VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE II

Master Thesis

# Implementation Aspects of Bitemporal Databases

Done by:
Dmitrij Biriukov signature

Supervisor:
dr. Linas Bukauskas

Vilnius
2018

# Contents

## Abstract

This thesis explores concepts of bitemporal data storage and querying techniques - from data structures to storage, extraction and performance. With stable growth of unstructured data, relational data becomes too complicated to manage and store. Therefore, this paper explores various NoSQL storing solutions that might help find a way to efficiently store and extract bitemporal data. As a proof-of-concept, an application capable of performing bitemporal operations performance tests was constructed. It's quite important to know how bitemporal concepts apply to non-relational database, as well as to traditional relation databases. All bitemporal queries are measured against various metrics - time, memory, IO and CPU usage.

# Santrauka

**Bitemporalinių duomenų bazių įgyvendinimo metodikos**

Šio darbo pagrindinis tikslas – išaiškinti įvarius bitemporalinių duomenų bazių įgyvendino aspektus. Darbe nagrinėjami duomenų struktūrų, jų saugojimo ir ištraukimo aspektai. Pagrindinis dėmesys skiriamas SQL/NoSQL duomenų bazių bitemporaliams aspektams. Sukurtas įrankis, leidžia lengvai testuoti sudėtingas duomenų manipuliavimo bei atrinkimo užklausas. Atliekama palyginamoji analizė tarp reliacinių ir nereliacinių duomenų bazių valdymo sistemų analizuojant jų našumą ir sudėtingumą. Bitemporalinės užklausos lyginamos pagal laiką, atminties, disko bei procesoriaus naudojamus resursus.

# Introduction

Previous research on this topic has partially shown bitemporal data storage techniques and approaches using relational databases. However, this thesis will focus more on modern data management systems - NoSQL. With continuous growth of semi-structured and unstructured data, managing and maintaining relational data have become much harder. Thus, it is more beneficial to explore other possible data storage mechanisms, that might help to remedy relational problems.

Utmost of the data in today's systems are temporal. Tracking validity of an entity in a single point in time could be described as a trivial task. However, bitemporality adds another dimension to the game. Bitemporal model supports two different timelines - valid time and transaction time. First one represents object's factual time while transaction time underlines when object's became known to database. In order words 'what you know' or 'when you knew it'. Bitemporal solutions picked up traction in financial institutions, trading, legal practices or even medicine. When you have an option to wind back clocks at any time, some of common complexities become trivial. For example, in medicine, bitemporality can present more precise patient history. You can then correlate illnesses with preexisting conditions. Furthermore law practices can significantly benefit from this. Knowing which laws where active during given time interval, it can be applied to legal artifacts. Nevertheless, financial institutions can use bitemporal system to comply with financial regulations - for example, having transparent trading - uniformed, consistent data and rewinding trading metrics.

First problem that can be pointed out is scalability issue. Conventional database management systems are designed to be vertically scaled. Such architecture creates both technological and business barriers, that will not be easy to solve. The issue of such vertical scaling is explained in the detail later on. Second problem is adapting to modern data structures, which in itself could be a daunting task. Variety of different services and features are leading to diverse data structures, which not all relational databases are equipped to handle. NoSQL can offer semi-structured or even unstructured data handling. Furthermore, there are numerous types of NoSQL database, which are analyzed and compared in the second chapter of this thesis. In addition to this, there is one more issue, that is equally important - speed. Data saving or retrieving speed is significant to any database management system. This thesis does substantially cover relational and non-relational speed comparison on data insertion, change, deletion and extraction.

The end-goal of this thesis is to construct a model, which can help to create a tool that can manipulate and extract raw bitemporal data in relational and non-relational database solutions. Created proof-of-concept will help to shape assorted bitemporal performance tests and measure them against variety of metrics.

# 1 Related work

As previous scientific research states bitemporal modeling is a design technique, used for saving and extracting historical data that evolve over time. This model is used for storing two different sets of time instances - valid and transaction times. By using two dimensional model we can actually "go back" to the past and see what was the state of an object, more precisely, of its properties, at that point in time. Figure 1 shows us a visualization of bitemporal data. Rectangles represent objects and their validity in time.

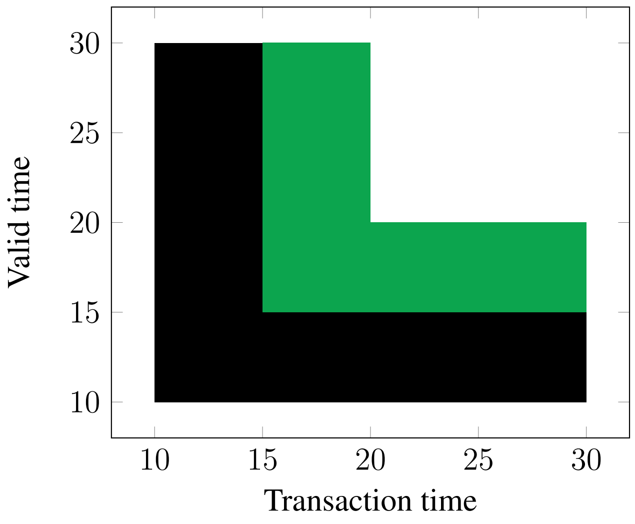


Figure 1. Transaction and valid times example

## 1.1 Time dimensions

Bitemporal data stores two dimensional time intervals, meaning we have two attributes defining validity of tuple [28]. First is VT[1] - valid time is actual time of object factual validity [29]. Other time attribute is TT[2] - transaction time which is responsible for time tuple is actually valid [29]. Having these two attributes we can tell the state of an object in the past. For example, we have a warehouse database which supports bitemporal model. Also, we have inventory records that are bitemporal as well. Having TT and VT we can tell what was the inventory and some point is the past.

### 1.1.1 Valid time

Valid time presents the factual time, during which the object is valid. It can be set in any time interval [26]. For example, Jane worked at Company A from 2013 to 2016. Judging by our current year 2018, we can state that Jane no longer works at Company A, but is working somewhere else.

### 1.1.2 Transaction time

Transaction time marks the fact, during which the object is processed in database [25]. Usually TT is interpreted as DBMS operation duration interval. Besides, it is unlike VT, TT start time must

[1]Valid time
[2]Transaction time

not be set in future [20]. To this fact, one of the most obvious consistency constrains of both VT and TT is that period beginning of the interval should not be larger than ending interval.

## 1.2 Bitemporal entity

Entity can be called bitemporal if it has support for two dimensions of time [26]. First dimension must be VT, whereas second is TT. Following table 1 shows basic example for bitemporal data in comparison to temporal seen in table 2.

| **Id** | **Name** | **VT_START** | **VT_END** | **TT_START** | **TT_END** |
|---|---|---|---|---|---|
| 1 | *Peter* | 2017-01-01 | 2018-02-16 | 2017-01-01 | 2018-02-16 |
| 2 | *John* | 2018-02-17 | $\infty$ | 2018-02-16 | $\infty$ |

Table 1. Sample bitemporal table

Temporal entity loses transactional dimension. It has support for only one time dimension - VT. In table 2 we can not tell when record was changed, therefore, we can not "restore" object's state changes.

| **Id** | **Name** | **VT_START** | **VT_END** |
|---|---|---|---|
| 1 | *Peter* | 2017-01-01 | 2018-02-16 |
| 2 | *John* | 2017-02-17 | $\infty$ |

Table 2. Temporal entity that represents employee's validity

Moreover, snapshot entity (see table 3) is a traditional structure, that has none of time dimensions [25]. It is most commonly shown as conventional relation, where validity of tuple can be determined by its physical existence in database. Conventional relation has no way of determining whether relations instance is valid or it was physically created.

| **Id** | **Name** |
|---|---|
| 1 | *Peter* |
| 2 | *John* |

Table 3. Snapshot entity that stores employees

## 1.3 Bitemporal data types

Various DBMS have vast majority of time-related data types. All of them have different purpose, but only few can be applicable to temporal model [28]. Since bitemporal database concept requires at least four more additional time-based columns, data types should be selected carefully.

First of all, time-related types can have different allocation sizes and if incorrect type is selected, system can face performance or compatibility issues. In this table (see table 4) you can see comparison of different date and time types [12].

| Name | Size | Format | Time resolution |
|---|---|---|---|
| *timestamp* | 8 bytes | yyyy-mm-dd hh:mm:ss | 1 microsecond |
| *timestampz* | 8 bytes | yyyy-mm-dd hh:mm:ss Z | 1 microsecond |
| *date* | 4 bytes | yyyy-mm-dd | 1 day |
| *time* | 8 bytes | hh:mm:ss | 1 microsecond |
| *timez* | 12 bytes | hh:mm:ss Z | 1 microsecond |
| *interval* | 16 bytes | dd day hh hours mm min ss sec | 1 microsecond |

Table 4. PostgreSQL date and time data types

In addition to this, time-related data types have precision parameter [12]. It is primarily responsible for increasing time resolution. By default PostgreSQL stores timestamp instance as a eight-byte integers, but changing precision parameter to a higher value, will result in floating-point number storage, which maybe be useless. For example, timestamp column with highest precision can be defined like this (see listing 1):

```
vt_from timestamp(6) without time zone
```

Listing 1: Highest precision timestamp

## 1.4 Data definition principles

To create a bitemporal entity, table with four additional columns should be created. First pair is for valid time dimension, other - transaction time dimension. Both valid and transaction times are intervals, which means that they must have a beginning and end values. Also, ending columns (VT_END, TT_END) must have NULL requirement constraints - end date can be empty. Despite that, start date (VT_START, VT_END) can not be empty, it must have **NOT NULL** constraint.

Furthermore, columns must be date-typed. Every DBMS has vast variety of date types. For example, PostgreSQL has fairly large selection of time-based types (see table 4 in previous subsection). Regardless of DBMS, basic bitemporal data definition concepts still remain the same. Either new columns are added via **ALTER** statement or they appear in DDL statement.

Let's examine table 5, in which table's structure is defined. It has four timestamp columns, which represent bitemporal dimensions. Structurally, entity only differs in having additional columns. Nonetheless, defining hollow columns is not enough. Bitemporal data flow functionality, defined in next subsection, should also be implemented.

| Column | Type | Properties |
|---|---|---|
| id | **bigserial** | PK |
| name | **varchar(255)** | NOT NULL |
| surname | **varchar(255)** | NOT NULL |
| salary | **numeric** | DEFAULT 0 |
| vt_start | **timestamp** | NOT NULL |
| vt_end | **timestamp** | NULL |
| tt_start | **timestamp** | NOT NULL |
| tt_end | **timestamp** | NULL |

Table 5. Example of bitemporal entity

## 1.5 Data flow

In order to have a better understanding of bitemporal data flow we have to imagine an information flow, where nothing is deleted and where all prominent data management operations are outsourced to either inserting new tuple or updating existing one. Let's examine a specific examples (see tables 6, 7 and 8) and give each operation a brief description.

In table 6 we perform **DELETE** operation. Row is identified by PK[3] "Id", which value is 1. This record should be eliminated. This will result in **UPDATE** operation which will change bitemporal attributes. When tuple is deleted, we need to void the valid time, by assigning a discrete value to it. If we need to do immediately, then current timestamp is assigned. Furthermore, it can be a different value set in the future. For example, employee handed his resignation letter (see table 6), which was processed at **TT_END** = 2018-02-16 and finally voided on **VT_END** = 2018-02-17.

| Id | Name | VT_START | VT_END | TT_START | TT_END |
|---|---|---|---|---|---|
| 1 | *Peter* | 2015-01-01 | **2018-02-17** | 2015-01-01 | **2018-02-16** |

Table 6. An example of bitemporal deletion

Next is **UPDATE** operation (see table 7). This operation is rather complex, because any change to tuple's attributes will result in the spawning of a new record. For example, if person wishes to change his name, a new row should be created (with new name). This is somewhat similar to **DELETE** operation. However, **UPDATE** creates, a new row with values and bitemporal time intervals (shown in table 7). In the example, Peter changed his name to John on *2018-02-16*. New name became valid at *2018-02-16* and is yet to be voided.

| Id | Name | VT_START | VT_END | TT_START | TT_END |
|---|---|---|---|---|---|
| 1 | *Peter* | 2017-01-01 | **2018-02-16** | 2017-01-01 | **2018-02-16** |
| 2 | *John* | 2018-02-17 | $\infty$ | **2018-02-16** | $\infty$ |

Table 7. An example of bitemporal modification

In **INSERT** operation (see table 8) we need to be account for bitemporal intervals. They should be changed according to bitemporal rules. When tuple is inserted, valid time is assigned. Usually it is the time, when the object is created. For example, we want to record calendar events with specific times and meeting places. For each instance, we need to create a row, which will have different valid times (shown in table 8). This would make meetings valid at the different points time. Since meetings were inserted in the database the same day and they still are valid, **TT_START** is equal to 2018-03-01 and **TT_END** is NULL.

| Id | Name | Place | VT_START | VT_END | TT_START | TT_END |
|---|---|---|---|---|---|---|
| 1 | *Peter* | Office | 2018-05-03 | $\infty$ | 2018-03-01 | $\infty$ |
| 2 | *John* | Home | 2018-02-16 | $\infty$ | 2018-03-01 | $\infty$ |
| 3 | *Jack* | University | 2018-02-10 | $\infty$ | 2018-03-01 | $\infty$ |
| 4 | *Joseph* | Home | 2018-04-11 | $\infty$ | 2018-03-01 | $\infty$ |
| 5 | *Aaron* | Online | 2018-05-29 | $\infty$ | 2018-03-01 | $\infty$ |

Table 8. An example of bitemporal insertion

[3]PK - Primary key

# 2 Relational databases

In modern world are plenty of non-temporal databases. Vast majority of them support ISO/IEC 9075-1:2016 [1]. Standard consists of three basic principles.

- DDL - Database definition language
- DML - Data manipulation language
- QL - Query language

All three of them play significant role in bitemporal relational database concepts.

## 2.1 Database engine

As an example let's take a look at database management system engine and outline basic components. Since we use PostgreSQL [12] DBMS throughout this thesis, we will focus on PostgreSQL internal engine below. It's also worth mentioning that components presented below can apply to different relational database engine. In figure 2 we can observe basic components of PostgreSQL core engine.

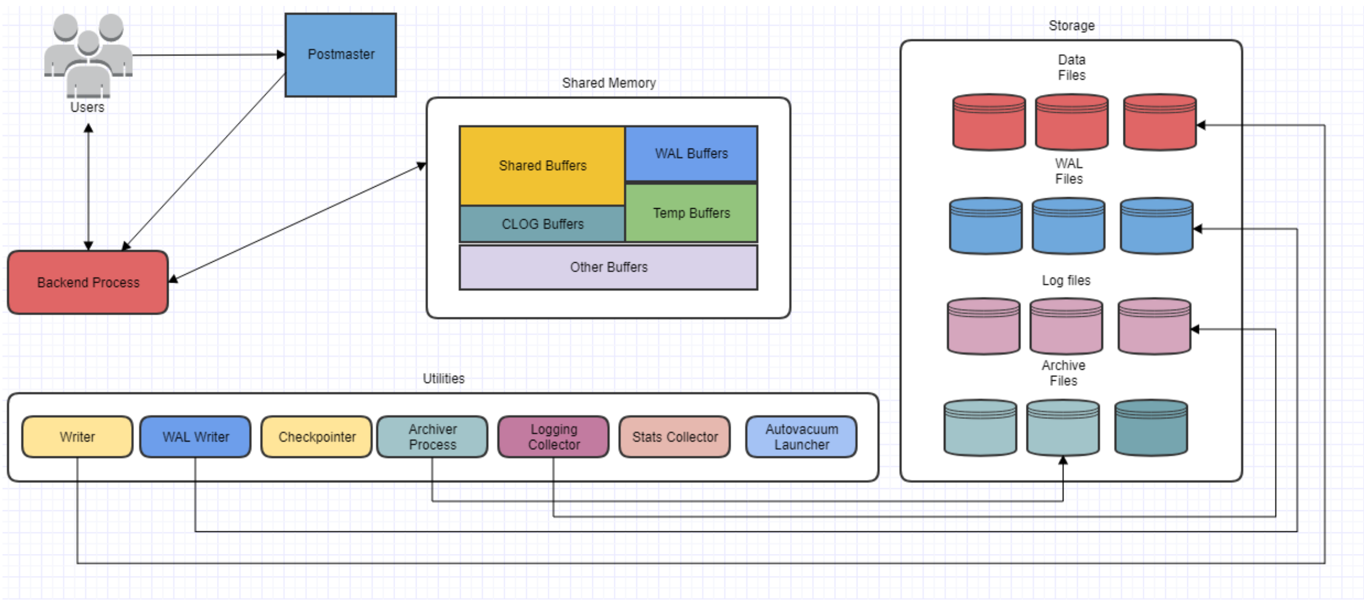


Figure 2. Example of PostgreSQL database engine

Below are crucial components used in nowadays PostgreSQL DBMS.

