## Introduction

This is a description of a few useful ways of tracking activities and changes in your database. These options are not comprehensive, of course – there are countless ways to track changes.

These are the 3 categories that we've found successfully applied most often.

They each serve a different purpose, and they're not mutually exclusive – depending on the business case, it may make sense to use one or another … or even incorporate all three.

## More Information / Consulting Services

Visit: www.scoutcorpsllc.com or www.hipaapotamus.com

Mail: info@scoutcorpsllc.com

## Trace Tables

In this model, each insert, delete, update, and read operation creates a row in a tracking table separate from the table that was the subject of the operation. You're tracking **who** did something, **when**, and (potentially) **how** and **why**.

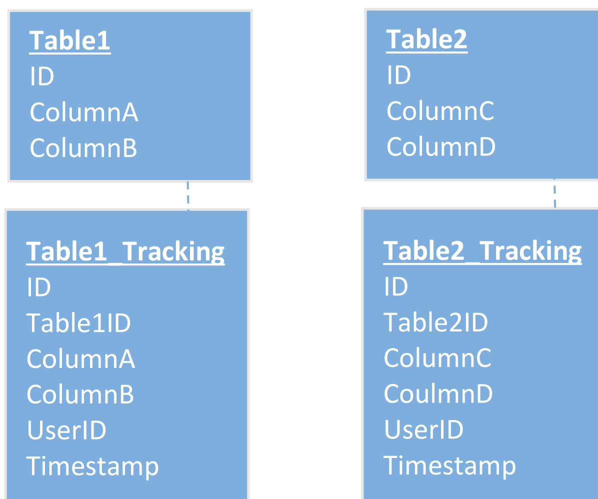
*Note: depending on the use case, you may choose only a subset of these operations. For example, choosing to only log operations on certain tables or columns, or to only log deletions and edits (ignoring inserts and views).*

*Also note: this is written from the perspective of a relational database, but if you replace "tracking table" with "collection" or "log file," it could apply to other data storage systems as well.*

We'll cover 4 different implementations.

### (1) Naïve Implementation (a tracking table for each tracked data table)

Simply have a tracking table for each table in the database you want to track actions on. Each tracking table will have the same columns as the original table, plus a column for the user and timestamp. Insert a copy of the row into the tracking table when deleting or updating the data table.



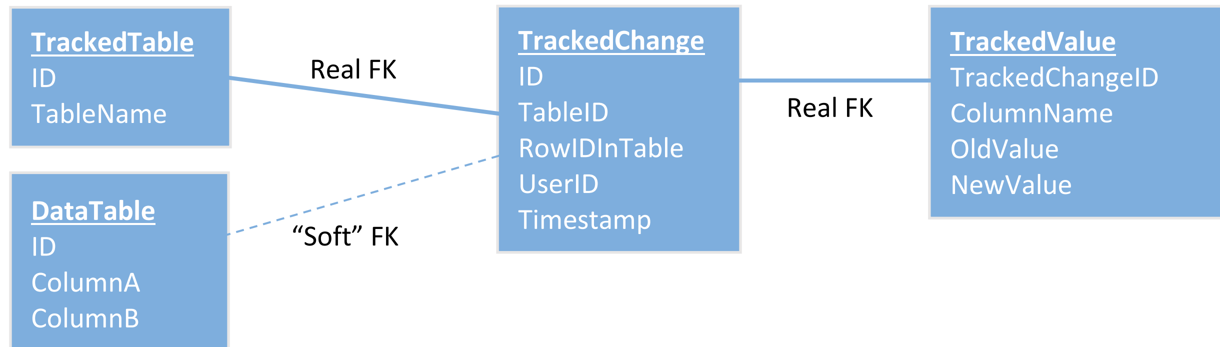

Note that there is no foreign key relationship, rows may be deleted from the data tables and the tracking history should be maintained.
*An exception: if your data model didn't allow for deletions – if rows were only "archived" or hidden from the UI – a foreign key would be useful.*

**Pros:** Conceptually simple – it's simple to query and easy to grasp the design.

**Cons:** A lot of rote work must be remembered and performed by engineers – requires creating a near-copy of every table's schema and keep those definitions in sync with the data tables; every tracking row must be created individually in your queries or ORM.

