	**2015-01-01**	**2016-01-01**
Darcy	1001	300	2016-01-01	9999-12-31
Edgar	1002	200	2015-06-01	9999-12-31
Frances	1003	100	2015-06-01	2016-05-01
Frances	1003	300	2016-05-01	9999-12-31

Try It!

Assuming the temporal_date is June 30, 2016, determine if the below rows are **history**, **current**, or **future**.

ename	eno	deptno	start_date	end_date
Gerald	1050	100	**2000-04-21**	**9999-12-31**

Is this row history, current, or future?

ename	eno	deptno	start_date	end_date
Hanna	1051	200	**2010-09-30**	**2012-06-15**

Is this row history, current, or future?

ename	eno	deptno	start_date	end_date
Ilsa	1053	300	**2017-01-01**	**9999-12-31**

Is this row history, current, or future?

If no temporal qualifiers are used in the FROM clause of a SELECT query, **the default for valid-time tables is to return all rows.**

If, however, you wish to filter based on the valid-time, Teradata Database provides special syntax to be used in the FROM clause. These are the same as those used for system-time. These temporal qualifiers take precedence over other criteria that may be specified in a WHERE clause, which can be used to further restrict the rows returned.

You have four options: **AS OF** a point in time, **BETWEEN ... AND**, **FROM ... TO**, and **CONTAINED IN**.

We will look at examples of these on the following pages.

For the examples on the following pages, consider the below valid-time table ***emp_territory***.

```
CREATE MULTISET TABLE emp_territory
   (empID INTEGER,
    territoryID INTEGER,
    valid_start DATE NOT NULL,
    valid_end DATE NOT NULL,
    PERIOD FOR duration (valid_start, valid_end) AS VALIDTIME)
PRIMARY INDEX ( empID );
```

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

A simple SELECT, with no other temporal qualifier, will retrieve **all** rows.

```
SELECT * FROM emp_territory;
```

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

Using **AS OF** returns rows that were/are/will be valid at a specific point in time.

```
SELECT * FROM emp_territory
FOR validtime AS OF TIMESTAMP'2015-11-15 09:48:00.000000+00:00';
```

empID	territoryID	valid_start	valid_end
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01

Notice that we can filter by a TIMESTAMP WITH ZONE, even though the data type of the valid time bounding columns is DATE. This query would produce the same result:

```
SELECT * FROM emp_territory
FOR validtime AS OF DATE'2015-11-15';
```

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

Using **BETWEEN .. AND** returns rows that were (or will be) valid during a specified period of time, or *immediately after*. Be careful! It is not a typical usage of the word "between" to also include immediately after. You may instead want to use **FROM ... TO** or **CONTAINED IN**.

```
SELECT * FROM emp_territory
FOR validtime BETWEEN DATE'2015-12-01' AND DATE'2016-01-01';
```

empID	territoryID	valid_start	valid_end
1001	**21**	**2016-01-01**	**9999-12-31**
1001	22	2015-09-01	9999-12-31
1002	**21**	**2015-06-01**	**2016-01-01**
1002	31	2016-01-01	9999-12-31

Both rows for ***territoryID*** 21 are returned, even though the row for ***employeeID*** 1001 is not valid until the ending date of the period defined in the query.

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

Using **FROM .. TO** returns rows that overlap a specified period of time.

```
SELECT * FROM emp_territory
FOR validtime FROM DATE'2015-12-01' TO DATE'2016-01-01';
```

empID	territoryID	valid_start	valid_end
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

Using **CONTAINED IN** qualifies rows with valid-time periods that are *entirely contained* between two points. The valid-time period must start at or after the first point, and must end before or at the second point.

Note: CONTAINED IN is a Teradata Database extension to ANSI.

```
SELECT * FROM emp_territory
FOR validtime CONTAINED IN
   (DATE'2015-06-01', DATE'2016-01-01');
```

empID	territoryID	valid_start	valid_end
1002	21	2015-06-01	2016-01-01

Only the row that **started *and* ended** between June 1, 2015 and January 1, 2016 is returned. No open-ended valid times will ever be returned with CONTAINED IN.