- Shared Memory is memory reserved either for caching, WAL[4] cache or temporary transaction logs.
  - Shared Buffer is used for disk IO minimization
  - WAL is temporary buffer between memory and actual data files. WAL plays crucial role in data recovery.
- Data Files are actual data structures - catalogues, tables, indexes, procedures, triggers, etc.

[4]In computer science, write-ahead logging (WAL) is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems.

# 3 NoSQL databases

In this section various NoSQL database types will be discussed. In addition to this, a brief definition of NoSQL concept will be presented. Lastly, a brief comparison of database types will be shown.

## 3.1 What is NoSQL?

A NoSQL database environment is non-relational and largely distributed database system that enables rapid, ad-hoc organization and analysis of extremely high-volume, disparate data types [7]. NoSQL databases are sometimes referred to as cloud databases, non-relational databases, Big Data databases and a myriad of other terms and were developed in response to the sheer volume of data being generated, stored and analyzed by modern users (user-generated data) and their applications (machine-generated data) [7]. In general, NoSQL databases have become the first alternative to relational databases, with scalability, availability, and fault tolerance being key deciding factors [18]. They go well beyond the more widely understood legacy, relational databases in satisfying the needs of today's modern business applications [7].

## 3.2 Key-value store

Key-value databases generally uses stores primitive structured associative data. The relationship between key and the value is always one-to-one. However, there could be many keys in the collection with same name.

Every single value in the associative array embody a varied length string. It does not depend on data modeling nor structure constraints. Some times values could be URI[5], BLOBs[6], geographical coordinates or text value.

Dominant key-value databases are:

- Redis [24]
- Memcached [19]
- Oracle NoSQL Database [5]

Following image (shown in figure 3) will help to illustrate how keys and values are stored. Keys are displayed on the left, whereas value are shown on the right. Arrows represent one-to-one connection between objects. Notice that key "User entity" can repeat.

[5]Uniform Resource Identifier

[6]Binary large object

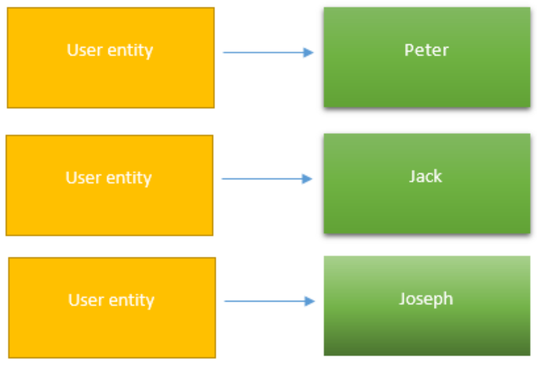


Figure 3. Example of key-value store database

## 3.3 Document-oriented database systems

Document-oriented database systems also known as document stores are well known schema free storage. The more complex data gets, the harder it will be to store. However, document-oriented system solves this issue by allowing user to store relatively complex data in schema-free manner. Furthermore, it supports multidimensional columns - arrays, multi-typed columns and nesting.

In addition to this, document-oriented systems inherited concepts from key-value stores. However, main advantage of document stores are querying. It's apparent that key value systems are limited to querying by the key. Document-oriented system can do more than that, for example: perform queries on loosely oriented structures, indexing and ad hoc queries. Example shown in figure 4 define simple relationship between 3 document records, in there we can observe schema-less data, that can be handled by document-oriented database.

Leading document-oriented databases are:

- CouchDB [3]
- MongoDB [17]
- Elasticsearch [4]

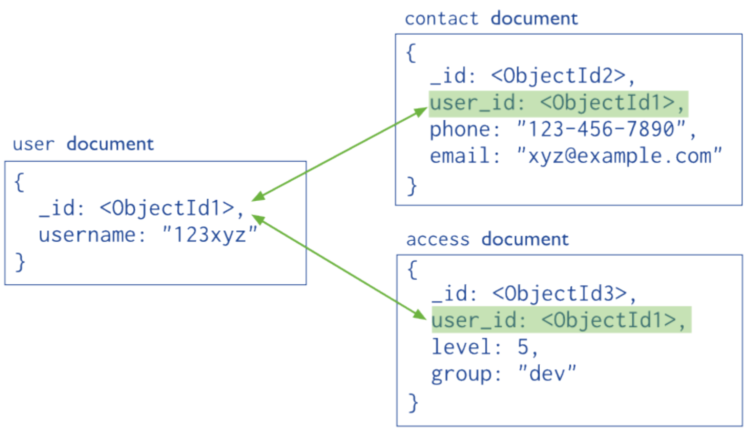


Figure 4. Example of document database

## 3.4 Graph-oriented databases

Graph-oriented databases adopt graph theory and semantic querying in order to map, store and process relationships between data. It can be viewed as connection of nodes and edges. Main advantage of such system are relation calculation efficiency.

A graph database does not compute data relationship at query time, but rather store is a main peace of information, to be read instantaneously. It allows you to pass through large amounts of relationships in efficient amount of time. Following example (look at figure 5) illustrates possible relationship between data.

Popular graph-oriented databases are:

- OrientDB [21]
- InfiniteGraph [23]
- AllegroGraph [15]

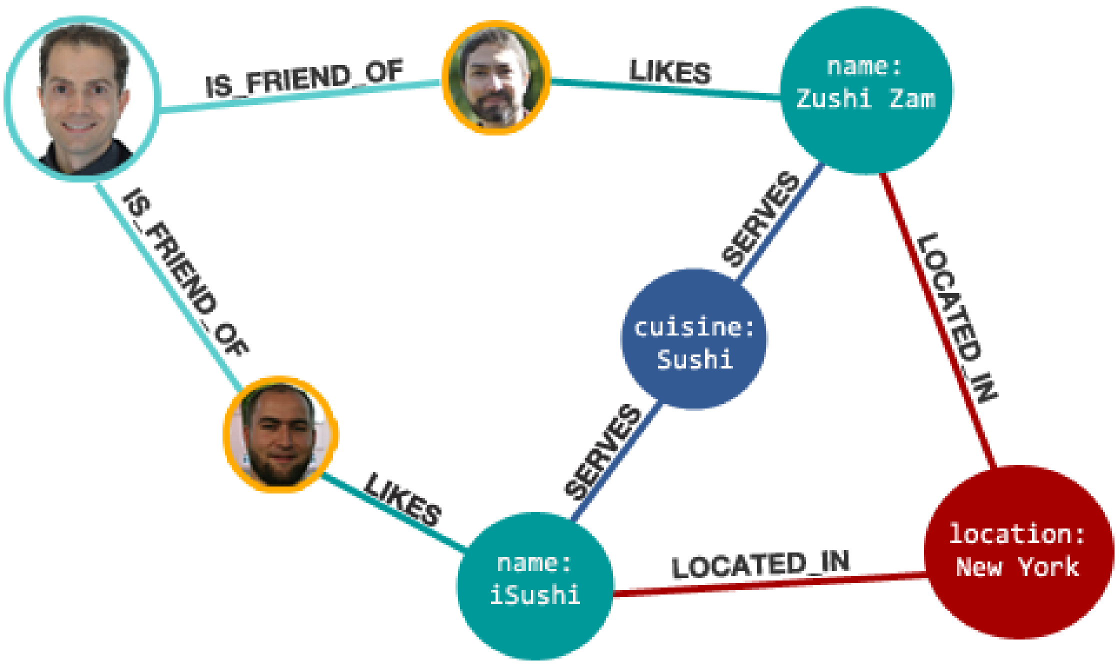


Figure 5. Example of graph database

## 3.5 Object-oriented database management systems

Object-oriented database management systems or object databases are system that store values as objects. Object can be modeled according to conception of object-oriented paradigm. However, such databases are are still in development. There has been complains of performance problems, lack of decent indexing system.

On the other side, the object persistence is easy to program, and complex data objects are surprisingly natural. Object-oriented data sample can be found in figure 6. Where a simple one-to-many relationship is shown.

Popular object-oriented databases are:

- ObjectDB [3]
- GemStone/S [17]

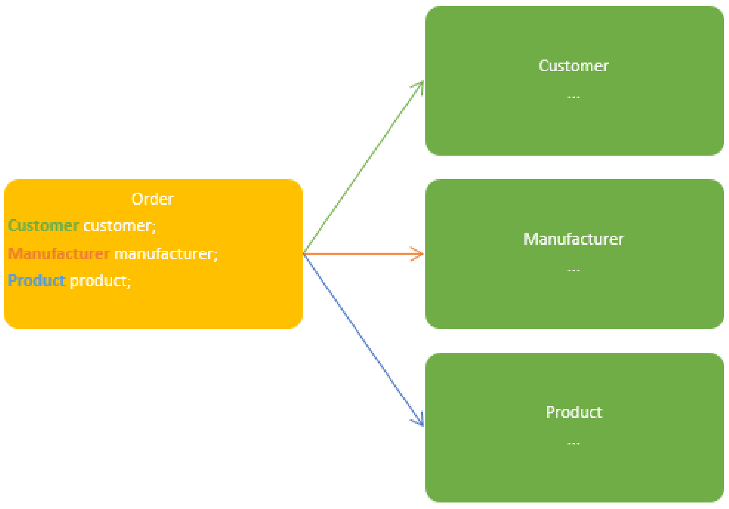


Figure 6. Example of object database

## 3.6 Column-oriented databases

Column-oriented serialize data into columns. Such systems are designed for parallelism and speed. A column of a distributed data store is a NoSQL object of the lowest level in a keyspace. For example, in figure 7 illustrates Apache Cassandra basic architecture [8]. We can see three main level of hierarchy. First is keyspace which references the main project name. Second is column family which is equivalent to entity object. Last level is column where the main data is actually store. It uses three attributes to define a column - name, value and timestamp.

Major column-oriented databases are:

- Apache Cassandra [8]
- Google Bigtable [16]

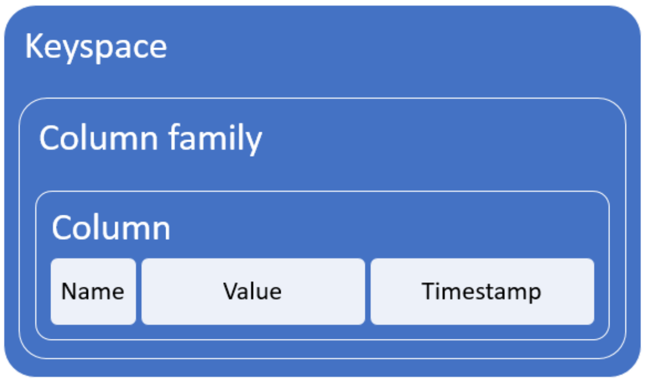


Figure 7. Example of column database

## 3.7 NoSQL database type comparison

All listed NoSQL types are known and used in specific conditions. However, a primitive analysis covering performance, scalability, flexibility and complexity can be performed. In general NoSQL databases focus on speed - almost all types have high performance. Similar can be said about scalability and flexibility of the system. The only category where NoSQL can be weak is complexity of data. Moreover, in most systems data is usually semi-structured and provides less complicated definition. In table 9, we can find summarized result of comparison.

| **Type** | **Performance** | **Scalability** | **Flexibility** | **Complexity** |
|---|---|---|---|---|
| Key-value store | High | High | High | None |
| Column Store | High | High | Moderate | Low |
| Document Store | High | High | High | Low |
| Graph Database | Variable | Variable | High | High |
| Object-oriented | Moderate | High | High | High |

Table 9. Comparison of NoSQL database types table

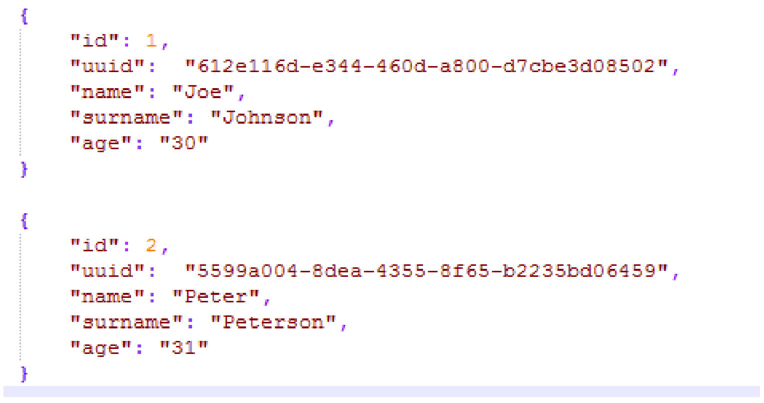
# 4 NoSQL properties

This chapter will cover basics of NoSQL database concepts such as schema-less modeling, CAP theorem, big data support, horizontal and vertical scaling.

## 4.1 Schema-less model

Schema-less design allows to define flexible structures of data [9]. Storing data without intermediate knowledge of data types or relationships [9]. This allows you to archive ease of maintenance. However, having schema-less designed system can lead to ambiguity, which can cause lack of constraints and poor data integrity.

JSON[7] example shown in (see figure 8) illustrates schema-full design because the column constraints are enforced. For example, field "name" can not be empty.

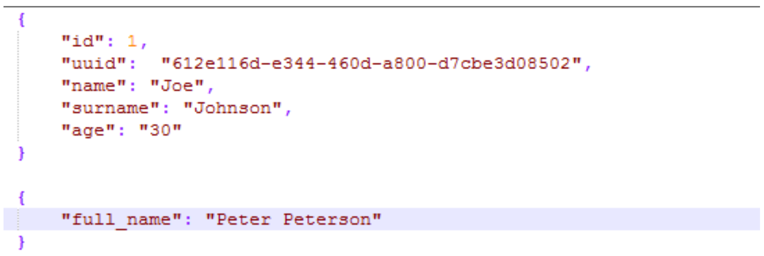


```
{
    "id": 1,
    "uuid":  "612e116d-e344-460d-a800-d7cbe3d08502",
    "name": "Joe",
    "surname": "Johnson",
    "age": "30"
}

{
    "id": 2,
    "uuid":  "5599a004-8dea-4355-8f65-b2235bd06459",
    "name": "Peter",
    "surname": "Peterson",
    "age": "31"
}
```

Figure 8. Example of schema-less

Following example (see figure 9) shows a schema less definition which uses two separate collections to store data. This creates certain ambiguity between data structures which can confuse developer.



```
{
    "id": 1,
    "uuid":  "612e116d-e344-460d-a800-d7cbe3d08502",
    "name": "Joe",
    "surname": "Johnson",
    "age": "30"
}

{
    "full_name": "Peter Peterson"
}
```

Figure 9. Example of schema-less model

## 4.2 Big Data support

Big Data is term outlining complexity of managing enormous data sets. Managing such data with traditional tools (e.g., relational databases) is inefficient. However, NoSQL provides a way to potentially handle big data.

Usually Big Data can be defined using these properties [14]:

[7]JavaScript Object Notation

- High data velocity – lots of data coming in very quickly, possibly from different locations.
- Data diversity – various data structures: structured, semi-structured or unstructured.
- Data size – sheer amount of data that leads to terabytes or petabytes in size.
- Data complexity – geographical data storage and management is not trivial.

## 4.3 Database scaling concepts

In this chapter will cover basics of database scaling and comparison between them. These concepts are scale-up and scale-out. Since two of them correspond to different configuration, they both are applicable for certain case uses. In this thesis, few uses cases are described in detail.

### 4.3.1 Scale-out

Scale out also known as horizontal scaling, has capability to scale adding more nodes the architecture, making parallel computing faster [13]. Prices of mid-range machine has dropped and scaling out became cheaper method of making system bigger and faster [13]. This technique frequently employs use of low-cost computer components which scale system both in capacity and performance.

