Reliable Context

RELIABLE CONTEXT / RESEARCH & PRACTICAL GUIDANCE

SQL:2011 Temporal Tables in Practice

> Reliable Context is a kernel of truth: a system of record for organizations that fuse private and public information and must defend every value they act on. One canonical record per real thing, a cited source and an observation date behind every value, unknowns declared rather than left blank, and any past state of the record reproducible.

Published by Reliable Context · 2026-10-29 · sql 2011 temporal tables · system versioned temporal tables · temporal tables sql server · mariadb system versioned tables · bitemporal sql

Original article: https://reliablecontext.com/articles/sql-2011-temporal-tables-system-versioned-in-practice



In this article

1. What SQL:2011 standardized
2. What each engine implements
3. Worked example: a back-dated suspension
4. What system versioning does not record
5. How history can be altered or purged
6. Native bitemporal stores
7. Migration checklist
8. Sources

What SQL:2011 standardized

SQL:2011, published by ISO/IEC in December 2011, added temporal tables to Part 2 (SQL/Foundation) of the SQL standard. The standard text is sold by ISO, whose public abstract says only that ISO/IEC 9075-2:2011 "defines the data structures and basic operations on SQL-data"; that edition is now withdrawn, and the current edition of Part 2 is ISO/IEC 9075-2:2023 (ISO, 9075-2:2011; ISO, 9075-2:2023). The readable public account of the temporal features is the paper by Krishna Kulkarni and Jan-Eike Michels of IBM, which this section follows (Kulkarni and Michels 2012). Statements below describe SQL:2011 as that paper reports it; the article does not compare later editions clause by clause.

The building block is the **period**: a named table component that points at two ordinary columns, a start and an end. SQL:2011 did not add a period data type. A period uses **closed-open** semantics, meaning it includes its start and excludes its end, and declaring one implies a constraint that the end is later than the start (Kulkarni and Michels 2012, section 2.1).

There are two kinds of period, and a table may have at most one of each:

- A **system-time period**, always named `SYSTEM_TIME`. The database sets its columns. A table with it, declared `WITH SYSTEM VERSIONING`, is a **system-versioned table**. This supports what the literature calls transaction time.
- An **application-time period**, with any name. The user sets its columns, to past, present or future values. This supports valid time.

A table with both is commonly called **bitemporal**, a term the paper uses although the standard does not define it. The conceptual distinction between the two time axes, and why neither can be derived from the other, is covered in valid, transaction, observation and publication time; this article stays with the SQL.

What the standard specifies for each kind, per Kulkarni and Michels:

Feature	Application-time period	System-time period
Who sets start and end	The user, any value	The database, at the transaction timestamp; users may not assign or change them
Columns declared as	Ordinary `DATE` or timestamp columns	`GENERATED ALWAYS AS ROW START` / `ROW END`
Partial-period changes	`UPDATE` / `DELETE ... FOR PORTION OF p FROM x TO y` split rows	Not needed and not allowed
Keys	`PRIMARY KEY (id, p WITHOUT OVERLAPS)` forbids overlapping periods per key	Ordinary keys, enforced on current rows only
Foreign keys	`FOREIGN KEY (col, PERIOD p) REFERENCES ...` requires the parent to cover the child's whole period	Ordinary foreign keys, current rows only
Querying	Ordinary predicates, plus `CONTAINS`, `OVERLAPS`, `EQUALS`, `PRECEDES`, `SUCCEEDS` and the `IMMEDIATELY` forms	`FOR SYSTEM_TIME AS OF`, `FROM ... TO`, `BETWEEN ... AND`

Two rules on system-versioned tables matter for audit use. `UPDATE` and `DELETE` act only on current rows, and "users are not allowed to update or delete historical system rows" (Kulkarni and Michels 2012, section 2.3). A query with no `FOR SYSTEM_TIME` clause behaves as `AS OF CURRENT_TIMESTAMP`, so existing applications see only current rows after a table is converted. Period joins and period aggregates are listed in the paper as future work, not SQL:2011 features. At the time of writing the authors knew of one commercial implementation, citing IBM DB2 10.

What each engine implements

Products implement different subsets, with different syntax and storage. The table summarizes the vendor documentation as read on 2026-10-28.