## (2) Universal Solution (a single tracking schema)

A denormalized TrackedChange / TrackedValue model.




Define a "soft" foreign key to the underlying data tables, expressed as a combination of [Table Identifier] and [Table's primary key]. The table "TrackedTable" would be a list of all tables in the database on which change tracking is implemented, the TableID of a tracked change would indicate in which table data was changed, and the RowIDInTable would point to the primary key in whichever table that was.

Usage would be simpler – there would be a function in the application's code that would take the TableID, the ID of the record in question, the user name, and a list of [column being updated] + [old value] + [new value].

Ideally, this would be implemented using generics (in a strongly-typed language) – the application function could then just take the old version of the object, the new version of the object, and the user's name. The list of columns being updated along with their old and new values could then be determined using reflection (or property enumeration, in a weakly-typed language); the table ID and record's ID would be known from the type of object passed in.

*Note: The TrackedValue table could also be replaced by an OldValues and NewValues JSON column on the TrackedChange table, depending on what the needs would be for downstream querying.*

**Pros:** Fewer tables, no schema synchronization. Sparser storage than (1), as you're only storing values for columns that have changed.

**Cons:** Conceptually more complex than (1), making it easier to write bugs in implementation. Reduces but doesn't *eliminate* the need to remember – the function still needs to be *invoked* when appropriate.

## (3) Automated Solution (designing a database to utilize framework features)

Using a combination of a data model (such those outlined in the other three implementations) and built-in database functionality (especially triggers), make change tracking happen automatically in the background.

Create a trigger on each table that tracks the timestamp and user whenever a row is inserted, updated, deleted, or read.

Application users must each have their own user account on the underlying database – if the application makes changes as a "service account," the actual user behind them won't be recorded.

There may be advantageous side-effects to giving each application user (or client company) their own SQL user, such as being able to use row-level-security, but also harmful ones, especially relating to scalability.

**Pros:** Foolproof; no action will go unrecorded. Must create a trigger when a new table is created, otherwise, no new code is needed.

**Cons:** Unable to track any information that doesn't exist at the database level (comments, source files, etc.). UI action grouping unavailable. Requires specific design decisions that affect the application more broadly.

This could also be considered as a fallback solution, and/or an error-catching tool in conjunction with (1) or (2) – anything recorded via this methodology but not *also* recorded through the other represents a bug to be addressed.

### (4) Full point-in-time database

If you're familiar with the concept of "soft deletes", think of this as "soft deletes *and* soft updates". In this model, we treat history as a first-order object in our database.

See "Point-in-Time Data – Irregular Validity Periods" heading below. A similar model can be used for tracking, as well.

| **DataTable** |
| --- |
| ID |
| LogicalID |
| ColumnA |
| CoulmnB |
| UserID |
| CreatedTimestamp |
| DeletedTimestamp |

This can be made foolproof by granting the application permissions to *create* rows in the relevant tables but **not** to *delete* rows from the relevant tables, or to *update* any column aside from the ValidToDate.

**Pros:** (Can be made) foolproof; no action will go unrecorded. Easy to implement when building a database from scratch.

**Cons:** Breaking change – requires changes to the schema of existing tables and (potentially, depending on the existing design) all code that interacts with them. Can result in confusing design when paired with data that is *already* point-in-time, time series, or both.

## ORM Mitigation

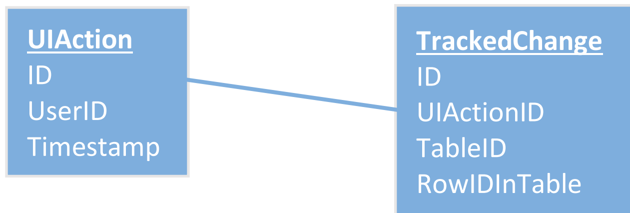
Either (1), (2), or (4) could have the manual steps / forgetfulness risk mitigated by integrating them into a wrapper for your ORM.