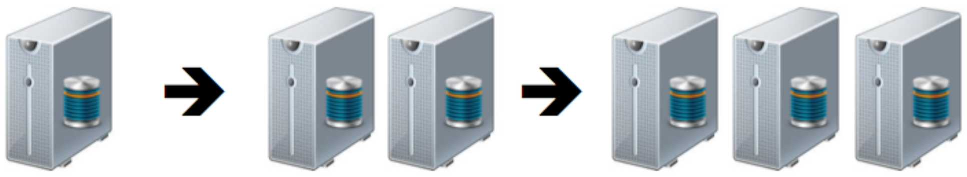


Figure 10. Example of scale-out

When talking about database scaling, relational horizontal scaling is not applicable. However, NoSQL is the perfect candidate for scale-out. With significant growth of data, it's impossible to construct such machine that would be able to withstand a huge amount of data input and output.

### 4.3.2 Scale-up

To scale vertically (or scale up) means addition of resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer. Such vertical scaling of existing systems also enables them to use virtualization technology more effectively, as it provides more resources for the hosted set of operating system and application modules to share [13]. Taking advantage of such resources can also be called "scaling up". Application scalability refers to the improved performance of running applications on a scaled-up version of the system [13].

Figure 11. Example of scale-up

## 4.4 CAP Theorem

CAP stands for Consistency, Availability and Partition tolerance [10]. As it was mentioned before, nowadays we have sheer amount of rapidly growing data that it a challenge to store, compute, manage and maintain. In addition to this, there is countless amount of software available. CAP concept can help to choose particular system, according to three main attributes. Figure 12 illustrates three main components of CAP theorem. However, CAP points out, that modern NoSQL systems, do not satisfy all three conditions of theorem [10].

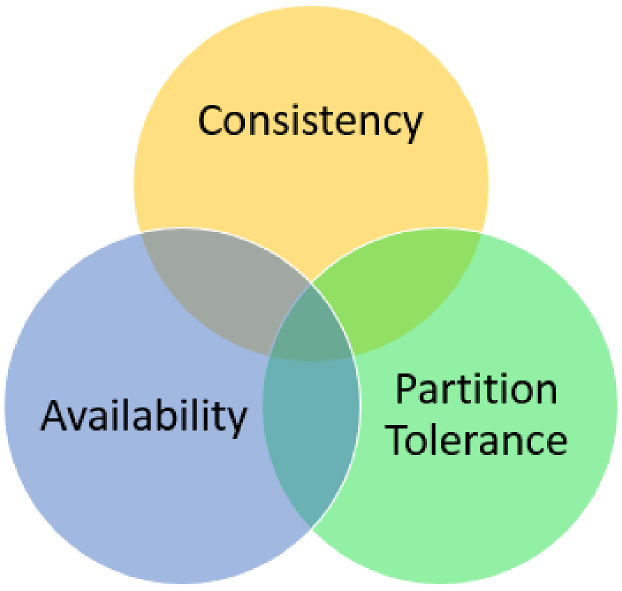


Figure 12. CAP theorem

### 4.4.1 Consistency

Consistency states that all nodes of the cluster must have the same state of data at the given time [10]. In other words, data should be synchronized amongst nodes. Any read operation must return exact data. In addition this, a system must have consistent state during a transaction, otherwise it rollback is performed. Same property can be found in ACID[8] characteristics. Therefore, this

[8]ACID - (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.

property can be applied not only to NoSQL, but to relational database systems as well.

### 4.4.2 Availability

This property requires operations to be stateful. This means that operations must return pass/-fail marks [10]. To archive availability means given system has to be operational 100% of the time.

### 4.4.3 Partition Tolerance

System continues to work neglecting partial failure [10]. Distributed systems have multiple instanced of database and do not have SPOF[9]. If one node fails, the entire distributed system should converge. A system that is partition-tolerant can sustain any amount of failure that paralyze the entire network [10].

[9]SPOF - Single point of failure

# 5 Goal and objectives

The main goal of this thesis is to create comprehensive NoSQL/SQL bitemporal database analysis from implementation and performance perspectives. It is quite important to know how database management system handles certain types of data. In our case, data saved is never deleted and additional values are appended. This that certainly can complicate how we view and analyze data. Nevertheless, data selection is as important as storage. Therefore, this thesis explores one of the way we can see bitemporal data. Finally, as for proof-of-concept, various relational and non-relational bitemporal performance test cases will be performed, analyzed and compared. For achieving goal these objective criteria must be met:

- Analyze bitemporal manipulation techniques (insertion, change, deletion)
- Inspect bitemporal data extraction techniques (coalescing, slicing)
- Implement database association algorithm, that will allow to manipulate variety of different relational and non-relational systems;
- Outline assorted bitemporal data extraction operations and implement one of them
- Perform thorough comparison among NoSQL systems and find one(s) suitable for bitemporality.
- Investigate how to increase performance of bitemporal data manipulation and extraction

# 6 Model

In this chapter, the data model, bitemporal interactions and algorithms are explained. Also, theory of storing bitemporal data in NoSQL system is outlined. The essential difference comes down to storing the two time dimensions as fields for existing document structures.

## 6.1 Extending database

By extending non-bitemporal database it is possible to achieve bitemporality. This approach seems to have one of the most popular onces. Due to fact that most the database functionality can be reused, you only need ensure data integrity and provide proper query language.

However, this approach deemed to have multiple disadvantages. First of all, by using non-bitemporal database, this solution inherently becomes non-bitemporal in the sense that non-bitemporal restrictions becomes bitemporal ones. Secondly, solution become harder to maintain and support. Lastly, making sure that performance state is at its best becomes a challenge.

## 6.2 Bitemporal operations

Extending relational database schema to hold bitemporal may seem like the way to go, however it is not enough. Bitemporal model states [29] that it should have proper data integrity contains as well as proper query domain. Following section will lay the ground plan for bitemporal operations.

## 6.3 Comparison of time

James F. Allen back in 1983 introduced thirteen basic comparison predicates for temporal intervals [2]. These base comparison serve as backbone to bitemporal operations.

Let's say we have time interval $T$, where $T_1$ and $T_2$ are interval's start and end respectively. In the table 10 we can observe main comparison predicates used in bitemporal operations.

| **Predicate** | **Operation** |
|---|---|
| $T_1$ equals $T_2$ | $start(T_1) = start(T_2) \wedge end(T_1) = end(T_2)$ |
| $T_1$ before $T_2$ | $end(T_1) < start(T_2)$ |
| $T_1$ after $T_2$ | $end(T_2) < start(T_1)$ |
| $T_1$ during $T_2$ | $(start(T_1) > start(T_2) \wedge end(T_1) \leqslant end(T_2)) \vee (start(T_1) \geq start(T_2) \wedge end(T_1) < end(T_2))$ |
| $T_1$ contains $T_2$ | $(start(T_2) > start(T_1) \wedge end(T_2) \leqslant end(T_1)) \vee (start(T_2) \geq start(T_1) \wedge end(T_2) < end(T_1))$ |
| $T_1$ overlaps $T_2$ | $start(T_1) < start(T_2) \wedge end(T_1) > end(T_2)) \wedge end(T_1) < end(T_2)$ |
| $T_1$ overlapped by $T_2$ | $start(T_2) < start(T_1) \wedge end(T_2) \leqslant end(T_1)) \wedge end(T_2) < end(T_1)$ |
| $T_1$ meets $T_2$ | $end(T_1) = start(T_2)$ |
| $T_1$ met by $T_2$ | $end(T_2) = start(T_1)$ |
| $T_1$ starts $T_2$ | $start(T_1) = start(T_2) \wedge end(T_1) < end(T_2)$ |
| $T_1$ started by $T_2$ | $start(T_2) = start(T_1) \wedge end(T_2) < end(T_1)$ |
| $T_1$ finishes $T_2$ | $start(T_1) > start(T_2) \wedge end(T_1) = end(T_2)$ |
| $T_1$ finished by $T_2$ | $start(T_2) > start(T_1) \wedge end(T_2) = end(T_1)$ |

Table 10. Time comparison predicates

Some of these predicates will be used in the implementation phase (in chapter 7).

## 6.4 Coalescing

The idea of coalescing was introduced by [27]. To demonstrate coalescing in action let's review table 11, containing library book log. This entity has reader's name, book's name and temporal attributes. For the sake of simplicity temporal intervals are expressed in years.

| **Name** | **Book** | **From** | **To** |
|---|---|---|---|
| Jack | Moby-Dick | 2015 | 2018 |
| Jack | Moby-Dick | 2018 | $\infty$ |
| Jane | Don Quixote | 2010 | $\infty$ |
| Tom | Ulysses | 2017 | $\infty$ |
| Andy | To Kill a Mockingbird | 2010 | 2017 |
| Alain | The Great Gatsby | 2017 | $\infty$ |
| Martin | Pride and Prejudice | 2010 | 2012 |

Table 11. Data before coalescing operation is performed

Result of coalescing operation is shown in table 12.

| **Name** | **Book** | **From** | **To** |
|---|---|---|---|
| Jack | Moby-Dick | 2015 | $\infty$ |
| Jane | Don Quixote | 2010 | $\infty$ |
| Tom | Ulysses | 2017 | $\infty$ |
| Andy | To Kill a Mockingbird | 2010 | 2017 |
| Alain | The Great Gatsby | 2017 | $\infty$ |
| Martin | Pride and Prejudice | 2010 | 2012 |

Table 12. Data after coalescing operation is performed

## 6.5 Vertical slice

Time slicing can be visualized as a vertical line splitting "Transaction time" axis in half. In figure 13 we can observe red vertical line splitting two plots, thus making a transaction time slice operation. For sake of simplicity full dates are omitted from the plot, leaving only day value.

Slicing can basically tell you the valid state of object at given amount in time. As an example (look at table 13), at transaction time 11 we can observe that Jack had valid book from $10^{th}$ to $15^{th}$ of October, after that book ownership was given to Jill. Please note that table's 13 data corresponds to previous figure 13.

| **Name** | **Book** | **Valid time** | **Transaction time** |
|---|---|---|---|
| Jack | Moby-Dick | 2017-10-10 - $\infty$ | 2017-10-10 - 2017-10-15 |
| Jack | Moby-Dick | 2017-10-10 - 2017-10-15 | 2017-10-15 - $\infty$ |
| Jill | Moby-Dick | 2017-10-15 - $\infty$ | 2017-10-15 - $\infty$ |

Table 13. Bitemporal relation for transaction time slice

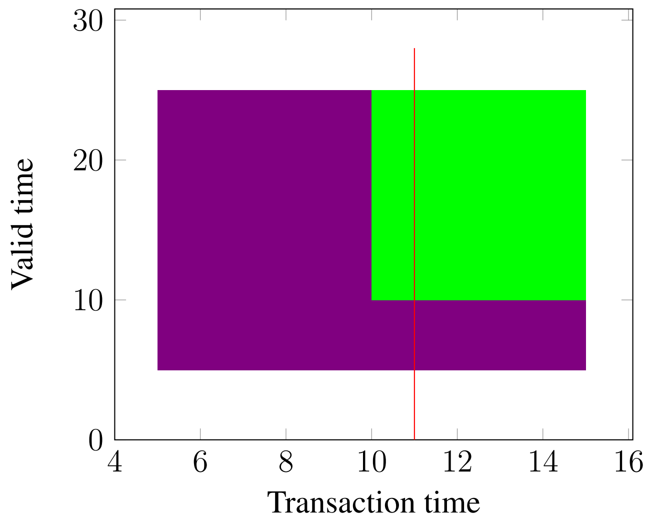


Figure 13. Transaction valid time slicing visualization

Pseudo-SQL vertical slice query in listing 2 shows a way bitemporal query can be written. In the query keywords "**VALIDTIME AS OF TIMESTAMP**" followed by a timestamp *"2017-10-11"* identifies valid time slice command. In practice this query should be converted into database query language command.

```
VALIDTIME AS OF TIMESTAMP "2017-10-11"
SELECT "Name", "Book", "Valid␣time" FROM Books
```

Listing 2: Vertical slice query example

So after slicing you will get this result as shown in table 14:

| Name | Book | Valid time |
|---|---|---|
| Jack | Moby-Dick | 2017-10-10 - 2017-10-15 |
| Jill | Moby-Dick | 2017-10-15 - $\infty$ |

Table 14. Result of slicing transaction time

This means that Jack owner book from $5^{th}$ October until $10^{th}$ of October until Jill finally became the owner from $10^{th}$ of October.

## 6.6 Horizontal slice

Alternatively, you can slice "Valid time" axis to intersect its dimension values. This slice is visualized in figure 14.

Below (see table 15) you can observe that sample used in horizontal slice.

| Name | Book | Valid time | Transaction time |
|---|---|---|---|
| Jack | Moby-Dick | 2017-10-10 - $\infty$ | 2017-10-05 - 2017-10-10 |
| Jack | Moby-Dick | 2017-10-10 - 2017-10-15 | 2017-10-15 - $\infty$ |
| Jill | Moby-Dick | 2017-10-15 - $\infty$ | 2017-10-10 - $\infty$ |

Table 15. Bitemporal relation for valid time slice

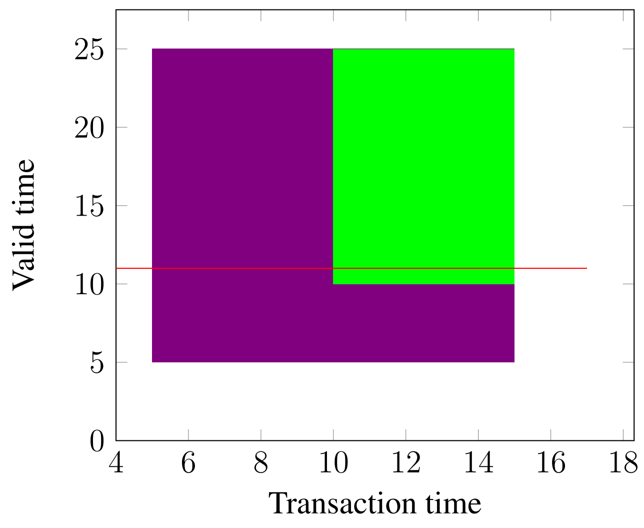


Figure 14. Valid time slicing visualization

As discussed in section 6.5, keyword **"TRANSACTIONTIME AS OF TIMESTAMP"** identifies valid time slice. In pseudo-SQL horizontal slice query (look at listing 3) can be written like this:

```
TRANSACTIONTIME AS OF TIMESTAMP "2017-10-11"
SELECT "Name", "Book", "Transaction␣time" FROM books
```

Listing 3: Vertical slice query example

Such operation will yield this result outlined in table 16:

| **Name** | **Book** | **Transaction time** |
|---|---|---|
| Jack | Moby-Dick | 2017-10-05 - 2017-10-10 |
| Jill | Moby-Dick | 2017-10-10 - $\infty$ |

Table 16. Result of slicing valid time

## 6.7 Bitemporal slice

Lastly, bitemporal slicing needs Valid time point and transaction as time point input. It will result in snapshot state of an object. This can illustrated here in figure 15:

| **Name** | **Book** | **Valid time** | **Transaction time** |
|---|---|---|---|
| Jack | Moby-Dick | 2017-10-10 - $\infty$ | 2017-10-10 - 2017-10-15 |
| Jack | Moby-Dick | 2017-10-10 - 2017-10-15 | 2017-10-15 - $\infty$ |
| Jill | Moby-Dick | 2017-10-15 - $\infty$ | 2017-10-15 - $\infty$ |

Table 17. Bitemporal relation for transaction and valid time slices

As a result (refer to table 18) we can see that at a bitemporal time-slice on a valid time of October $15^{th}$ and as of a transaction time of October $11^{th}$ resulted in Jill being the owner of the book at that point in time. In pseudo-SQL vertical slice query (look at listing 4) can be written like this:

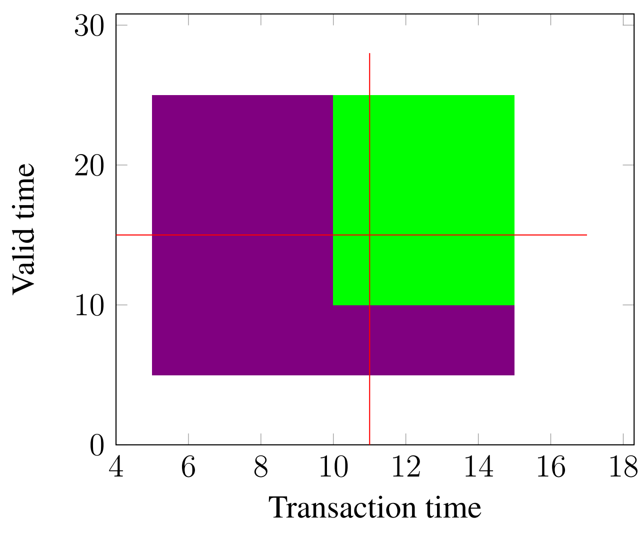


Figure 15. Transaction and valid time slicing visualization

```
TRANSACTIONTIME AS OF TIMESTAMP "2017-10-11"
AND
VALIDTIME AS OF TIMESTAMP "2017-10-15"
SELECT "Name", "Book", "Transaction␣time" FROM books
```

Listing 4: Bitemporal slice pseudo-query example

| **Name** | **Book** |
|---|---|
| Jill | Moby-Dick |

Table 18. Result of slicing transaction and valid time dimensions

## 6.8 Data definition

A unit of data being saved to NoSQL databases can be loosely described as a document. Within it, this document can have a number of properties, represented using key-value pairs. Some of those properties can be other documents. A simplified BNF[10] can be found at listing 5.

```
<symbols> := "character[character...]"
<key> := <symbols>
<value> := <symbols>|<document>
<field> := <key>:<value>
<document> := {<field>[<field>...]}
```

Listing 5: A simplified BNF explaining the main relationships in the data

A given *document* is made up of individual *fields*. Each field is a key-value pair, where the key is a sequence of characters and the value is either a sequence of characters or another document, nested within the first one. The values can only be series of symbols because all data is serialized into text form prior to storage. Because nesting documents is possible, a given document $d \in D$ (where $D$ is the set of all documents) might be a nested structure, such as in figure 16, where each tree node represents a different document.