Try it!

empID	territoryID	valid_start	valid_end
1001	21	2016-01-01	9999-12-31
1001	22	2015-09-01	9999-12-31
1002	21	2015-06-01	2016-01-01
1002	31	2016-01-01	9999-12-31
1003	35	2016-07-01	9999-12-31

Given the above table, what the results of this query be?

```
SELECT territoryID
FROM emp_territory FOR validtime AS OF DATE'2016-03-01'
WHERE empID = 1002;
```

21

31

Modifications to valid-time tables are more flexible than those to system-time tables. You can modify a row directly as you would do on a nontemporal table, or you can let the database do the temporal bookkeeping for you automatically by using the **FOR PORTION OF** syntax.

Unlike closed rows in system-time tables, history rows in valid-time tables remain accessible to all SQL operations. And because they model the real world, valid-time tables can have rows with a Period of Validity (PV) in the future, because things like contracts and policies may not begin or end until a future date.

We will look at some examples on the following pages.

For the examples on the following pages, consider the below valid-time table ***department***.

```
CREATE MULTISET TABLE department
   (dept INTEGER,
    mgr_EmpID INTEGER,
    valid_start DATE NOT NULL,
    valid_end DATE NOT NULL,
    PERIOD FOR dept_duration (valid_start, valid_end) AS VALIDTIME)
PRIMARY INDEX ( dept ) ;
```

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	9999-12-31

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	9999-12-31

When you INSERT a row, you need to provide the values for the beginning and ending bounds of the valid time (since they are NOT NULL).

```
INSERT INTO department
VALUES (200, 1011, DATE'2016-02-15', UNTIL_CHANGED);
```

You can use the UNTIL_CHANGED syntax or a date literal like '9999-12-31'. With either option, the resulting table will look like this:

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	9999-12-31
200	1011	2016-02-15	9999-12-31

Notice that you do not provide a value for the derived period column.

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	9999-12-31
200	1011	2016-02-15	9999-12-31

When you UPDATE a valid-time table *without* a Period of Applicability (PA), the behavior is the same as it would be for a nontemporal table.

```
UPDATE department SET mgr_EmpID = 1012 WHERE dept = 100;

UPDATE department SET valid_start = DATE'2016-03-01' WHERE dept =
200;
```

The resulting table will look like this:

dept	mgr_EmpID	valid_start	valid_end
100	**1012**	2015-01-01	9999-12-31
200	1011	**2016-03-01**	9999-12-31

Things to Notice:

- You can update the valid time directly.
- When you update any other column, the valid-time is not affected. This is different from system-time behavior. In this case, you are modifying "history" since you have updated the row to indicate that employee #1012 has been managing department 100 since January 2015.

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	9999-12-31
200	1011	2016-02-15	9999-12-31

Suppose, instead, that you want to update the manager for department 100, but **only from February 1, 2016 onward**. When you specify a date range, you can see how Teradata Database maintains the information in a time-aware way.

```
UPDATE department
FOR PORTION OF dept_duration
FROM DATE'2016-02-01' TO DATE'9999-12-31'
SET mgr_EmpID = 1012 WHERE dept = 100;
```

The resulting table will look like this:

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	**2016-02-01**
100	**1012**	**2016-02-01**	9999-12-31
200	1011	2016-02-15	9999-12-31

Things to Notice:

- You must reference the derived period column ***dept_duration*** in the **FOR PORTION OF** clause.
- The old row is updated to end at February 1, 2016.
- A new row is inserted, with the new manager, starting at February 1, 2016.

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	2016-02-01
100	1012	2016-02-01	9999-12-31
200	1011	2016-02-15	9999-12-31

Like UPDATE, when you DELETE from a valid-time table *without* a Period of Applicability (PA), the behavior is the same as it would be for a nontemporal table.

```
DELETE department WHERE dept = 200;
```

The resulting table will look like this:

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	2016-02-01
100	1012	2016-02-01	9999-12-31

Things to Notice:

- The row for department 200 is completely gone.