Every function in your ORM-wrapper would create the proper tracking records. By removing the application's direct reference to the ORM and forcing all database access to go through the ORM-wrapper, you'd be limiting tracking to a single component; the assumption during development and review would be that any new function in the ORM-wrapper requires tracking code, and that any code outside of the ORM-wrapper doesn't.

Of course, those tracking calls would still have to be written in the ORM wrapper, but it'd be easier to corral and review.

## UI Grouping

Either (1), (2), or (4) could be further augmented by grouping changes originating from the same UI interaction. In most modern applications, there are situations in which a "single" action – from a user's perspective – results in multiple actions performed on the database.



## When to Track

To address the question of *when* the change tracking records are created (in relation to when the underlying commands are run):

- Track the changes first
    - **Pros:** Changes will always be tracked. If the underlying operation fails, there will be “false” changes tracked, but these will be useful for debugging.
    - **Cons:** Delays the underlying operation. A failure to track changes causes the operation to fail – this may be desired, but often isn’t.
- Track the changes last
    - **Pros:** Least disruption and/or delay to the application. Changes can even be tracked asynchronously after a success result is returned to the caller.
    - **Cons:** Errors result in untracked changes. Not suitable for mission-critical tracking.
- Track the changes first *and* last
  *Create an “in-progress” change first, update the status to “completed” afterward*
    - **Pros:** Changes will always be tracked accurately. Very clear for debugging.
    - **Cons:** Double the computational and I/O overhead.
- Use a transaction
    - **Pros:** All the advantages of the first + last method, without the drawback of double writes.
    - **Cons:** Can create deadlocks on high-throughput applications or with long-running actions. Difficult to implement when actions are split across multiple commands. Impossible to implement on databases, such as many document stores, that don’t support transactions. Loses the debugging advantages of the first + last method.

## Point-in-Time Data

This model is used when data changes over time, and "the time period during which a record was relevant" is a fundamental aspect of each record.

This is most easily conceptualized in contrast with the previous, more typical, mode of thinking:

- In the **simple** model, a data point has a value of *x*; someone acts on it, changing the value to *y*.
- In a **point-in-time** model, a data point has a today-value of *x*; tomorrow, it may have a today-value of *y* – but that doesn't mean *x* was *overwritten* – *x* is still the valid yesterday-value.

An example that may be familiar is economic and financial data. A company makes $50 million in 2018. They make $60 million in 2019. $50 million isn't *wrong*; it's still the correct value for 2018. The question "who changed the income field?" is nonsensical.

| AnnualTimeSeries |
|---|
| ID |
| Value |
| Year |

| DailyTimeSeries |
|---|
| ID |
| Value |
| Date |

Updates aren't performed; new records merely are added to a time series. This is a natural choice for data sets that belong in a line graph.

**Note:** If you're using this model for tracking changes, you'll need two date columns – one to track *corrections* (i.e., when $50 million *was* wrong) and another to track *changes* (i.e., when it's still correct for 2018)

## Irregular Validity Periods

The above example assumes a consistent time period for which data points are valid. (example: annual data is valid from Jan 1 to Dec 31, so only the year need be specified)

A similar model can handle irregular data, in which data that's valid for a year is intermingled with data that's valid for a month, or any arbitrary number of days.

Replace the single "year" field with two fields: a beginning and an ending timestamp. Finding the value that was valid at a given time can be accomplished with a "between" operation ([given time] greater than beginValidTimestamp and less than endValidTimesamp).

| TimeSeries |
|---|
| ID |
| Value |
| ValidFromDate |
| ValidToDate |

## Double-point-in-time (Estimates)

This can be further complicated by the introduction of 2-dimensional time. There's one time-component in the definition of the data point, and a second in the validity of the records for that data point.

One example is estimates or predictions, when the prediction is for a specific instance of a recurring event (examples: a company's quarterly earnings, the winner of an annual sporting competition):

1. A prediction is made in (*as-of*-date) January, at the beginning of the season, as to the winner of this year's (*for*-date) championship
2. Another prediction is made in August, the day of the championship match, as to its winner (*for* the same event, *as of* a different date)
3. Another, more intrepid, prediction is also made at that time, as to the winner of *next year's* championship (*as of* the same date as #2, but *for* a different event)

This is handled with by having separate columns for the "for date" and the "as of date". The main source of complexity here is conceptual; there are two different dates on each record, each representing a very different concept, which can be confusing to anyone not well acquainted with the data set.