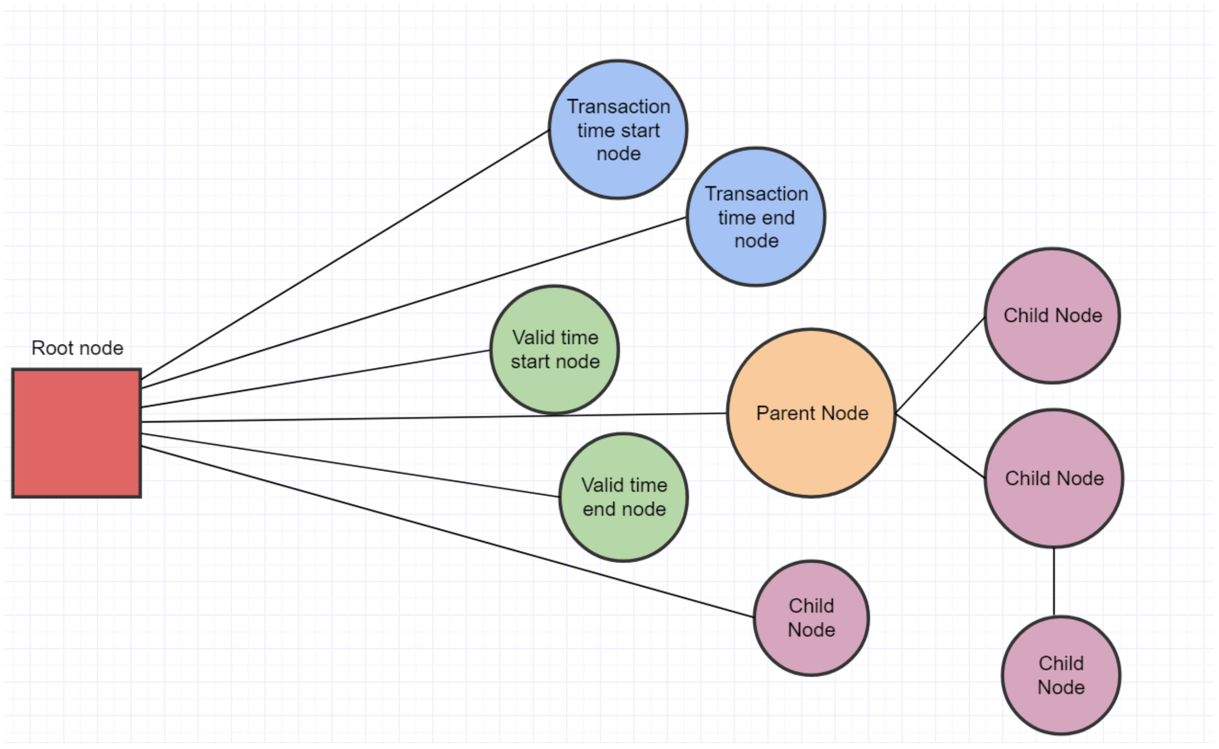


Figure 16. A nested document structure, each node is a separate document.

## 6.9 Output data association

The existing document models being stored into NoSQL databases are not bitemporal. They can have temporal properties among the document fields, but there is no guarantee. It makes sense

[10]BNF (Backus Normal Form or Backus–Naur Form) is one of the main notation techniques for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols. It is applied wherever an exact description of a language is needed.

to describe how a NoSQL document would look after being made bitemporal.

Because the database only stores documents with serialized data in their fields, the addition of 2 temporal dimensions can be described as adding 4 more values to the document (*transaction start, transaction end, validity start and validity end*). The bitemporal document, made up of the original with 4 more additional values can then be described as

$$d_t = (d, ts, te, vs, ve)$$

, where $d \in D$ and the remaining 4 represent the temporal values described, respectively.

The databases do not store tuples of 4 dimensions, but only singular documents. To reduce the set of required data into a single document, the document description BNF can be extended with listing 6. Now, the document is required to contain all possible bitemporal value fields.

```
<field_tt_start>:="tt_start":<symbols>
<field_tt_end>:="tt_end":<symbols>
<field_vt_start>:="vt_start":<symbols>
<field_vt_end>:="vt_end":<symbols>
<document>:={<field_tt_start><field_tt_end><field_vt_start><
   field_vt_end>
<field>[<field>...]}
```

Listing 6: The BNF extension used to store bitemporal data

Lets call the set of these extended documents $V$. While theoretically, $D \subset V$, from a purely technical standpoint, a random collection of symbol values that make up a field key for any document can take the form of a temporal key as well, meaning, that any NoSQL database that can store documents $d \in D$ can also store documents $v \in V$.

## 6.10 Association algorithm

Having the defined sets $D$ and $V$, the required association algorithm that is executed at the moment of storage can be described as the projection:

$$f : D \to V$$

The algorithm of this projection executes is made less trivial by the series of nested document structures that can make up a document tree at the time it is being saved. Such an algorithm should take the current *tt_start, tt_end, vt_start, vt_end* values of a document's child documents into account when deciding on the values to assign. A recursive algorithm $t(d, dts, dte, dvs, dve)$ is described at 1. The values $dts, dte, dvs, dve$ represent defaults for *transaction start, transaction end, validity start, validity end* respectively.

**Algorithm 1.** Make a nested document structure bitemporal prior to insertion.

```
Require: $d \in D$
  $curr\_node \Leftarrow DFS(d)$
  if $!curr\_node.tts$ then
    $curr\_node.tts \Leftarrow nvl(min\_child\_tts(curr\_node), dts)$
  end if
  if $!curr\_node.tte$ then
    $curr\_node.tts \Leftarrow nvl(max\_child\_tte(curr\_node), dte)$
  end if
  if $!curr\_node.vts$ then
    $curr\_node.tts \Leftarrow nvl(min\_child\_vts(curr\_node), dvs)$
  end if
  if $!curr\_node.vte$ then
    $curr\_node.tts \Leftarrow nvl(max\_child\_vte(curr\_node), dve)$
  end if
  if $curr\_node = d$ then
    return $d$
  else
    $t(d, dts, dte, dvs, dve)$
  end if
```

The basic idea is that when a document contains child documents nested into it, those child documents need to dictate the interval ranges of the bitemporal properties of that document. The default values supplied to the function ($dts, dte, dvs, dve$) are needed for nodes that have no more children to rely on for their intervals. In general, this algorithm should make sure, that any given parent document will have a *validity start* and *transaction start* no later than the earliest of its children and will have a *validity end* and *transaction end* no sooner than the latest of its children. The utility function $nvl(a, b)$ returns the second argument if the first one is an empty set, otherwise it returns the first. One thing to be noted about the functions $min\_child...$ and $max\_child...$ is that because the document tree is traversed in a depth-first manner (using the $DFS$[11] algorithm), all of the children of a given document will already have all 4 dimensional values assigned to them before the parent gets inspected. Therefore, it is possible to only check the immediate children of any document at that stage and still get the valid values required. The conditional blocks checking for the presence of known keys are a safeguard in case a bitemporal dimension was already defined for the data, as the algorithm would not want to override those.

The association algorithm complexity by time is loosely defined. Let's define it for $n$ document trees with an height of $h$ and an average layer width of $w$. Then,

$$O(n, w, h) \approx n * (O(DFS) * 4w * h)$$

, where $O(DFS)$ is the algorithm complexity by time of the standard tree-search DFS algorithm. The complexity for a single document tree is multiplied by $n$ to measure it for $n$ trees. For a single tree, the algorithm is repeated on every layer, meaning that the complexity for a single layer needs to be multiplied by $h$. On a single layer, 4 temporal values are written by checking all descendants of a specific node, for an average of $w$ per layer. Lastly, to get to that node for assignments, a DFS needs to be performed, this is added for every layer.

[11] Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures.

# 7 Implementation

This chapter introduces proof-of-concept for relational and non-relational databases. In the beginning application is visualized. Following section outline point out environment, sample dataset. Later you can observe how bitemporal data is stored in various databases from performance and implementation perspective. For the rest of this chapter bitemporal slicing performance is analyzed. In the last section, you can read comparison summarized results.

Previous scientific research has shown that storing bitemporal data in relational database is non-trivial task and strictly depends on database management system. Non only it requires manual intermission, but it's dependable on programming language as well. However, storing and extracting bitemporal data in non-conventional database management system is challenging task. Also, for comparison purpose, relational database PostgreSQL is included in the test cases as well.

## 7.1 Application architecture

Following figure 17 shows basic overview of application used to measure performance.

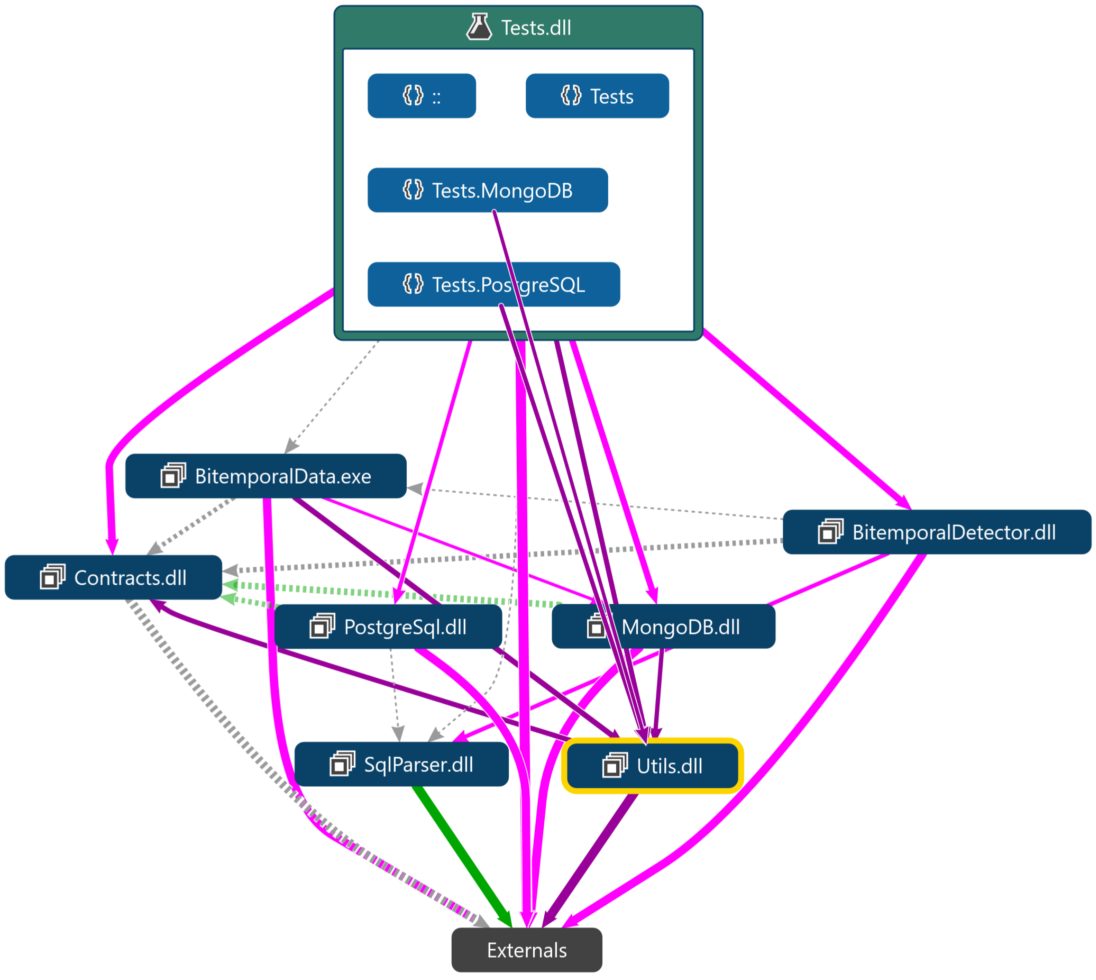


Figure 17. Bitemporal application overall architecture

- **Test** assembly is an entry point to application, executing various types of integration and performance tests.

- **Contracts** interfaces and abstract classes used throughout the project
- **BitemporalDetector** interprets given pseudo commands into database language.
- **BitemporalData** prepares given dataset for data manipulation into given data source
- **PostgreSQL** project holds PostgreSQL related algorithms to achieve bitemporality.
- **MongoDB** project holds MongoDB related algorithms to achieve bitemporality.
- **Utils** bitemporal utilities and helper collection
- **SqlParser** is a helper project that parses SQL standard queries.
- **External** sources is various NoSQL and SQL data adapters used thought out this project

## 7.2 Testing

To test given model hypotheses xUnit testing framework [33] was selected. xUnit lets you write easier test cases using .NET framework. As a result a series on test collections were created, spanning from bitemporal insertion to data slicing operations. In order to organize multitude of tests various test traits were assigned - "Slicing", "Insertion", "Change", "Deletion".

## 7.3 CLI application

Alternatively, there is a command line interface application. Application can accept multiple parameters and produce certain output. Using command parameters application can differentiate between database types and import formats. Listing 7 shows what command parameters user can select.

```
./Bitemporality.exe database_type import_format
    file_location [config_file_location]
```

Listing 7: Command's parameter description

So far only options for database_type are "MongoDB, "CouchDB" or "Redis". Also, "import_format" can be either "CSV" or "JSON". Following parameter is actual file location. Last application's parameter is optional. In configuration file user can specify database connection string. However, if nothing is specified, application will connect to local database without authentication.