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	2016-02-01
100	1012	2016-02-01	9999-12-31
200	1011	2016-02-15	9999-12-31

Suppose, instead, that you want to DELETE the department 200, but **only for the year 2017**. You can use the same FOR PORTION OF syntax that we saw with UPDATE.

```
DELETE department
FOR PORTION OF dept_duration
FROM DATE'2017-01-01' TO DATE'2018-01-01'
WHERE dept = 200;
```

The resulting table will look like this:

dept	mgr_EmpID	valid_start	valid_end
100	1010	2015-01-01	2016-02-01
100	1012	2016-02-01	9999-12-31
200	1011	2016-02-15	**2017-01-01**
200	**1011**	**2018-01-01**	**9999-12-31**

Things to Notice:

- You must reference the derived period column ***dept_duration*** in the **FOR PORTION OF** clause.
- The old row is updated to end at January 1, 2017.
- A new row is inserted, starting at January 1, 2018, and ending at the maximum date.

Scenario: Where are the jeans?

To demonstrate a use of temporal tables, we will examine a scenario based in the retail industry.

Here is a case of a valid time temporal table in a retail environment. Each row seen here represents a change in the state of the pair of blue jeans being tracked.

Row #1 – The jeans are in a specific case, on a specific pallet, in a specific warehouse
Row #2 – The jeans have been loaded on a specific truck in transit to a store.
Row #3 – The jeans have arrived at a specific store.
Row #4 – The jeans have been sold and are no longer in the store.

On the next few pages, we will show some examples.

These examples show queries that are created to answer basic questions about inventory. The side-by-side queries are designed to show the difference in the complexity of coding needed to answer these questions, with and without a temporal database. In some cases the difference is slight. In others, the difference using a temporal database is dramatically simpler.

It is not the intention of these examples to demonstrate the SQL mechanics of each query; rather it is to show the difference in the complexity and length of the compared queries.

Question: How many Wrangler denim pants does store R have in the store now?

With ANSI Temporal Support

```
SELECT COUNT(*)
FROM (SELECT location, item_id
      FROM objectlocation_vt
      FOR VALIDTIME AS OF CURRENT_DATE) loc,
     product p
WHERE loc.location = 'Store R'
AND loc.item_id = p.item_id
AND p.item_name = 'denim pants'
AND p.brand = 'Wrangler';
```

Without Temporal Support

```
SELECT COUNT(*)
FROM objectlocation_nontemp loc,
     product p
WHERE loc.location = 'Store R'
AND loc.item_id = p.item_id
AND p.item_name = 'denim pants'
AND p.brand = 'Wrangler'
AND loc.start_dttm <= CURRENT_DATE
AND loc.end_dttm > CURRENT_DATE;
```

Things to Notice

- The ANSI temporal syntax is different from Teradata temporal syntax.
- In the temporal query, we set off the temporal qualifier in a subquery.
- In the nontemporal query, the user needs to know the names of the history handling columns.

Question: How many Wrangler denim pants did store R have in the store on December 26, 2015?

With ANSI Temporal Support

```
SELECT COUNT(*)
FROM (SELECT location, item_id
      FROM objectlocation_vt
      FOR VALIDTIME AS OF DATE'2015-12-26') loc,
     product p
WHERE loc.location = 'Store R'
AND loc.item_id = p.item_id
AND p.item_name = 'denim pants'
AND p.brand = 'Wrangler';
```

Without Temporal Support

```
SELECT COUNT(*)
FROM objectlocation_nontemp loc,
     product p
WHERE loc.location = 'Store R'
AND loc.item_id = p.item_id
AND p.item_name = 'denim pants'
AND p.brand = 'Wrangler'
AND loc.start_dttm <= DATE'2015-12-26'
AND loc.end_dttm > DATE'2015-12-26';
```

Things to Notice

- This is very similar to the current query on the previous page.
- In the temporal query, we set off the temporal qualifier in a subquery.
- In the nontemporal query, the user needs to know the names of the history handling columns.

Question: Which milk products have been in Store R for more than 7 days?

With ANSI Temporal Support

```
SELECT loc.item_id, loc.item_serial_num
FROM objectlocation_vt loc,
     product p
WHERE loc.location = 'Store R'
AND loc.item_id = p.item_id
AND p.item_name = 'milk'
AND loc.start_dttm <
   (CURRENT_TIMESTAMP - INTERVAL '7' DAY)
AND loc.end_dttm IS UNTIL_CHANGED;
```

Without Temporal Support

```
SELECT loc.item_id, loc.item_serial_num
FROM objectlocation_nontemp loc,
     product p
WHERE loc.location = 'Store R'
AND loc.item_id = p.item_id
AND p.item_name = 'milk'
AND loc.start_dttm <
   (CURRENT_TIMESTAMP - INTERVAL '7' DAY)
AND loc.end_dttm =
   '9999-12-31 23:59:59.999999+00:00';
```

Things to Notice

- The queries are almost identical for a ***nonsequenced*** query.
- The difference is that UNTIL_CHANGED can only be used in relation to a PERIOD data type.
- You can use the hard-coded maximum time stamp with Temporal but you cannot use UNTIL_CHANGED with the nontemporal table.