Capability	SQL Server / Azure SQL	MariaDB	PostgreSQL 18
System-versioned tables	Yes, SQL Server 2016 and later (Microsoft)	Yes (MariaDB)	No; PostgreSQL 19 documentation says system time is not supported (PostgreSQL)
Where history lives	Separate history table	Same table, optionally partitioned `BY SYSTEM_TIME`	Not applicable
`FOR SYSTEM_TIME` forms	`AS OF`, `FROM TO`, `BETWEEN AND`, `CONTAINED IN`, `ALL`	`AS OF`, `BETWEEN AND`, `FROM TO`, plus non-standard `ALL`	Not applicable
Application-time period	Not in Microsoft's temporal documentation, which covers system time	`PERIOD FOR name (start, end)` (MariaDB)	One range or multirange column, not a column pair
`FOR PORTION OF`	Not offered	`UPDATE` and `DELETE`	Absent from the PostgreSQL 19 `UPDATE` synopsis (PostgreSQL)
`WITHOUT OVERLAPS` key	Not offered	Primary key or unique index	`PRIMARY KEY` and `UNIQUE`, new in 18 (release notes)
Temporal foreign key	Not offered	Not documented	`PERIOD` foreign keys, new in 18
Built-in history purge	`HISTORY_RETENTION_PERIOD`	`DELETE HISTORY`, dropping history partitions	Not applicable

SQL Server

A SQL Server temporal table is a pair: the current table and a history table with a mirrored schema. Every table declares two `datetime2` period columns. On insert, the start column is set to the begin time of the current transaction in UTC and the end column to `9999-12-31`. On update or delete, the previous row version is copied to the history table with its end set to the transaction begin time (Microsoft, Temporal tables). A temporal table must have a primary key; the history table cannot have constraints, and `INSERT` and `UPDATE` cannot reference the period columns (Microsoft, Considerations and limitations).

A naming warning: Microsoft's examples call the system-time columns `ValidFrom` and `ValidTo`. They record when the database held a row, not when the fact was valid in the world. Naming them `SysStart` and `SysEnd` avoids that confusion once you add real valid-time columns.

The fictional example used throughout: Halden Therapeutics AG keeps a supplier qualification register. In SQL Server:

```
-- Fictional example: Halden Therapeutics AG
CREATE TABLE dbo.SupplierQualification
(
    SupplierID    INT            NOT NULL PRIMARY KEY CLUSTERED,
    Status        VARCHAR(20)    NOT NULL,
    ChangedBy     NVARCHAR(128)  NOT NULL,
    ChangeReason  NVARCHAR(400)  NULL,
    SysStart      DATETIME2 GENERATED ALWAYS AS ROW START,
    SysEnd        DATETIME2 GENERATED ALWAYS AS ROW END,
    PERIOD FOR SYSTEM_TIME (SysStart, SysEnd)
)
WITH (SYSTEM_VERSIONING = ON (
        HISTORY_TABLE = dbo.SupplierQualificationHistory,
        HISTORY_RETENTION_PERIOD = 10 YEARS));

-- The row as the database held it at 2026-03-25 00:00 UTC
SELECT SupplierID, Status, SysStart, SysEnd
FROM dbo.SupplierQualification
    FOR SYSTEM_TIME AS OF '2026-03-25T00:00:00'
WHERE SupplierID = 4411;
```

`AS OF t` returns rows where start is at most `t` and end is greater than `t`. `FOR SYSTEM_TIME` hides zero-duration rows, which SQL Server creates when one transaction updates the same key several times; to see them, query the history table directly (Microsoft, Temporal tables).

`HISTORY_RETENTION_PERIOD` (SQL Server 2017 and later) accepts `DAYS`, `WEEKS`, `MONTHS` or `YEARS`, defaults to `INFINITE`, and needs the database option `TEMPORAL_HISTORY_RETENTION` to be on before any cleanup happens. A background task deletes aged rows in arbitrary order; temporal queries filter them out first, while direct reads of the history table may still show them (Microsoft, Manage historical data). How long a register must be kept is a records-management and legal question for your jurisdiction; this article is not legal advice.

MariaDB

MariaDB accepts the standard declaration and a shorthand, `CREATE TABLE t (x INT) WITH SYSTEM VERSIONING`, which keeps history in hidden pseudo-columns `ROW_START` and `ROW_END` (MariaDB, System-versioned tables). History is stored in the same table by default; partitioning `BY SYSTEM_TIME` moves it into history partitions, rotated by `LIMIT` (rows, approximate) or `INTERVAL`, with `AUTO` creation from 10.9.1.

