Bitemporality in Go

Burke Carter

```
^  ^
|  |
|  |
|  |------>
|
|--------->
```

Copyright © 2025 by Burke Carter
All rights reserved. No part of this book may be reproduced, stored, or transmitted in any form without the prior written permission of the publisher.

First Edition: February 2025

Published by Burke Carter

Library of Congress Control Number: 2025903203

The information provided is for educational purposes only and does not constitute professional advice. The author has endeavored to eliminate as many bugs as possible, but it is unlikely that they have all been squashed.

Foreword

Databases aren't magic, but they sometimes seem to be. From those that hold all of their data in only a single file on your local disk to those that store data on a network of machines that spans the globe, there seems to be a database for every niche. What they have in common is that in fractions of a second, a single record among millions – or even billions, or even more! – of others can be retrieved, modified, or even erased. During the process, other records – and even other queries – can be affected or unaffected depending on the guarantees of the software. I am endlessly fascinated by that software.

If you were to ask a contemporary software engineer to name a few types (not versions or vendors) of databases, you would probably hear a fairly consistent list: relational, NoSQL, key-value, or maybe even columnar. Missing from that list is the enormously clever model called the bitemporal model, which is arguably on a different axis than the others but is nevertheless just as important in its own domain. I would go so far as to argue that the bitemporal model is more distinct from the others than is row store from column store. It takes some time to learn to think bitemporally, but it's time well spent for the queries that can be so elegantly expressed this way.

Working on databases is great fun. I hope that you enjoy working on this one as much as I have.

Table of contents

Chapter 1: Why bitemporal data

Rarely is data static. Our knowledge of the world changes continuously, and we continuously make decisions based on that data. Different data compels us to make different decisions in the present, but that doesn't change the state of the world as we knew it in the past, nor does it change the decisions that we made at that time. In order to preserve the historical record and to understand why what now seems foolhardy was once reasonable, we need to maintain that history. This is the *raison d'être* of bitemporal data.

Let's illustrate what I mean with a simple example. Suppose that our HR application keeps track of employees and the department in which they work:

ID	First name	Last name	Department
1	Alice	Andrews	IT
2	Ben	Baker	Operations
3	Clara	Campbell	R&D

It would be easy enough in a relational model[1] to count how many employees work in each department:

```
SELECT department, COUNT(*)
FROM employess
GROUP BY department;
```

That's a perfectly good point-in-time snapshot. How about transferring an employee to a different department?

```
UPDATE employees
SET department = 'Operations'
WHERE id = 3;
```

That works, but it also causes data loss: we no longer know Clara's previous department! There are at least three cumbersome solutions to this problem:

- Keep both an `old_department` and a `current_department` field. This accurately tracks the first transfer an employee makes, but not the second (or the third, or ...) move. To track an arbitrary number of transfers, we would have to add an arbitrary number of columns.
- Keep `department` as a sorted list of the employee's history such that the first (or last) entry in the list is the employee's current department. This keeps the full history, but it is cumbersome to query and may require a very wide column if employees frequently change departments. For completeness, one would probably also like the dates associated with each move, further expanding the column.
- Keep a "history table" with the employee's ID, their old department, the date of their transfer, and their new department. So long as the main employees table is kept in sync with the history table, this does not lose information. However, some queries, such as "which department had the most employees on date X?" become difficult to answer because the state of the

> world on date X must be carefully reconstructed from the history.

But there's an even worse problem with the third cumbersome solution above: what if the transfer was effective as-of date d_1 but was recorded on a later date d_2? Suddenly, it's not clear what is meant by X in "which department had the most employees on date X?" Do we mean to ask how many employees were effectively in the department on date X, or do we mean to ask how many employees were known to be in the department on date X? Perhaps it's not immediately obvious that this distinction matters. To see the significance, you need only imagine that if Clara's transfer to the Operations department would increase her salary, then she would certainly want the increase to begin on $d_1 < d_2$, perhaps requiring backpay that would be paid on d_2.

The right solution is to model this data bitemporally by assigning four extra fields to each piece of data:

Field	Significance
`tt_from`	When the data was first recorded by the database
`tt_to`	When the data was first invalidated by the database
`vt_from`	When the data became true in the real world
`vt_to`	When the data stopped being true in the real world

In our example, the `tt_from` would be d_2, since that's when Clara's transfer was recorded in the database, and the `vt_from` would be d_1, since that's when her transfer became effective in the real world. Both `tt_to` and `vt_to` are initially set to infinity since when we write a record, we don't know how (or even if) that record will change in the future. Note that while the `vt_from` is set by the user, the `tt_from` is always chosen by the database and is monotonically increasing so that history cannot be overwritten. As we'll see, even though data initially extends to infinity, we cannot read it after the `lsqt`, or "last safe query time," which is the latest time at which all writes have either completed or failed such that the data up to that time will never change. Together, these two rules mirror two principles of life: you cannot change the past, and you cannot see the future.