## 7.4 Testing environment

As a test machine, a computer with these hardware parameters was used:

- CPU: Intel Core i7 6700, 3.8 GHz
- Memory: DDR4 16 GB
- Disk drive: 256 GB rapid solid-state drive.
- OS: Microsoft Windows 10
- Programming framework: C# based on .NET 4.6.1

## 7.5 Dataset

In order to test different database performance, a dataset containing 167939 rows was used [6]. Test data contains complete baseball batting and pitching statistics from *1871* to *2014*. Description of sample data can be found in table 19.

| **Column** | **Description** |
|---:|---|
| playerID | Player ID code |
| yearID | Year |
| stint | player's stint (order of appearances within a season) |
| teamID | Team |
| lgID | League |
| Pos | Position |
| G | Games |
| GS | Games Started |
| InnOuts | Time played in the field expressed as outs |
| PO | Putouts |
| A | Assists |
| E | Errors |
| DP | Double Plays |
| PB | Passed Balls (by catchers) |
| WP | Wild Pitches (by catchers) |
| SB | Opponent Stolen Bases (by catchers) |
| CS | Opponents Caught Stealing (by catchers) |
| ZR | Zone Rating |

Table 19. Description of test dataset

## 7.6 Dataset parsing

Dataset was presented in CSV[12] format. Therefore, parsing utility was created. It allows to convert this dataset file of baseball fielding data to MongoDB, PostgreSQL and other formats. It supports CSV-to-BSON[13], CSV-to-SQL, CSV-to-POCO[14] type of parsing. Following code snippet (see listing 8) will parse CSV value structure into BSON.

```
public static IEnumerable<BsonDocument> ParseCSVToBSON(string[] content)
{
    var headers = content
        .First()
        .Split(',')
        .AsParallel()
        .Select(item => item)
        .ToList();
```

[12]Comma-separated value

[13]BSON (Binary JSON) is a computer data interchange format used mainly as a data storage and network transfer format in the MongoDB database

[14]Plain old CLR object (POCO) is a simple object created in the Common Language Runtime (CLR) of the .NET Framework which is unencumbered by inheritance or attributes.

```
    var result = content.Skip(1).AsParallel().Select(value =>
    {
      var bson = new BsonDocument();
      int index = 0;
      bson.AddRange(value
          .Split(',')
          .Select(element => new BsonElement(headers[index++], element)));

      return bson;
    });

    return result;
}
```

Listing 8: Parsing CSV into BSON format

Algorithm takes CSV file headers, assigns them to a list. Then it loops though file, skipping headers. During loop it creates new *BsonDocument* instance with *BsonElement* items accordingly. It is worth mentioning that element keys are taken from CSV header list according to array index at the time. Final result is a collection of BSON objects. All content iterations operation are done in parallel using .NET LINQ[15] library to decrease dataset preparation time.

## 7.7 Valid and transaction time values

As for this moment, application supports only new bitemporal data. This means that, if tool encounters an object without bitemporal domains, it will append them accordingly to the final collection. Bitemporal domains are added to intermediary BSON data before insertion. Following algorithm (see listing 9) shows how addition BSON elements are added to initial data.

```
public static BsonDocument AddTimeDimensions(this BsonDocument content,
    DateTime validTime)
{
  content["vt_start"] = validTime;
  content["vt_end"] = DateTime.MaxValue;

  content["tt_start"] = DateTime.UtcNow;
  content["tt_end"] = DateTime.MaxValue;

  return content;
}
```

Listing 9: Code snippet that adds time dimensions

There's an additional aspect of this: by default tuple is considered valid, therefore transaction time as well as valid time ends should be of maximum value. Although, if you need to insert tuple with customer bitemporal parameters, it's possible to do so. There is a method overload that does accept valid and transaction time *DateTime* intervals as parameters.

Furthermore, UTC[16] timing is recorded and used throughout application. It's important to use globally coordinate time and not store additional information about timezones. It is done to avoid major data inconsistencies. If a need arises, application could convert dates back to user in a preferred timezone.

[15]Language-Integrated Query

[16]Coordinated Universal Time

## 7.8 MongoDB implementation

Connecting to MongoDB works though a connector class **MongoClient** which requires a singleton connection. All operations done with .NET framework are asynchronous. For successful connection to database, we can use listing 10 code.

```
var client = new MongoClient();
var database = Client.GetDatabase(databaseName);
```

Listing 10: Code snippet that allows to connect to MongoDB

Next snippet shows collection insertion into document-store. MongoDB inserts BSON[17] input. Therefore, all storage data must be converted to *BsonDocument* class object before insertion. Listing 11 shows main insertion method for MongoDB.

```
var collection = Database.GetCollection<BsonDocument>(collectionName);
await collection.InsertOneAsync(document);
```

Listing 11: Inserting collection into MongoDB

## 7.9 CouchDB implementation

CouchDB uses similar .NET connection wrapper as MongoDB. It is called MyCouchClient [32]. Listing 13 shows how database connection can be opened.

```
var couchClient = new MyCouchClient(address, databaseName);
```

Listing 12: Connecting to CouchDB

Inserting data into CouchDB happens in similar manner as Mongo. However, database stores document data differently. CouchDB uses plain JSON as storage type. Since application had to parse plain JSON, a parsing library was used - Json.NET [22]. It helped to properly parse application's input into JSON. In listing 13, you insertion can be initiated.

```
Client.Documents.PostAsync(data);
```

Listing 13: Inserting collection into CouchDB

## 7.10 Redis implementation

For successfully connection to Redis database a StackExchange [31] interaction library was used. It allowed to connect and perform operations on Redis key-value store. In listing 14, you can see how it can be implemented.

```
var redis = ConnectionMultiplexer.Connect(address);
RedisDatabase = redis.GetDatabase();
```

Listing 14: Connecting to key-value database Redis

In order to insert data into Redis key-value store, it was decided that a single object's JSON data will represent value and key will be value SHA1 hash result. Therefore, all objects with corresponding hashes could be stored. Listing 15 uses *System.Security.Cryptography* utility library to compute hash value for a given JSON input. Inserting key-value sets into Redis happens through a setter method which takes key and value as its parameters.

[17]Binary JSON

```
var thumbprint = Utils.GetSHA1(data);
RedisDatabase.StringSet(key, data);
```

Listing 15: Inserting into Redis

## 7.11 PostgreSQL implementation

PostgreSQL [12] was used as main relational database management system. In addition to that, to SQL Parser [11] library was used to manipulate SQL queries into .NET data structures. In order to query bitemporal data in .NET, Dapper [30] framework was used. Dapper is micro-ORM[18] library that let us perform queries from .NET platform with less overhead and with balanced convenience. Moreover, any interaction with bitemporal PostgreSQL database goes though ADO.NET's *NpgsqlConnection* connection (see listing 16).

```
using (var connection = GetConnection())
{
    await connection.ExecuteAsync(query, data);
}
```

Listing 16: PostgreSQL Dapper query

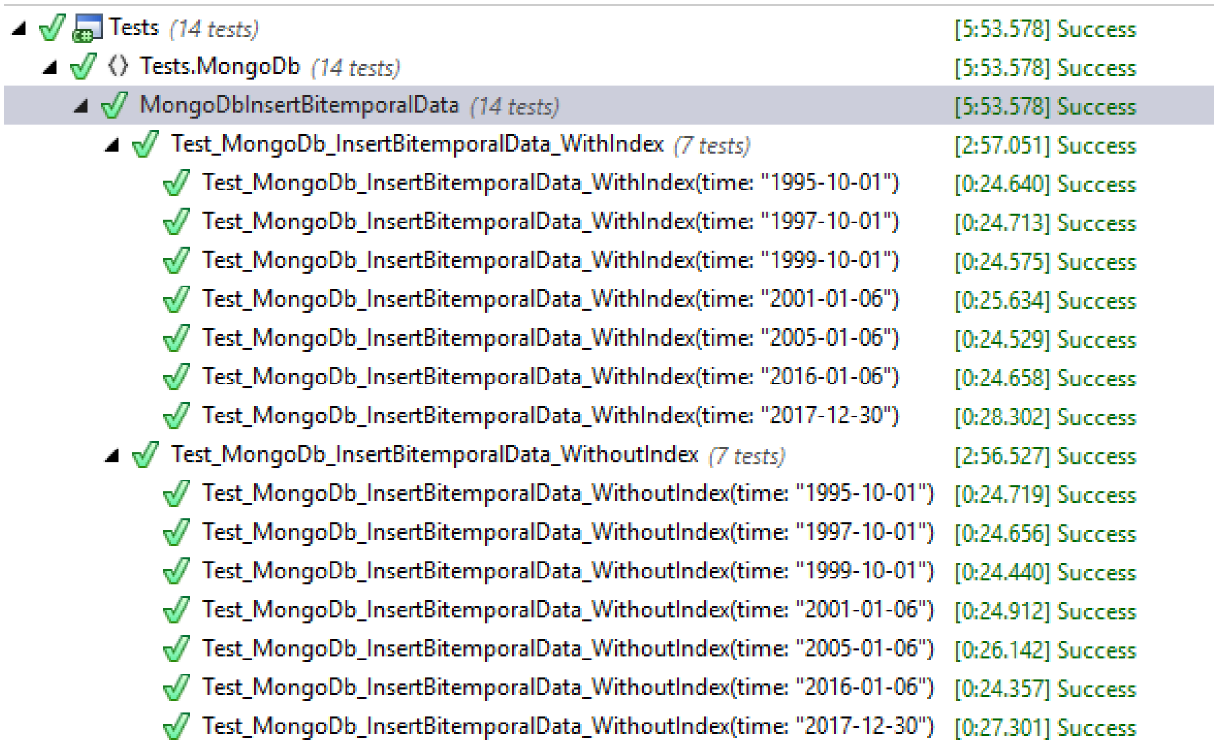
## 7.12 Insertion benchmarking

In order to compare different NoSQL database types, several data insertion tests were performed. Tests used fairly large dataset and powerful machine in closed environment. However, this thesis only covers singe node testing. In the future, it would be beneficial to test NoSQL scale-out and compare performance amongst nodes in the cluster.

### 7.12.1 MongoDB import

Following picture 18 shows MongoDB insertion statistics. Both test suites were executed on mirrored collection, containing same data, but with different indexing strategy. Additionally. you can take a look at execution times. It's quite predictable that collection without bitemporal indexes will perform write operations a bit quicker.

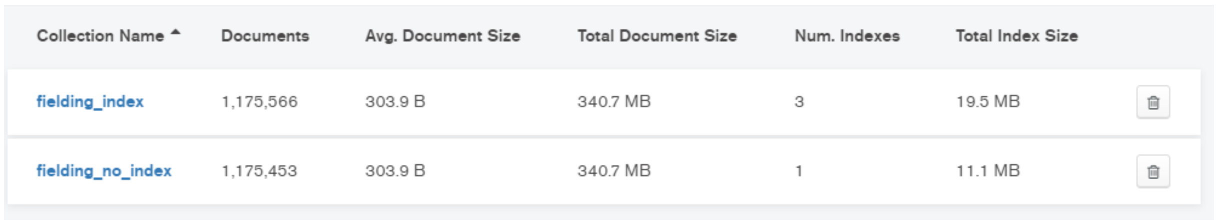
[18]Micro Object-Relational Mapper



```
◢ ✓ Tests (14 tests)                                                                        [5:53.578] Success
  ◢ ✓ ⟨⟩ Tests.MongoDb (14 tests)                                                           [5:53.578] Success
    ◢ ✓ MongoDbInsertBitemporalData (14 tests)                                              [5:53.578] Success
      ◢ ✓ Test_MongoDb_InsertBitemporalData_WithIndex (7 tests)                             [2:57.051] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "1995-10-01")                 [0:24.640] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "1997-10-01")                 [0:24.713] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "1999-10-01")                 [0:24.575] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "2001-01-06")                 [0:25.634] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "2005-01-06")                 [0:24.529] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "2016-01-06")                 [0:24.658] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithIndex(time: "2017-12-30")                 [0:28.302] Success
      ◢ ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex (7 tests)                          [2:56.527] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "1995-10-01")              [0:24.719] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "1997-10-01")              [0:24.656] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "1999-10-01")              [0:24.440] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "2001-01-06")              [0:24.912] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "2005-01-06")              [0:26.142] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "2016-01-06")              [0:24.357] Success
          ✓ Test_MongoDb_InsertBitemporalData_WithoutIndex(time: "2017-12-30")              [0:27.301] Success
```

Figure 18. MongoDB insertion test suites

As a result (shown in figure 19) we can observe that on average storage of "Fielding" collection doesn't take much space. For almost 2 millions of records, it only take 300 MB of disk storage. Besides, indexing doesn't introduce any storage issues - only 20 MB in size is considered normal of an amount of data inserted.



| Collection Name ⏶ | Documents | Avg. Document Size | Total Document Size | Num. Indexes | Total Index Size |
|---|---|---|---|---|---|
| fielding_index | 1,175,566 | 303.9 B | 340.7 MB | 3 | 19.5 MB |
| fielding_no_index | 1,175,453 | 303.9 B | 340.7 MB | 1 | 11.1 MB |

Figure 19. MongoDB bitemporal collections

In addition to this, we can see (look at figure 20) very low usage of both CPU and memory. Total runtime in MongoDB accounted is 25.28 seconds on average.

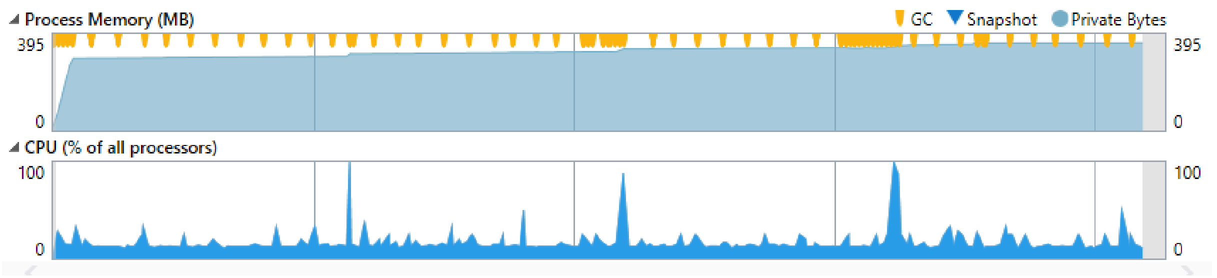
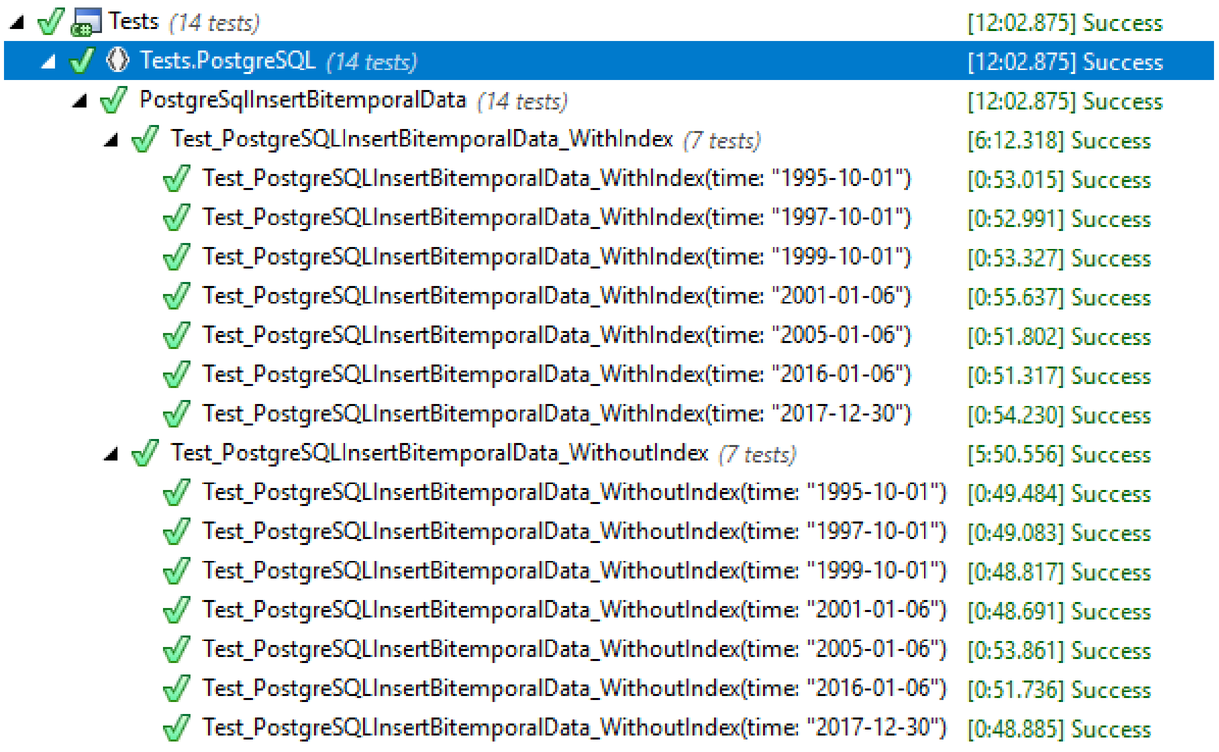


Figure 20. Result of profiling MongoDB

### 7.12.2 PostgreSQL import

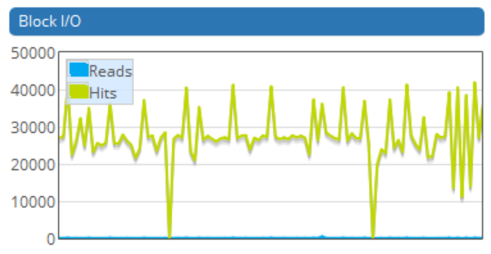
Figure 21 shows PostgreSQL bitemporal write results. It's done in similar manner with MongoDB tests. Additionally, we can compare you can take a look at execution times. It's quite apparent that relational PostgreSQL will take significantly more time than document system, therefore write operations are much more slower here - almost 2.4 slower than MongoDB (outlined in figure 18).