Question: A laptop (item id 150, serial number 101) was found to be broken due to shock. Which other items were on the same route (same truck) at the same time?

With ANSI Temporal Support

```
SELECT t2.item_id, t2.item_serial_num
FROM objectlocation_vt t1,
     objectlocation_vt t2
WHERE t1.item_id = 150
AND t1.item_serial_num = 101
AND t1.location = 'Route to Store R'
AND t1.location = t2.location
AND t1.duration OVERLAPS t2.duration;
```

Without Temporal Support

```
SELECT t2.item_id, t2.item_serial_num
FROM objectlocation_nontemp t1,
     objectlocation_nontemp t2
WHERE t1.item_id = 150
AND t1.item_serial_num = 101
AND t1.location = 'Route to Store R'
AND t1.location = t2.location
AND (t1.start_dttm, t1.end_dttm) OVERLAPS (t2.start_dttm, t2.end_dttm);
```

Things to Notice

- The queries are almost identical. They each have a self join to connect the laptop information with information about other items.
- You can use the OVERLAPS function (and many other PERIOD functions) with a derived temporal period or a period expression.

We will revisit this scenario after we discuss bitemporal tables.

Bitemporal Tables

What is a bitemporal table?

Bitemporal tables have both a valid time and system time dimension. Queries against a bitemporal table can be done using valid time qualifiers, system time qualifiers or both.

The rules for both valid time tables and system time tables apply to a bitemporal table.

When should you use a bitemporal table?

Bitemporal tables are most useful when you need the ability to completely re-construct the history of a table at a detailed level.

Recall that the valid time informs us when a particular fact was valid, whereas the system time informs us when something was entered into (or known by) the database.

In the bitemporal example below, you get a clear picture of what has happened to this employee and her salary.

Employee 1001 was hired on June 1, 2015.

On November 26, 2015, she got a raise with the effective date of December 1, 2015. This causes the following 3 changes:

1. **UPDATE** - The old row with the UNTIL_CHANGED valid time is closed in system-time (row #3 below).
2. **INSERT** - A new row with the old salary but a valid-time ending bound of December 1 is inserted, open in system-time (row #2 below).
3. **INSERT** - A new row with the new salary with a valid-time beginning on December 1 is inserted, open in system-time (row #1 below).

empno	salary	valid_start	valid_end	sys_start	sys_end
1001	50000	2015-12-01	9999-12-31	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01	2015-12-01	2015-11-26 09:49:44.520000+00:00	9999-12-31 23:59:59.999999+00:00
1001	40000	2015-06-01	9999-12-31	2015-06-01 09:47:22.520000+00:00	2015-11-26 09:49:44.520000+00:00

How big will my table get?

A temporal table will require more space than a nontemporal table. But how much more? It depends on the frequency of changes that are captured.

For example, let's look at a ***customer*** table. The following metrics can be used to estimate the additional storage requirements.

- Address Changes: approximately 16% of people move per year
- Marital/Family Status Changes:
  - approximately 4M births in US per year (1.4% of population)
  - approximately 2.4M deaths in US per year (.8%)
  - approximately 2M marriages in US per year (.74% x2)
  - approximately 1M divorces in US per year (.37% x2)

  Total approximately 4.4% of people change marital or family status per year

The aggregate of the changes experienced by customers suggests that the table will grow by approximately 20.4% (16% + 4.4%) as new rows are added to support these changes.

Scenario (Revisited): Where are the jeans?

We recall the scenario we examined before, but now we add a system time dimension.

Consider the scenario seen in a prior module.

The three rows seen in this diagram tell a story about an experience with a specific pair of jeans.