Graphically, the data exist in the bitemporal plane where the `tt`-axis is the x-axis and the `vt`-axis is the y-axis. On Clara's start date, her bitemporal space would look like this, with the * showing the `(tt, vt)` at which she began her career:

```
^ ^
| |
| |
| |
| |
| |
| | Rectangle 1
| | First name: Clara
| | Last name:  Campbell
| | Department: R&D
| *----------------------------------------------->
|
-------------------------------------------------->
```

To record her transfer at **, rectangle 1 is split so that instead of its `tt` extending to infinity, it now extends only to Y. Rectangle 2 is created with the same data as rectangle 1, and it extends from `tt = Y` to `tt = infinity` with the same `vt_from` as rectangle 1 `vt_to = X`. Finally, rectangle 3 is created with the new data with both `tt_to = infinity` and `vt_to = infinity`.

```
  ^ ^                      ^
  | |                      | Rectangle 3
  | |                      | First name: Clara
  | |                      | Last name:  Campbell
  | |                      | Department: Operations
X | |                      **----------------------->
  | | Rectangle 1          | Rectangle 2
  | | First name: Clara    | First name: Clara
  | | Last name:  Campbell | Last name:  Campbell
  | | Department: R&D      | Department: R&D
  | *----------------------------------------------->
  |
  -------------------------------------------------->
                           Y
```

Now, we can ask questions like, "as of `(∗ ≤ tt < Y, vt == X)`, what was Clara's department?" Restating this question, "at `∗ ≤ tt < Y`, what did we think was Clara's department starting at `vt == X`?" In the image above, we can see that these temporal coordinates fall inside the rectangle labeled 1, so the answer is the R&D department. If we were to modify the question slightly and ask, "at `tt == Y`, what did we think was Clara's department starting at `∗ ≤ vt < X`?" These temporal coordinates fall inside rectangle 2, so the answer is again the R&D department. Finally, we can ask, "at `tt == Y`, what did we think was Clara's department starting at `vt == X`?" That falls inside rectangle 3, so the answer is the Operations department, the department to which Clara transferred.

Why this book

There are plenty of resources available that will tell you what bitemporality is, but until now, there has been no comprehensive guide to teach you how to build a database and client that inherently support bitemporality. By the end of this book, not only will you have a working knowledge of the core concepts, but also you will know how to program in that model, and you will have some ideas for additional features that you may want or need.

This book is structured as both a guide and an implementation. If you follow along, you'll implement a server and a client for a bitemporal object database.

Throughout the book, you'll also read about alternatives that would provide different properties and trade-offs. It's up to you to decide how to extend the provided source code to match your specific use case. For brevity, most of the thousands of lines of tests are omitted from the book and exist only in the source code itself. Most of the non-test code appears in the book in short snippets with accompanying explanations. At times, the code is broken into shorter functions than usual in order to avoid functions longer than a single page. Likewise, the physical pages of a book are more narrow than a large monitor with an IDE, so some code that would appear on a single line in an IDE is broken into multiple lines.

Dependencies

The main dependency is Go, the programming language that we will use throughout this book. Its combination of clarity, performance, and ease-of-use make it the ideal language for this endeavor. To build a faster version, you could use Rust or C++, but as a language for teaching, Go wins with simplicity. Along the way, we will also use some open source libraries such as Protocol Buffers.

We'll use Go version 1.24.0, but we won't use any advanced features specifically from that release. A slightly older version should also work.

```
$ go version
go version go1.24.0 linux/amd64
```

The most surprising dependency might be SQLite. This book is not about how to implement a storage engine, but rather how to implement a bitemporal database on top of your existing storage—after all, most projects already have a traditional or general-purpose database, and the piece that they're missing is bitemporality. Because of the architecture that we'll use, it would be possible after reading this book to swap the storage with, say, PostgreSQL, or any other storage of your choice.

We'll use SQLite version 3.49.0. As with our choice of Go version, we won't use any advanced features specifically from that release, so a slightly older version should also work.

```
$ sqlite3 --version
3.49.0 2025-02-06 11:55:18 \
  4a7dd425dc2a0e5082a9049c9b4a9d4f199a71583d014c24b4cfe276c5a77cde (64-bit)
```

For hermetic builds, we'll use Bazel, and to make generating BUILD files easier, we'll use Gazelle.

We'll use a bazel version manager called bazelisk with version 1.18.0, bazel version 7.3.2, and gazelle version 0.35.0.

```
$ bazelisk version
Bazelisk version: v1.18.0
```

```
$ bazelisk --version
bazel 7.3.2
```

```
// WORKSPACE

http_archive(
    name = "bazel_gazelle",
    integrity = \
      "sha256-MpOL2hbmcABjA1R5Bj2dJMYO2o15/Uc5Vj9Q0zHLMgk=",
    urls = [
        "https://mirror.bazel.build/github.com/" + \
        "bazelbuild/bazel-gazelle/releases/download/" + \
        "v0.35.0/bazel-gazelle-v0.35.0.tar.gz",
        "https://github.com/bazelbuild/bazel-gazelle/" + \
        "releases/download/v0.35.0/" + \
        "bazel-gazelle-v0.35.0.tar.gz",
    ],
)
```

1. The examples in this book use vanilla SQL, but any similar query language should suffice. ↩