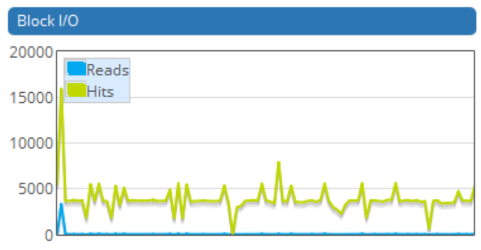
```
◢ ✓ Tests (14 tests)                                                          [12:02.875] Success
  ◢ ✓ Tests.PostgreSQL (14 tests)                                             [12:02.875] Success
    ◢ ✓ PostgreSqlInsertBitemporalData (14 tests)                             [12:02.875] Success
      ◢ ✓ Test_PostgreSQLInsertBitemporalData_WithIndex (7 tests)             [6:12.318] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "1995-10-01")    [0:53.015] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "1997-10-01")    [0:52.991] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "1999-10-01")    [0:53.327] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "2001-01-06")    [0:55.637] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "2005-01-06")    [0:51.802] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "2016-01-06")    [0:51.317] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithIndex(time: "2017-12-30")    [0:54.230] Success
      ◢ ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex (7 tests)          [5:50.556] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "1995-10-01") [0:49.484] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "1997-10-01") [0:49.083] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "1999-10-01") [0:48.817] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "2001-01-06") [0:48.691] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "2005-01-06") [0:53.861] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "2016-01-06") [0:51.736] Success
          ✓ Test_PostgreSQLInsertBitemporalData_WithoutIndex(time: "2017-12-30") [0:48.885] Success
```

Figure 21. PostgreSQL insertion test suites

Furthermore, in here same indexed vs. non-indexed testing strategy was used. You can see than non-index inserts are moderately quicker.



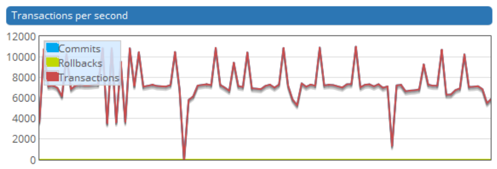
(a) PostgreSQL IO with bitemporal indexes



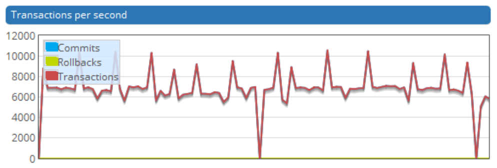
(b) PostgreSQL IO without indexes

If we examine these graphs (figures 22a and 22b) we clearly see significant drop of IO. Graph of the measures IO of indexes bitemporal relation while graph on the left take IO readings of plain non-indexed inserts.

On the other hand, transaction number for PostgreSQL remained the same. If we examine figures 23a and 23b, we can tell that on average there were around 7000 transactions per second registered. Alternatively, you can desire this hypothesis from previously shown Figure 18 where execute were approximately the same.



(a) PostgreSQL transactions per second metric with indexes



(b) PostgreSQL transactions per second metric without indexes

### 7.12.3 Other NoSQL database results

In figure 24 we can observe CouchDB memory and CPU usage that fluctuate over time of data insertion. At the start of data insertion, we can clearly see the growth in RAM, whereas CPU is almost fully utilized. With intensive utilization of CPU and memory, CouchDB managed to insert all 167939 documents in 48.31 seconds, which is fastest result.

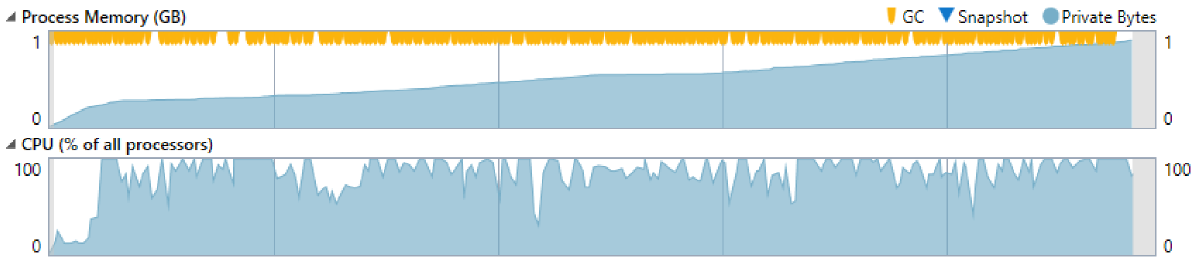


Figure 24. Result of profiling CouchDB

However, Redis performed the least, with insertion lasting over 8 minutes. In figure 25, we can observe slight CPU usage with moderate amount of RAM used.

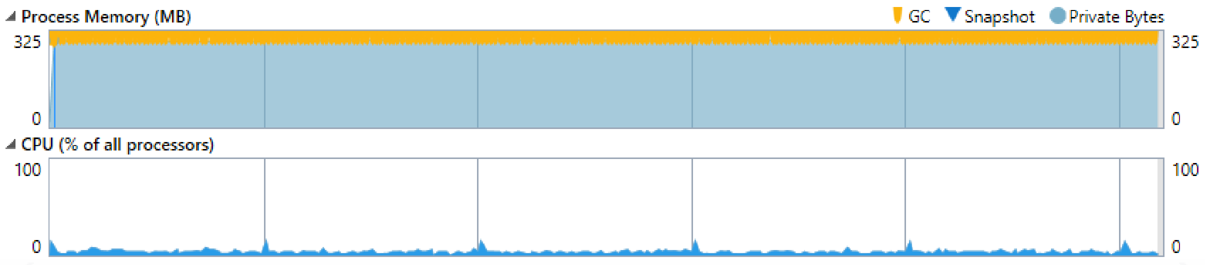


Figure 25. Result of profiling Redis

### 7.12.4 Summary of example insertion results

Summarized results of performed data insertion tests can be found in table 20.

| Name | Database type | Time elapsed (s) | Memory usage | Processor usage | Disk IO |
|---|---|---|---|---|---|
| CouchDB | Document store | 48.31 | Moderate | High | High |
| MongoDB | Document store | 88.52 | Low | Low | Moderate |
| PostgreSQL | Relational | 175.50 | Low | Low | Moderate |
| Redis | Key-value store | 518.09 | Moderate | Low | High |

Table 20. Comparison of NoSQL insertion performance

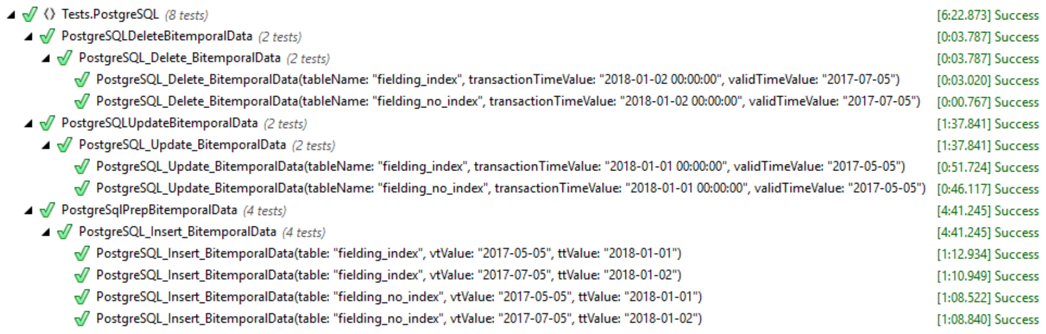
CouchDB and MongoDB performed fastest. While PostgreSQL was twice as slow to insert bitemporal dataset. Test results in table 20 were taken of non-indexed entities. Benchmarks performed in section 7.13.1 will take indexing and other bitemporal data operations into account.

## 7.13 Change and deletion benchmarking

Following figures 26 and 27 outline test suite results. At the beginning preparation bitemporal data is **inserted** then part of the is **deleted**, while other part is **updated**. Below is more detailed description of performance test steps:

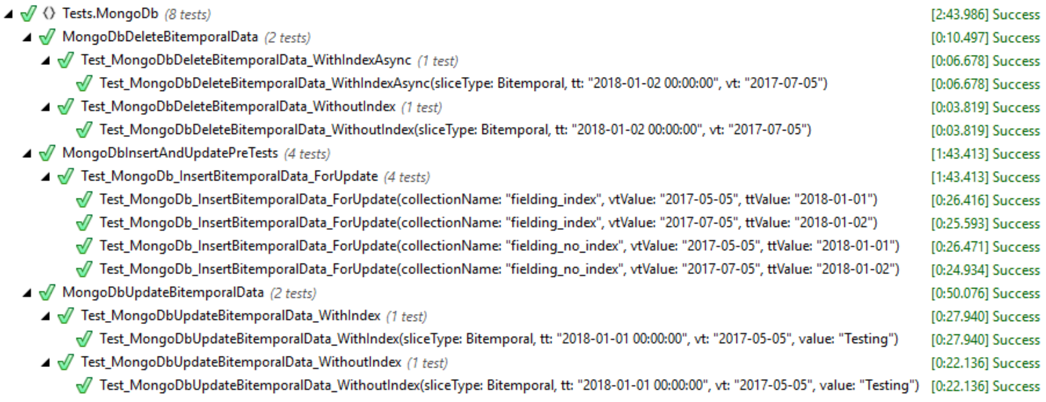
- **Insert** bitemporal dataset (described in section 7.5) with transactional time *"2018-01-01"* and *"2018-01-02"*
- **Delete** data inserted on *"2018-01-01"* when it is valid on *"2017-07-05"*
- **Update** Zone Rating value to 1 on data inserted *"2018-01-02"* when it is valid on *"2017-05-05"*

Since for updates and deletes we are using different transaction times (*"2018-01-01"* and *"2018-01-02"*), it ensures that bitemporal operations results will be segregated and no double updates or inserts will occur.



```
◢ ✓ ⟨⟩ Tests.PostgreSQL (8 tests)                                                                                                              [6:22.873] Success
  ◢ ✓ PostgreSQLDeleteBitemporalData (2 tests)                                                                                                 [0:03.787] Success
    ◢ ✓ PostgreSQL_Delete_BitemporalData (2 tests)                                                                                             [0:03.787] Success
        ✓ PostgreSQL_Delete_BitemporalData(tableName: "fielding_index", transactionTimeValue: "2018-01-02 00:00:00", validTimeValue: "2017-07-05")    [0:03.020] Success
        ✓ PostgreSQL_Delete_BitemporalData(tableName: "fielding_no_index", transactionTimeValue: "2018-01-02 00:00:00", validTimeValue: "2017-07-05") [0:00.767] Success
  ◢ ✓ PostgreSQLUpdateBitemporalData (2 tests)                                                                                                 [1:37.841] Success
    ◢ ✓ PostgreSQL_Update_BitemporalData (2 tests)                                                                                             [1:37.841] Success
        ✓ PostgreSQL_Update_BitemporalData(tableName: "fielding_index", transactionTimeValue: "2018-01-01 00:00:00", validTimeValue: "2017-05-05")    [0:51.724] Success
        ✓ PostgreSQL_Update_BitemporalData(tableName: "fielding_no_index", transactionTimeValue: "2018-01-01 00:00:00", validTimeValue: "2017-05-05") [0:46.117] Success
  ◢ ✓ PostgreSqlPrepBitemporalData (4 tests)                                                                                                   [4:41.245] Success
    ◢ ✓ PostgreSQL_Insert_BitemporalData (4 tests)                                                                                             [4:41.245] Success
        ✓ PostgreSQL_Insert_BitemporalData(table: "fielding_index", vtValue: "2017-05-05", ttValue: "2018-01-01")                                 [1:12.934] Success
        ✓ PostgreSQL_Insert_BitemporalData(table: "fielding_index", vtValue: "2017-07-05", ttValue: "2018-01-02")                                 [1:10.949] Success
        ✓ PostgreSQL_Insert_BitemporalData(table: "fielding_no_index", vtValue: "2017-05-05", ttValue: "2018-01-01")                              [1:08.522] Success
        ✓ PostgreSQL_Insert_BitemporalData(table: "fielding_no_index", vtValue: "2017-07-05", ttValue: "2018-01-02")                              [1:08.840] Success
```

Figure 26. PostgreSQL end-to-end data manipulation



```
◢ ✓ ⟨⟩ Tests.MongoDb (8 tests)                                                                                                                     [2:43.986] Success
  ◢ ✓ MongoDbDeleteBitemporalData (2 tests)                                                                                                        [0:10.497] Success
    ◢ ✓ Test_MongoDbDeleteBitemporalData_WithIndexAsync (1 test)                                                                                   [0:06.678] Success
        ✓ Test_MongoDbDeleteBitemporalData_WithIndexAsync(sliceType: Bitemporal, tt: "2018-01-02 00:00:00", vt: "2017-07-05")                     [0:06.678] Success
    ◢ ✓ Test_MongoDbDeleteBitemporalData_WithoutIndex (1 test)                                                                                     [0:03.819] Success
        ✓ Test_MongoDbDeleteBitemporalData_WithoutIndex(sliceType: Bitemporal, tt: "2018-01-02 00:00:00", vt: "2017-07-05")                       [0:03.819] Success
  ◢ ✓ MongoDbInsertAndUpdatePreTests (4 tests)                                                                                                     [1:43.413] Success
    ◢ ✓ Test_MongoDb_InsertBitemporalData_ForUpdate (4 tests)                                                                                      [1:43.413] Success
        ✓ Test_MongoDb_InsertBitemporalData_ForUpdate(collectionName: "fielding_index", vtValue: "2017-05-05", ttValue: "2018-01-01")              [0:26.416] Success
        ✓ Test_MongoDb_InsertBitemporalData_ForUpdate(collectionName: "fielding_index", vtValue: "2017-07-05", ttValue: "2018-01-02")              [0:25.593] Success
        ✓ Test_MongoDb_InsertBitemporalData_ForUpdate(collectionName: "fielding_no_index", vtValue: "2017-05-05", ttValue: "2018-01-01")           [0:26.471] Success
        ✓ Test_MongoDb_InsertBitemporalData_ForUpdate(collectionName: "fielding_no_index", vtValue: "2017-07-05", ttValue: "2018-01-02")           [0:24.934] Success
  ◢ ✓ MongoDbUpdateBitemporalData (2 tests)                                                                                                        [0:50.076] Success
    ◢ ✓ Test_MongoDbUpdateBitemporalData_WithIndex (1 test)                                                                                        [0:27.940] Success
        ✓ Test_MongoDbUpdateBitemporalData_WithIndex(sliceType: Bitemporal, tt: "2018-01-01 00:00:00", vt: "2017-05-05", value: "Testing")         [0:27.940] Success
    ◢ ✓ Test_MongoDbUpdateBitemporalData_WithoutIndex (1 test)                                                                                     [0:22.136] Success
        ✓ Test_MongoDbUpdateBitemporalData_WithoutIndex(sliceType: Bitemporal, tt: "2018-01-01 00:00:00", vt: "2017-05-05", value: "Testing")      [0:22.136] Success
```

Figure 27. MongoDB end-to-end data manipulation

As a side note, judging by total test execution time, MongoDB was three times faster than PostgreSQL. Following section will in dive more details.