MariaDB also implements application-time periods with `FOR PORTION OF` and `WITHOUT OVERLAPS` (MariaDB, Application-time periods), and a table may combine both period kinds (MariaDB, Bitemporal tables). That makes it the engine in this comparison that comes closest to the full SQL:2011 temporal feature set. The

bitemporal version of the Halden register:

```
-- Fictional example: Halden Therapeutics AG
CREATE TABLE supplier_qualification (
  supplier_id    INT          NOT NULL,
  status         VARCHAR(20)  NOT NULL,
  eff_from       DATE         NOT NULL,
  eff_to         DATE         NOT NULL,   -- '9999-12-31' = open-ended
  changed_by     VARCHAR(100) NOT NULL,
  change_reason  VARCHAR(400),
  row_start TIMESTAMP(6) AS ROW START INVISIBLE,
  row_end   TIMESTAMP(6) AS ROW END INVISIBLE,
  PERIOD FOR effective (eff_from, eff_to),
  PERIOD FOR SYSTEM_TIME (row_start, row_end),
  UNIQUE (supplier_id, effective WITHOUT OVERLAPS)
) WITH SYSTEM VERSIONING;
```

The `system_time` period cannot be used in `FOR PORTION OF`; only the application-time period can.

PostgreSQL

PostgreSQL 18, the current release on postgresql.org, added `WITHOUT OVERLAPS` for `PRIMARY KEY` and `UNIQUE` and `PERIOD` for foreign keys, applied to the last key column (PostgreSQL 18 release notes). That column must be a range or multirange type, and the constraint behaves like an exclusion constraint using GiST (PostgreSQL 18, CREATE TABLE). The PostgreSQL 19 documentation, published with release candidate 1 on 15 October 2026 (general availability is planned for 19 November 2026, per the RC 1 announcement), adds a temporal tables chapter. It says that creating a temporal key in practice requires the `btree_gist` extension, that temporal foreign keys support only `NO ACTION`, and that "PostgreSQL does not currently support system time" beyond emulation with triggers or external extensions (PostgreSQL 19, Temporal tables).

This ran as shown on a local PostgreSQL 18.6 instance:

```
-- Fictional example: Halden Therapeutics AG
CREATE EXTENSION btree_gist;
CREATE TABLE dosage_label (
  product_id   integer,
  max_daily_mg integer NOT NULL,
  valid_at     daterange,
  PRIMARY KEY (product_id, valid_at WITHOUT OVERLAPS)
);
CREATE TABLE batch_release (
  batch_id   integer,
  product_id integer,
  valid_at   daterange,
  PRIMARY KEY (batch_id, valid_at WITHOUT OVERLAPS),
  FOREIGN KEY (product_id, PERIOD valid_at)
```

```
    REFERENCES dosage_label (product_id, PERIOD valid_at)
);
```

Inserting a label for product 7 that overlapped an existing one failed with "conflicting key value violates exclusion constraint". A batch whose period started before any label existed failed the foreign key. A batch spanning two adjacent label rows succeeded, because the parent rows together covered it. PostgreSQL thus enforces valid-time integrity but keeps no history of its own changes.

Worked example: a back-dated suspension

All names and dates are fictional. Using the MariaDB table above:

1. On 2026-03-10 a clerk records supplier 4411 as `Approved`, effective 2026-03-10 to open end. MariaDB sets `row_start` to the insert time.
2. On 2026-04-02 at 09:14 a quality auditor learns that the supplier's certificate was withdrawn on 2026-03-20. She records the suspension from that date without erasing what was believed before:

```
UPDATE supplier_qualification
  FOR PORTION OF effective FROM '2026-03-20' TO '9999-12-31'
  SET status = 'Suspended',
      changed_by = 'qa.auditor.2',
      change_reason = 'Certificate withdrawn 2026-03-20; finding QA-0412'
  WHERE supplier_id = 4411;
```

`FOR PORTION OF` splits the row: an `Approved` row now covers 2026-03-10 to 2026-03-20 and a `Suspended` row covers 2026-03-20 onward. System versioning closes the old single `Approved` row at 2026-04-02 09:14 and keeps it as history. Two questions now have different, correct answers:

```
-- What did the register say on 2026-03-25 about 2026-03-25?
SELECT status FROM supplier_qualification
  FOR SYSTEM_TIME AS OF TIMESTAMP '2026-03-25 00:00:00'
WHERE supplier_id = 4411
  AND eff_from <= '2026-03-25' AND eff_to > '2026-03-25';

-- What do we know today about 2026-03-25?
SELECT status FROM supplier_qualification
WHERE supplier_id = 4411
  AND eff_from <= '2026-03-25' AND eff_to > '2026-03-25';
```

Question	Mechanism	Answer
Status on 25 March, as recorded on 25 March	`AS OF` plus valid-time predicate	Approved
Status on 25 March, as known now	Current rows plus valid-time predicate	Suspended
When the register changed	`row_start` of the new rows	2026-04-02 09:14

Question	Mechanism	Answer
Who changed it and why	`changed_by`, `change_reason`	Only because the table has those columns

The first answer explains why a purchase order placed on 25 March passed an automated check. The second is the corrected fact. A system-versioned table alone gives the first and the third; an application-time period alone gives the second; neither gives the fourth. Re-running a past decision against an as-of query is the core of reproducing a past state.