Row #1 – The first row show us that between 08-24 and 08-25 the jeans were known to be in a truck on their way to the store.
Row #2 – This row tells us that the jeans were 'lost' between 08-25 and 08-27. We are not sure if this is due to lost merchandise or a scanner failure at the store.
Row #3 - This row tells us that the jeans were found on 08-27 at the store so that we are able to conclude that the jeans were never really 'lost' but that a scanner failure made them appear to be lost.

How would we be able to recreate this event using the data in temporal tables?

Well, it depends on what type of temporal table is used. We will now look at how this information would be presented in a valid-time table, a system-time table and a bitemporal table.

What does a bitemporal table tell us?

item_id	item_serial_num	location	valid_begin	valid_end	sys_start*	sys_end*
125	102	warehouse	2015-08-04	9999-12-31	2015-08-04	2015-08-24
125	102	warehouse	2015-08-04	2015-08-24	2015-08-24	9999-12-31
125	102	transit	2015-08-24	9999-12-31	2015-08-24	2015-08-27
125	102	transit	2015-08-24	2015-08-25	2015-08-27	9999-12-31
125	102	store	2015-08-25	9999-12-31	2015-08-27	9999-12-31

Row #1 – The jeans entered the warehouse on 08-04.

Row #2 – The jeans left the warehouse on 08-24. Row #1 is closed and inserted Row #2 is open.

Row #3 – The jeans were put on the truck on 08-24. Initially, this row is open.

Row #4 – The truck arrived at the store on 08-25. Notice the system time here is 08-27. This tells us that it was not until 08-27 that we figured out that the jeans did actually make it to the store. In other words, we found the jeans in the store on 08-27 and the data reflects that.

Row #5 – The jeans are in the store on 08-25. Again, notice the system time is 08-27. We didn't know where the jeans were until 08-27, but we have concluded that they had been in the store all along.

In short, **we are able to recreate exactly what we knew and when we knew it using the bitemporal table**.

* The system-time values are truncated to dates to make this easier to read.

What does a valid-time table tell us?

item_id	item_serial_num	location	valid_begin	valid_end
125	102	warehouse	2015-08-04	9999-12-31
125	102	warehouse	2015-08-04	2015-08-24
125	102	transit	2015-08-24	9999-12-31
125	102	transit	2015-08-24	2015-08-25
125	102	store	2015-08-25	9999-12-31

What if we had opted to use a valid-time table for this application instead of a bitemporal table?

Rows #4 and #5 correctly show the jeans being in the store on 08-25. But there is no way of knowing when we knew this since there is no system time information. Thus, it is not obvious that we had a problem with missing jeans or what the solution was.

In short, **the valid time table accurately tracks the movement of the jeans, but it does not give us any indication that the jeans were presumed missing for a period**.

What does a system-time table tell us?

item_id	item_serial_num	location	sys_start*	sys_end*
125	102	warehouse	2015-08-04	2015-08-24
125	102	warehouse	2015-08-24	9999-12-31
125	102	transit	2015-08-24	2015-08-27
125	102	transit	2015-08-27	9999-12-31
125	102	store	2015-08-27	9999-12-31

What if we had opted to use a system-time table for this application instead of a bitemporal table?

If we look at system time alone, it tells us that the jeans arrived in the store on 08-27, which is not true. The jeans actually arrived on 08-25, however the database did not know about it until 08-27. So system time alone does not tell us that the jeans were ever considered missing. It leaves the incorrect impression that the jeans were in transit for three days.

* The system-time values are truncated to dates to make this easier to read.

item_id	item_serial_num	location	valid_begin	valid_end	sys_start*	sys_end*
125	102	warehouse	2015-08-04	9999-12-31	2015-08-04	2015-08-24
125	102	warehouse	2015-08-04	2015-08-24	2015-08-24	9999-12-31
125	102	transit	2015-08-24	9999-12-31	2015-08-24	2015-08-27
125	102	transit	2015-08-24	2015-08-25	2015-08-27	9999-12-31
125	102	store	2015-08-25	9999-12-31	2015-08-27	9999-12-31

A bitemporal table allows us to get whatever level of detail we might find useful for researching a historical incident such as the 'lost jeans'.

Teradata's implemention of ANSI temporal syntax supports both valid time and system time:

- Valid-Time for modeling slow changing dimensions
- System-Time for auditing and past query replication

Bitemporal support adds more information. History can be more completely reconstructed with both valid- and system-time.