### 7.13.1 Data manipulation summary

Bitemporal data manipulation are summarized in table 21. Table is sorted by bitemporal operation, description and execution time in ascending order.

| Name | Bitemporal operation | Description | Time elapsed (s) | Memory usage | Processor usage | Disk usage |
|---|---|---|---|---|---|---|
| **MongoDB** | INSERT | Indexed timestamps | 26,004 | Low | Moderate | High |
| PostgreSQL | INSERT | Indexed timestamps | 71.944 | High | High | High |
| **MongoDB** | INSERT | Non-indexed timestamps | 25.702 | Low | Low | Low |
| PostgreSQL | INSERT | Non-indexed timestamps | 68,681 | High | Low | Moderate |
| **MongoDB** | UPDATE | Indexed timestamps | 27.940 | Moderate | Very high | Moderate |
| PostgreSQL | UPDATE | Indexed timestamps | 51.724 | Moderate | Very high | High |
| **MongoDB** | UPDATE | Non-indexed timestamps | 22.136 | Moderate | High | Moderate |
| PostgreSQL | UPDATE | Non-indexed timestamps | 46.117 | Low | High | Low |
| **PostgreSQL** | DELETE | Indexed timestamps | 3.020 | Moderate | Moderate | High |
| MongoDB | DELETE | Indexed timestamps | 6.780 | Moderate | Low | Moderate |
| **PostgreSQL** | DELETE | Non-indexed timestamps | 0.767 | Low | Moderate | Low |
| MongoDB | DELETE | Non-indexed timestamps | 3.819 | Low | Low | Moderate |

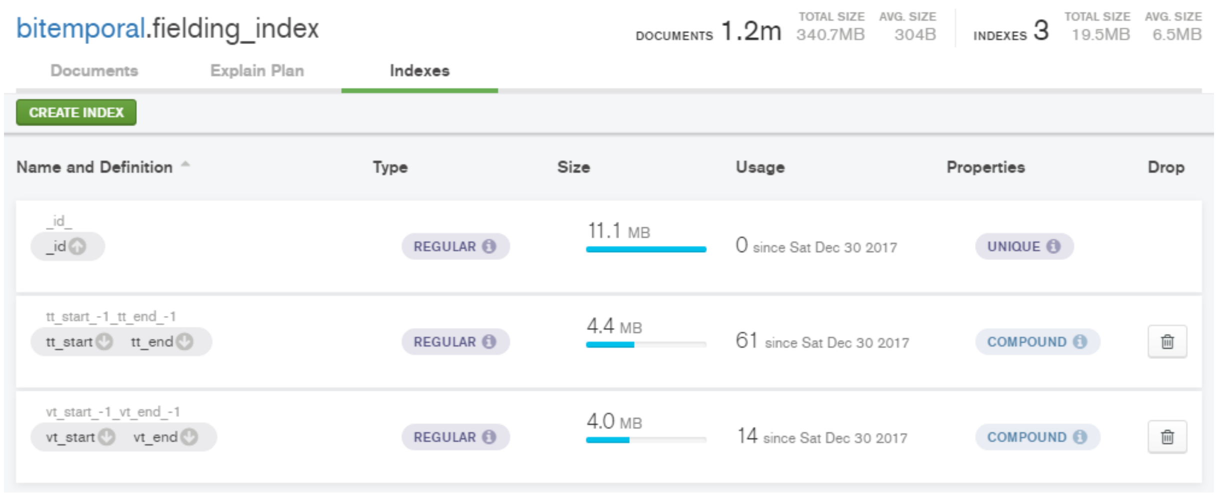
Table 21. Comparison of NoSQL data manipulation performance

Some preliminary conclusions can be aggregated from this:

- MongoDB bitemporal operations are faster under in similar conditions
- Bitemporal deletion in PostgreSQL yields betters results
- Indexed collections in PostgreSQL and MongoDB take more CPU and disk IO
- Insertion time for MongoDB is 3 times faster than relational storage
- Update in both MongoDB and PostgreSQL has very high CPU usage

## 7.14 Indexing strategy

Indexing bitemporal relation in a balanced manner can be quite cumbersome. When developing this proof-of-concept, it became quite clear how to best select indexing strategy. Finding a balance between write and read operations are no less quite important. As of now, it best to put separate compound indexes on time intervals - valid time and transaction time. Following picture (look at figure 28) will show summary of indexes created in MongoDB.


bitemporal.fielding_index

DOCUMENTS 1.2m TOTAL SIZE 340.7MB AVG. SIZE 304B INDEXES 3 TOTAL SIZE 19.5MB AVG. SIZE 6.5MB

| Name and Definition | Type | Size | Usage | Properties | Drop |
|---|---|---|---|---|---|
| _id_ (_id) | REGULAR | 11.1 MB | 0 since Sat Dec 30 2017 | UNIQUE | |
| tt_start_-1_tt_end_-1 (tt_start, tt_end) | REGULAR | 4.4 MB | 61 since Sat Dec 30 2017 | COMPOUND | 🗑 |
| vt_start_-1_vt_end_-1 (vt_start, vt_end) | REGULAR | 4.0 MB | 14 since Sat Dec 30 2017 | COMPOUND | 🗑 |

Figure 28. MongoDB bitemporal indexing strategy

While this collection has 2 million documents, index only take few megabytes of storage. In addition to this, data manipulation operations were not significantly affected by these indexes.

Another aspect of this is that indexed are of descending timestamp. Even though search criteria are not affected by ascending or descending type of order, but ordering of data does. It mainly affects MongoDB **sort()** function and PostgreSQL **ORDER BY** clause. Thus, it was decided to give priority to latest data in the entity. However, there could be use cases when extracting old data is more vital. In that case, it would be a good idea to add double indexes - with **ascending** order, while other should be **descending**.

## 7.15 Indexes in practice

After NoSQL entity is created, there is an option to add indexing. By default bitemporal properties of MongoDB collections are automatically indexed. However, there is an option to turn it off. It has proven to be useful for running various indexed vs. non-indexed tests. Listing 17 shows to descending indexes are created on both valid and transaction timestamp intervals.

```
collection.Indexes.CreateOneAsync(Builders<Fielding>.IndexKeys
    .Descending(_ => _.tt_start)
    .Descending(_ => _.tt_end)
);

collection.Indexes.CreateOneAsync(Builders<Fielding>.IndexKeys
    .Descending(_ => _.vt_start)
    .Descending(_ => _.vt_end)
);
```

Listing 17: MongoDB indexing snippet

On the other hand, PostgreSQL requires to write more explicit SQL statements. Follow listing 18 shows how indexed are created in PostgreSQL database.

```
var tasks = new[]
{
  connection.ExecuteAsync(@"CREATE␣INDEX␣IF␣NOT␣EXISTS␣@VT_Index␣ON␣
      @Table␣USING␣btree␣(vt_start␣DESC,␣vt_end␣DESC)", data),
  connection.ExecuteAsync(@"CREATE␣INDEX␣IF␣NOT␣EXISTS␣@TT_Index␣ON␣
      @Table␣USING␣btree␣(tt_start␣DESC,␣tt_end␣DESC)", data)
};

await Task.WhenAll(tasks);
```

Listing 18: PostgreSQL indexing snippet

Please note that both of listing 17 and 18 are using multi-threading for the creation of bitemporal indexes. That means that creation of index on application layer is parallelized on both MongoDB and PostgreSQL.

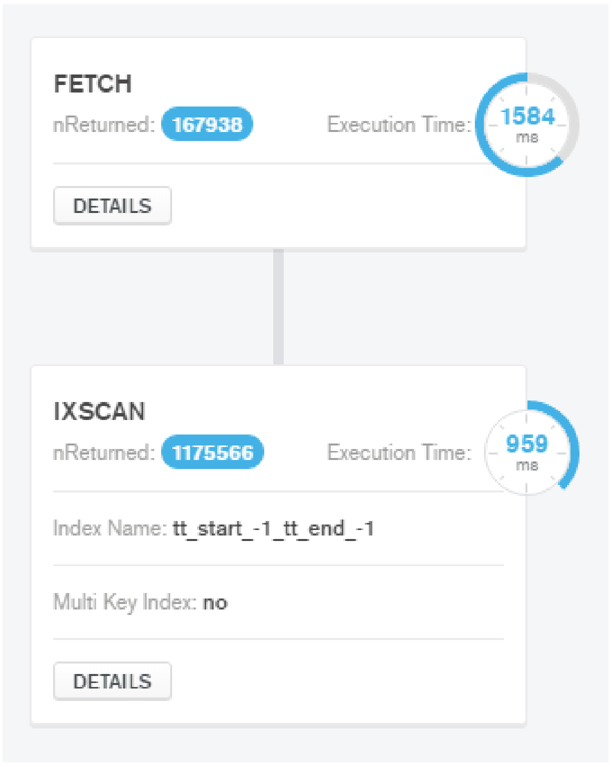
## 7.16 Slicing operations

Slicing operation is one of the most used data extraction technique in bitemporal databases. It introduces a way to look at the data in different time dimensions. As a proof of concept, application is able to test these scenarios:

- Slice valid time dimension
- Slice transaction time dimension
- Slice valid and transaction time dimensions
- Indexed valid time dimension slice
- Indexed transaction time dimension slice
- Indexed valid and transaction time slice

### 7.16.1 Slicing without indexes

An application will produce bitemporal query that is later sent to MongoDB for execution. Let's consider following example. On the left side (look at figure 29a) you can notice that index scan is performed, resulting in much more robust data extract time. Whereas figure 29b performs rather poorly, almost twice the time is long on full document scan.



FETCH

nReturned: 167938 Execution Time: 1584 ms

DETAILS

IXSCAN

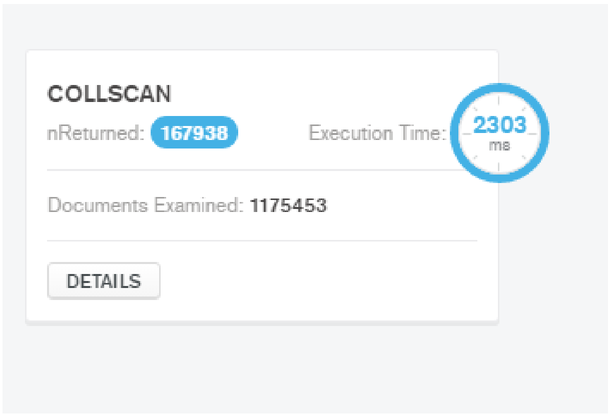
nReturned: 1175566 Execution Time: 959 ms

Index Name: tt_start_-1_tt_end_-1

Multi Key Index: no

DETAILS

(a) Result of query MongoDB



COLLSCAN

nReturned: 167938 Execution Time: 2303 ms

Documents Examined: 1175453

DETAILS

(b) Result of query MongoDB without indexes

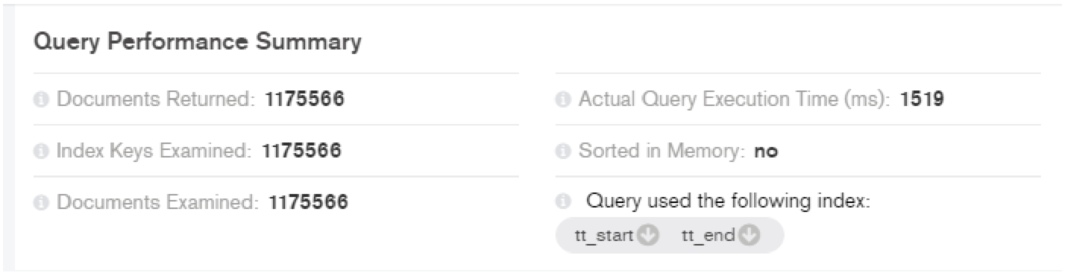
We can learn from this that indexing strategy mentioned in section 7.14, without a doubt, is an important part of bitemporal query execution. If incorrect indexing strategy is selected, performance of any query can significantly degrade. Obviously, full column scan for huge collections will always going to be delaying. However, in this we see that it more twice slower than regular index scan.

### 7.16.2 Slicing benchmarking

In these benchmark tests without indexes will be omitted. Following examples, encompass vertical, horizontal and bitemporal slicing.

### 7.16.3 Vertical slicing results

Vertical slicing operation described in section 6.5 were performed on MongoDB against 1175666 documents in the collection.



Query Performance Summary

Documents Returned: 1175566

Index Keys Examined: 1175566

Documents Examined: 1175566

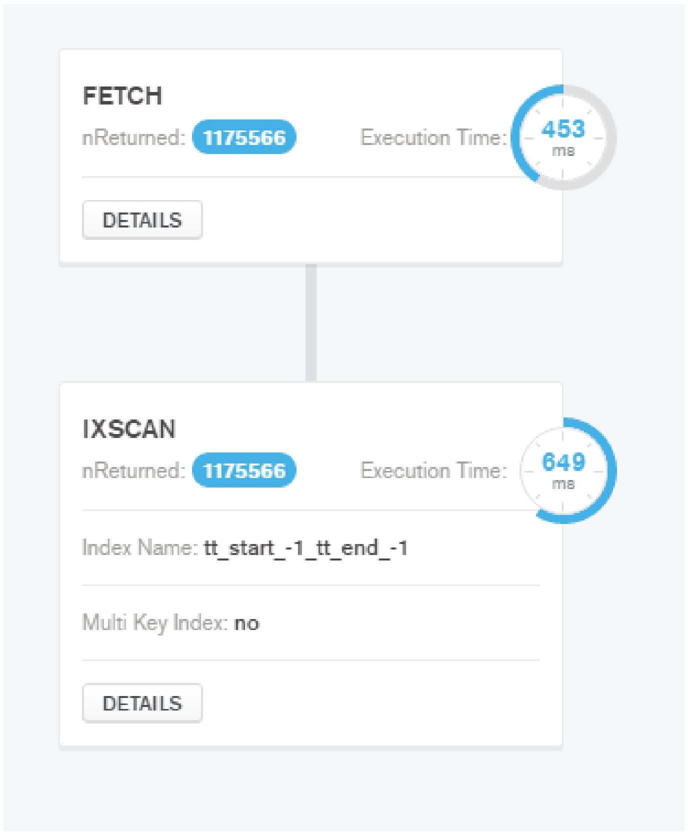
Actual Query Execution Time (ms): 1519

Sorted in Memory: no

Query used the following index: tt_start tt_end

Figure 30. Vertical slicing statistics

To visualize this, we can see clear transaction time index hit on bitemporal dimensions in "fielding_index" collection. Figure 30 shows that index scan took 649 milliseconds, while document's schema fetching completed in 453 milliseconds.

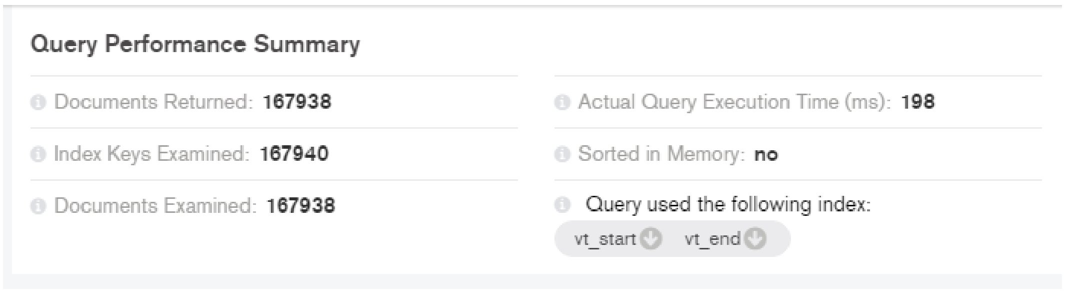

FETCH
nRetumed: 1175566 Execution Time: 453 ms
DETAILS

IXSCAN
nRetumed: 1175566 Execution Time: 649 ms
Index Name: tt_start_-1_tt_end_-1
Multi Key Index: no
DETAILS

Figure 31. Vertical slicing performance

### 7.16.4 Horizontal slicing results

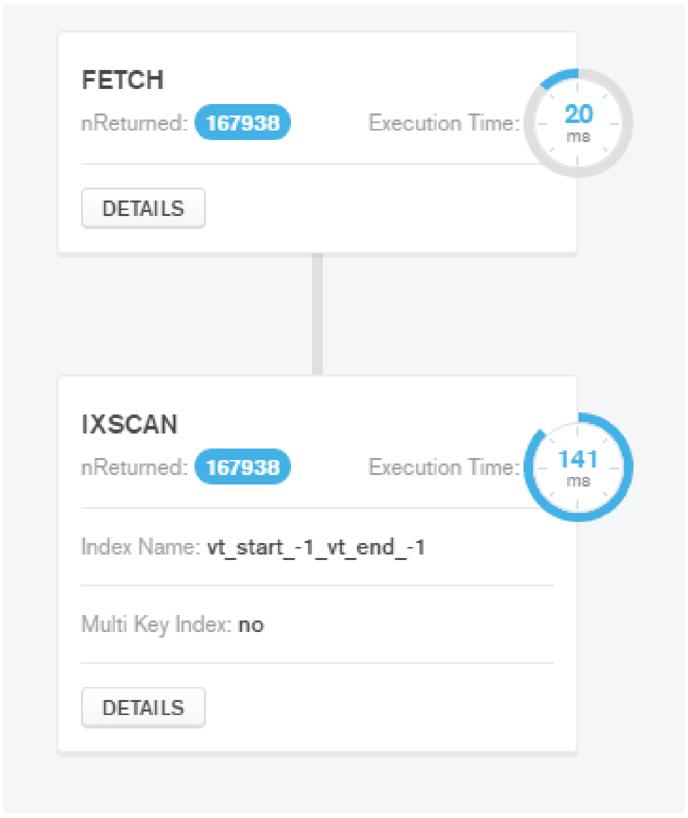
Horizontal slicing bitemporal query, in a collection of 1175666 documents found 167938 documents. Since valid time dimensions timestamps were scattered, less data was fetched. Thus it took this query significantly less time to execute.