What system versioning does not record

System versioning records the database's own change history: which row values existed and between which system timestamps. It does not record:

- **Who made the change.** Neither the standard's period columns nor the vendor implementations above store a user. If you need it, add a column and have the application or a trigger write it on every change, as in the examples.
- **Why.** A reason, ticket or evidence reference exists only if you model it.
- **When the fact became true.** That is valid time, which needs an application-time period or your own columns.
- **Commit time, by default.** SQL Server uses the transaction's begin time. MariaDB notes that a row inserted in a long transaction "became visible hours after it was inserted", and offers transaction-precise history on InnoDB with `BIGINT UNSIGNED` transaction-ID columns, which cannot be combined with `PARTITION BY SYSTEM_TIME` (MariaDB).
- **Whether anything actually changed.** SQL Server adds a history row for any modification "even if no column values change"; MariaDB does the same for any `UPDATE` whose `SET` list names a versioned column.

How history can be altered or purged

The standard forbids users from changing historical rows. The products give privileged users documented ways to do so, mostly for retention:

SQL Server. While `SYSTEM_VERSIONING` is `ON`, history cannot be modified directly by anyone, whatever their permissions. Turning versioning off requires `CONTROL` on both tables (Microsoft, Temporal table security). Microsoft's own custom cleanup procedure does exactly that: it sets versioning `OFF`, deletes from the history table, and sets it back `ON` with `DATA_CONSISTENCY_CHECK = OFF`, inside one transaction (Microsoft, Manage historical data). Partitions can be switched out of the history table while versioning stays on. Dropping a column from the current table also drops it from history, given `ALTER` permission on both. Auditing settings on the current table do not carry over to the history table; you must enable them there explicitly.

MariaDB. `DELETE HISTORY FROM t BEFORE SYSTEM_TIME '...'` prunes history and requires the `DELETE HISTORY` privilege. Dropping and re-adding system versioning deletes all history, as does dropping history partitions. With `system_versioning_alter_history = KEEP`, `ALTER TABLE` is allowed but "the history will

become incorrect". Setting the `timestamp` session variable changes the `row_start` assigned to new rows, and `system_versioning_insert_history` permits direct inserts into `ROW_START` and `ROW_END` where `secure_timestamp` allows. `mariadb-dump` omits history unless run with `--dump-history`, available from 10.11 (MariaDB).

System versioning is therefore a reconstruction aid, not a tamper-evident record. If history must be defensible against an administrator, enable the database audit on the history table and on the privileges above, restrict `CONTROL` and `DELETE HISTORY`, and check that backups include history.

Native bitemporal stores

Some databases make both time axes the default rather than an option. XTDB maintains `_system_from`, `_system_to`, `_valid_from` and `_valid_to` on every table; system time is maintained automatically and valid time is user-managed (XTDB, Key concepts). Datomic offers an `as-of` filter that returns the database at a transaction id or instant, and a `history` view of present and past including retractions; its filters page does not address valid time (Datomic, Filters). For how these compare with temporal knowledge graphs, see bitemporal versus temporal knowledge graphs.

Migration checklist

1. **Name the axes.** Decide which columns are system time and which are valid time, and name them so nobody confuses the two.
2. **Check the primary key.** SQL Server requires one. In MariaDB, system versioning implicitly adds `row_end` to the primary key, which can cause duplicate-key errors when statements are replayed from the binary log; `secure_timestamp = YES` on the replica is the documented mitigation.
3. **Add accountability columns.** Who, why and evidence reference, written on every change.
4. **Model valid time** with an application-time period (MariaDB) or range column (PostgreSQL), and protect it with `WITHOUT OVERLAPS`.
5. **Set retention deliberately** and record who may purge, through `HISTORY_RETENTION_PERIOD`, partitions or `DELETE HISTORY`.
6. **Grant history access separately.** In SQL Server, `SELECT` on the current table does not imply `SELECT` on the history table.
7. **Audit the escape hatches**: versioning toggles, `DELETE HISTORY`, timestamp overrides.
8. **Test reconstruction**: restore a backup and run one known `AS OF` query against it.