**Note:** If you're using this model for tracking changes, you'll need *a third* date column.

| Predictions |
|---|
| ID |
| Value |
| ForYear |
| ValidFromDate |
| ValidToDate |

## Carbon Copies / Snapshots

This is the simplest model, conceptually – take a static “snapshot” of the database (or some subset of it) on a regular schedule, and/or when given events occur.

### Backups

The obvious use case is as a backup, an emergency store for recovering from a natural disaster, an attack, or a truly destructive coding mistake. This would be a weekly, daily, or hourly backup taken using your database technology’s built-in functionality.

While these hopefully won’t ever be needed or used in normal operation of your software, it’s the most foolproof fallback option to get to the state of the database as of an arbitrary date in the past, if it becomes necessary.

### Trust

More relevant to the topic of change tracking is to bridge a lack of trust. A snapshot of the relevant information is taken and sent to both parties, along with (optionally) a neutral third party. This can also be thought of as a “download to file” or “save report” functionality. The data is taken out of the database, written to a file (text, PDF, etc.) and then stored to a file system where it’s never changed, and ideally *can’t* be changed or even reached by the software that generated it.

One example would be a legal agreement. Your software would generate a PDF, which would be signed through a third-party e-sign provider, a copy could be stored in your own file storage, another copy emailed to the signer, and the third-party provider would maintain a third copy. The very fact that the e-sign provider is a separate legal entity from yours provides trust, as you can hypothetically tamper with *your own* database after the event has occurred, but not theirs.

## All Three

An example of using all methods at once in a (simplified) medical setting:

1. A patient tracking system. Someone is admitted into a hospital. They have a status of “admitted”. They are later treated and sent home.
    - There’s a need to trace who added this patient to the system, and when.
    - There’s a need to trace who marked this patient as “discharged”, when, and why.
2. A bed tracking system. When beds are filled and vacated, they are marked as such.
    - There’s a need to know how many beds were filled on a given date, and/or graph occupancy over time.
    - There’s a need to know which nurse was on duty at a given time.
    - There also may be a need to develop estimates as to peak occupancy – in order to review and refine the predictive and planning value of those estimates, it’s important to know how our estimate for “occupancy this winter” changed as the season approached – an accurate estimate far before the event in question is more valuable than an accurate one immediately prior.
3. Signatures for consent to sharing information.
    - If a patient signs a form stating that they consent to a certain use of their data, a copy should be furnished to them immediately.
    - Use of a third-party e-sign (or even a physical carbon copy sent to an outside facility) would help mitigate potential disputes, and aid in discovering and diagnosing systemic issues or bad actors.

### Bugs and Corrections

Note that if a user of our bed tracking system made a *typo* – and a record was added for a bed not actually filled – we’ll want to use trace table methodology to track the later correction (if we want to track it at all) rather than append to the time series, which would leave an incorrect record.

For the purposes of generating a historical report, i.e. “what was the data on this date in the past?” it’s important to note that the point-in-time data (intentionally) will not reproduce any errors that may have been present on that date. It therefore may be different from a report generated on the day in question.

If the ability to reproduce old errors is required, that can be achieved either by attaching trace data to the point-in-time data, or via the snapshot method.

## A Nod to Some Other Techniques

### Hot Potato

Each data table has two additional columns: LastEditedTimestamp and LastEditedBy.

This loses the detail of *what* changes were made, and it loses any history prior to the most recent change for each record. It's simple, low storage, and (sometimes) good enough.

### Change Comments, Source File, Other Details

Any of the above techniques may be supplemented by a UI requirement for the user to enter a "descriptive comment" describing the reason for the action. You may also track the source of the change (name of a file uploaded), IP address, or anything at the application's disposal.

### Git

While it's unsuitable as a model for tracking data changes in most database-driven applications, no list of change-tracking methodologies would be complete without a nod to the most popular tool among modern engineering teams to track the changes we make to our own code! Familiarity with Git (and/or other version control software) is highly recommended for anyone seeking to develop new change tracking systems.