A system of record is easier to trust when a reader can ask both what was true on a date and what the system said on that date, and get two separately stored answers.

Sources

1. Kulkarni and Michels, "Temporal features in SQL:2011", SIGMOD Record 41(3), 2012: SQL:2011 periods, application-time and system-versioned semantics, constraints, query forms, future work.
2. Microsoft Learn, Temporal tables: SQL Server mechanics, `FOR SYSTEM_TIME` subclauses, timestamps.

3. Microsoft Learn, Manage historical data in system-versioned temporal tables: retention policy and custom cleanup.
4. Microsoft Learn, Temporal table security: permissions and auditing.
5. Microsoft Learn, Temporal table considerations and limitations: table restrictions.
6. MariaDB, System-versioned tables: syntax, partitioning, purging, system variables, limitations.
7. MariaDB, Application-time periods: `PERIOD FOR`, `FOR PORTION OF`, `WITHOUT OVERLAPS`.
8. MariaDB, Bitemporal tables: combined periods; `system_time` excluded from `FOR PORTION OF`.
9. PostgreSQL 18 release notes: `WITHOUT OVERLAPS` keys and `PERIOD` foreign keys added in 18.
10. PostgreSQL 18, CREATE TABLE: range or multirange column, exclusion-constraint behaviour.
11. PostgreSQL 19, Temporal tables: `btree_gist`, `NO ACTION` only, system time not supported.
12. PostgreSQL 19, UPDATE: synopsis without `FOR PORTION OF`.
13. XTDB, Key concepts: four temporal columns, system time automatic, valid time user-managed.
14. Datomic, Filters: `as-of`, `since` and `history` views.
15. ISO/IEC 9075-2:2011, ISO: public abstract; edition withdrawn.
16. ISO/IEC 9075-2:2023, ISO: current edition of SQL/Foundation.
17. PostgreSQL 19 RC 1 Released, PostgreSQL Global Development Group, 15 October 2026: release candidate date and planned general availability.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://www.iso.org/standard/53682.html
2. https://www.iso.org/standard/76584.html
3. https://sigmodrecord.org/publications/sigmodRecord/1209/pdfs/07.industry.kulkarni.pdf
4. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal/overview
5. https://mariadb.com/docs/server/reference/sql-structure/temporal-tables/system-versioned-tables
6. https://www.postgresql.org/docs/19/ddl-temporal-tables.html
7. https://mariadb.com/docs/server/reference/sql-structure/temporal-tables/application-time-periods
8. https://www.postgresql.org/docs/19/sql-update.html
9. https://www.postgresql.org/docs/18/release-18.html
10. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal/considerations-limitations
11. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal/manage-retention
12. https://mariadb.com/docs/server/reference/sql-structure/temporal-tables/bitemporal-tables
13. https://www.postgresql.org/docs/18/sql-createtable.html
14. https://www.postgresql.org/about/news/postgresql-19-rc-1-released-3409
15. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal/security
16. https://docs.xtdb.com/concepts/key-concepts.html
17. https://docs.datomic.com/reference/filters.html

THE PUBLISHER

About Reliable Context

Reliable Context is a system of record for organizations that fuse private and public information and must defend every value they act on.

The guarantees

One canonical record per real thing: duplicates resolve by rule, ambiguity is shown rather than hidden, and related organizations relate without ever being merged. A verbatim, cited source and an observation date sit behind every value. Unknowns are declared as not found, blocked or not applicable with a reason, never left as an empty cell. Any past state of the record is reproducible, together with what changed since.

Where it runs

It runs inside the customer's perimeter, one database per organization. Permitted sources, model routing, reader clearance and retention are set as the customer's own policies, and every value is exportable with its chain of custody for an auditor or regulator.

What it is not

It is not a vector database or a RAG framework, not a data catalog, MDM or data-quality suite, not a CRM or data-enrichment vendor, and not a chatbot or AI assistant. Where a reader needs one of those, the honest answer is to say so and point them there.

Visit

See Reliable Context.

Reliable Context is in production use in two sectors; more than six thousand of its identity decisions were made by rule and none has been reversed. Articles published here explain general practice, cite their primary and academic sources, and name the jurisdiction of any regulatory rule they describe. They are not legal, regulatory or compliance advice for a specific organization, and requirements change.

Website: https://reliablecontext.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.