Query Performance Summary

| | |
|---|---|
| Documents Returned: 167938 | Actual Query Execution Time (ms): 198 |
| Index Keys Examined: 167940 | Sorted in Memory: no |
| Documents Examined: 167938 | Query used the following index: vt_start vt_end |

Figure 32. Horizontal slicing statistics

Figure 33 shows that index scan only took 141 milliseconds, while document's schema fetched in as little as 20 milliseconds.

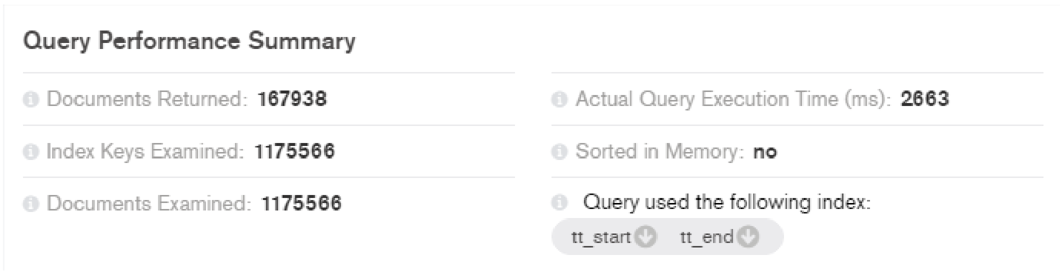

FETCH
nReturned: 167938
Execution Time: 20 ms

IXSCAN
nReturned: 167938
Execution Time: 141 ms
Index Name: vt_start_-1_vt_end_-1
Multi Key Index: no

Figure 33. Horizontal slicing performance

### 7.16.5 Bitemporal slicing results

Bitemporal slicing operation accounts for two time dimensions. As a result, 167938 documents were fetched.


Query Performance Summary

| | |
|---|---|
| Documents Returned: 167938 | Actual Query Execution Time (ms): 2663 |
| Index Keys Examined: 1175566 | Sorted in Memory: no |
| Documents Examined: 1175566 | Query used the following index: tt_start tt_end |

Figure 34. Bitemporal slicing statistics

To visualize this, we can see clear index hit on bitemporal dimensions in "Fielding" collection.

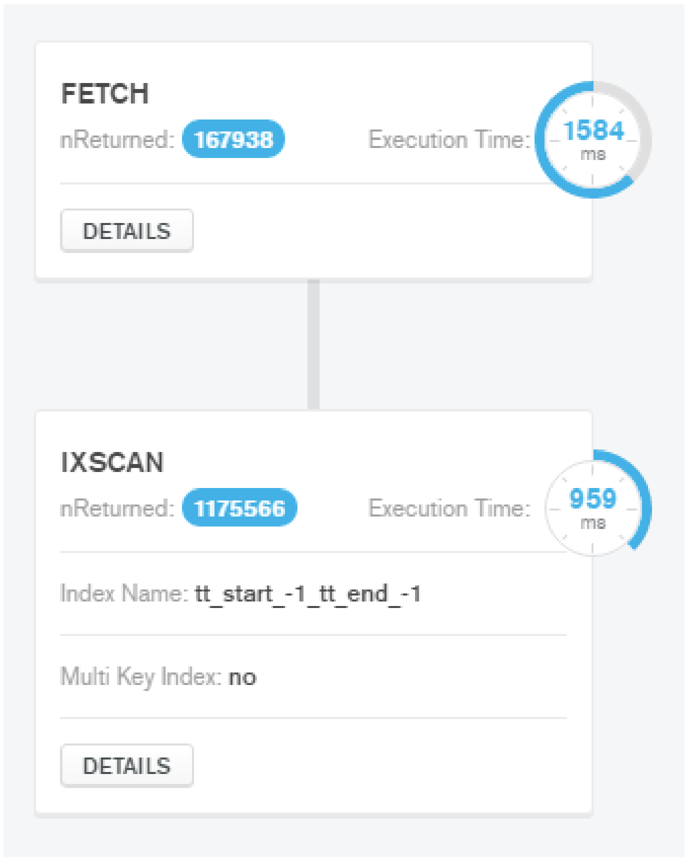
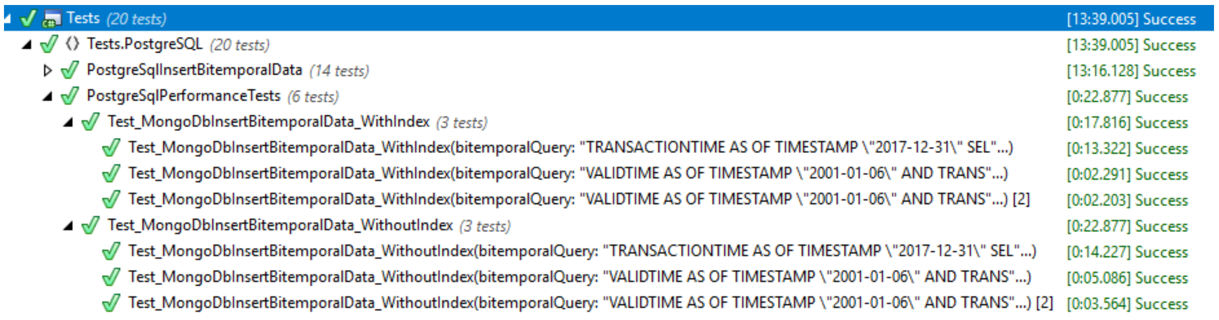


Figure 35. Bitemporal slicing performance

However, bitemporal slice was one of slower MongoDB queries in these test. Figure 36 shows that index scan only took 959 milliseconds. But fetching data completed only after 1584 milliseconds.

### 7.16.6 Slicing in PostgreSQL

For the sake fo full comparison PostgreSQL slicing was introduced and measured. Refer to figure 36 for more detailed results.



```
◢ ✓ Tests (20 tests)                                                                                                   [13:39.005] Success
  ◢ ✓ Tests.PostgreSQL (20 tests)                                                                                      [13:39.005] Success
    ▷ ✓ PostgreSqlInsertBitemporalData (14 tests)                                                                      [13:16.128] Success
    ◢ ✓ PostgreSqlPerformanceTests (6 tests)                                                                           [0:22.877] Success
      ◢ ✓ Test_MongoDbInsertBitemporalData_WithIndex (3 tests)                                                         [0:17.816] Success
          ✓ Test_MongoDbInsertBitemporalData_WithIndex(bitemporalQuery: "TRANSACTIONTIME AS OF TIMESTAMP \"2017-12-31\" SEL"...)      [0:13.322] Success
          ✓ Test_MongoDbInsertBitemporalData_WithIndex(bitemporalQuery: "VALIDTIME AS OF TIMESTAMP \"2001-01-06\" AND TRANS"...)      [0:02.291] Success
          ✓ Test_MongoDbInsertBitemporalData_WithIndex(bitemporalQuery: "VALIDTIME AS OF TIMESTAMP \"2001-01-06\" AND TRANS"...) [2]  [0:02.203] Success
      ◢ ✓ Test_MongoDbInsertBitemporalData_WithoutIndex (3 tests)                                                      [0:22.877] Success
          ✓ Test_MongoDbInsertBitemporalData_WithoutIndex(bitemporalQuery: "TRANSACTIONTIME AS OF TIMESTAMP \"2017-12-31\" SEL"...)   [0:14.227] Success
          ✓ Test_MongoDbInsertBitemporalData_WithoutIndex(bitemporalQuery: "VALIDTIME AS OF TIMESTAMP \"2001-01-06\" AND TRANS"...)   [0:05.086] Success
          ✓ Test_MongoDbInsertBitemporalData_WithoutIndex(bitemporalQuery: "VALIDTIME AS OF TIMESTAMP \"2001-01-06\" AND TRANS"...) [2] [0:03.564] Success
```

Figure 36. Result of PostgreSQL bitemporal queries

As previous results have shown, it quite expected index queries would perform much more faster. Unsurprisingly, it is the case for PostgreSQL (see figure 36). However, PostgreSQL performed slower than NoSQL outlined in the previous sections.

### 7.16.7 Data slicing summary

Various bitemporal data slicing is listed in the table 22 below. Table is sorted by slice type, index strategy and elapsed time in ascending order. Better performing system is in bold.

| Name | Operation | Slice type | Indexed | Time elapsed (s) | Memory usage | Processor usage | Disk usage |
|---:|:---|---:|---:|---:|---:|---:|---:|
| **MongoDB** | SELECT | Vertical | Yes | 1.102 | NA | NA | NA |
| PostgreSQL | SELECT | Vertical | Yes | 13.322 | NA | NA | NA |
| **MongoDB** | SELECT | Horizontal | Yes | 0.16 | NA | NA | NA |
| PostgreSQL | SELECT | Horizontal | Yes | 2.291 | NA | NA | NA |
| **PostgreSQL** | SELECT | Bitemporal | Yes | 2.203 | NA | NA | NA |
| MongoDB | SELECT | Bitemporal | Yes | 2.54 | NA | NA | NA |
| **MongoDB** | SELECT | Vertical | No | 2.102 | NA | NA | NA |
| PostgreSQL | SELECT | Vertical | No | 14.227 | NA | NA | NA |
| **MongoDB** | SELECT | Horizontal | No | 0.16 | NA | NA | NA |
| PostgreSQL | SELECT | Horizontal | No | 5.086 | NA | NA | NA |
| **PostgreSQL** | SELECT | Bitemporal | No | 3.564 | NA | NA | NA |
| MongoDB | SELECT | Bitemporal | No | 3.887 | NA | NA | NA |

Table 22. Summary of slicing performance for MongoDB and PostgreSQL

Here we can observe that MongoDB bitemporal data slicing almost always beats relational database. In similar environmental conditions, MongoDB on total is almost four times faster than PostgreSQL. However, in some cases relational database is faster. For instance, in case of bitemporal slicing both of them score a similar mark. This can be positively identified, because bitemporal slice is one of most used operation in bitemporal databases. On contrary, in this thesis we outlined *JOINless* queries. If we take bitemporal JOIN into account, relational database performance can deteriorate with every single table joined, while NoSQL will not have the same issue.

# Conclusions and Recommendations

As a result of this thesis, all core aspects of storing bitemporal data were analyzed. Leading types of non-relational data stores were mentioned in detail. Followed by pointing out benefits of having schema-less and horizontally scaled architecture to archive better performance and availability. In addition to this, thesis model was presented. It helped to archive end-goal of final work - comparing relational performance to non-relational for bitemporal data operations. This paper only covered few database type implementations - relational, key-value store and document-oriented systems. Bitemporal operations, such as insertion, change, deletion and slicing were implemented on multiple database management systems. A benchmark test suites bitemporal operations capabilities for different NoSQL/SQL engines, revealing that non-relational databases less time to execute. In this last chapter, the most important results of this thesis were summarized and the topics worth investigating further are discussed.

Another contribution of this thesis is an investigation of different approaches for achieving bitemporal support including existing database systems. As for the future of this topic, a lot more metrics have to be taken into account. In addition to that, it would be beneficial to run these performance test on a big scale. Even though, in this paper all tests were contained within the single machine, a scale-out test would make valuable addition to the research.

Nonetheless, making even more diverse system bitemporal NoSQL database system analysis would add some value as well. This research tried to cover bitemporal operations in PostgreSQL and MongoDB in great detail. However, taking into account diverse variety of NoSQL variety, this was out of the scope. Moreover, slicing operations could be extended - adding support of more bitemporal predicates.

After testing performance of various databases, few conclusions can be drawn. First of all, most of bitemporal data manipulation operations are faster in NoSQL databases. With exception to bitemporal deletion, where PostgreSQL yielded better results. Next, MongoDB outstandingly performed with bitemporal data insertion. Document system is three times faster than relation database. Speaking of computer resources, both of databases took significant amount of CPU power and IO for bitemporal change. All-in-all, judging from various performance metrics - time, CPU load, indexing, it can be said that NoSQL does better job doing executing bitemporal operations.

## References

[1] ISO/IEC JTC 1/SC 32. ISO/IEC 9075-1:2016 describes the conceptual framework used in other parts of ISO/IEC 9075 to specify the grammar of SQL and the result of processing statements in that language by an SQL-implementation. Standard, International Organization for Standardization, Geneva, CH, December 2016.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[3] Apache. Apache CouchDB is open source database software that focuses on ease of use and having a scalable architecture. http://couchdb.apache.org/, 2017. [Online; accessed 2017-09-15].

[4] Shay Banon. Elasticsearch is a search engine based on Lucene. https://www.elastic.co/products/elasticsearch, 2017. [Online; accessed 2017-10-20].

[5] Oracle Corporation. Oracle NoSQL Database is a NoSQL-type distributed key-value database. http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html, 2017. [Online; accessed 2017-12-10].

[6] Lahman's Baseball Database. Dataset used in bitemporal performance testing. http://seanlahman.com/baseball-archive/statistics/, 2017. [Online; accessed 2017-09-03].

[7] DataStax Enterprise. Article on NoSQL new systems. https://academy.datastax.com/planet-cassandra/what-is-nosql, 2017. [Online; accessed 2017-05-30].

[8] Apache Software Foundation. Apache Cassandra is a free and open-source distributed NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. http://cassandra.apache.org/, 2017. [Online; accessed 2017-03-10].

[9] Martin Fowler. ThoughtWorks on NoSQL Databases Explained in online article. http://martinfowler.com/articles/schemaless/, 2017. [Online; accessed 2017-04-18].

[10] Nancy Ann Gilbert, Seth; Lynch. Perspectives on the CAP Theorem. *Institute of Electrical and Electronics Engineers*, 2012.

[11] Sergey Gorbenko. SQL Parser. https://www.codeproject.com/Articles/32524/SQL-Parser, 2017. [Online; accessed 2017-10-01].

[12] PostgreSQL Global Development Group. Postgres, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance. https://www.postgresql.org/, 2017. [Online; accessed 2017-10-01].

[13] Mostafa Abd-El-Barr Hesham El-Rewini. Advanced Computer Architecture and Parallel Processing, 2005.

[14] IBM. The Four V's of Big Data. http://www.ibmbigdatahub.com/infographic/four-vs-big-data, 2017. [Online; accessed 2017-05-11].

[15] Franz Inc. AllegroGraph is a closed source triplestore which is designed to store RDF triples. https://franz.com/agraph/allegrograph/, 2017. [Online; accessed 2017-02-02].

[16] Google Inc. Bigtable is a compressed, high performance, and proprietary data storage system. https://cloud.google.com/bigtable/, 2017. [Online; accessed 2017-12-01].

[17] MongoDB Inc. MongoDB is a free and open-source cross-platform document-oriented database program. https://www.mongodb.com/, 2017. [Online; accessed 2017-05-15].

[18] MongoDB Inc. NoSQL Databases Explained in online article. https://www.mongodb.com/nosql-explained, 2017. [Online; accessed 2017-05-03].

[19] Danga Interactive. Memcached is a general-purpose distributed memory caching system. https://www.memcached.org/, 2017. [Online; accessed 2017-04-10].

[20] Christian S. Jensen and Richard T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[21] OrientDB Ltd. OrientDB is an open source NoSQL database management system. http://orientdb.com/, 2017. [Online; accessed 2017-02-20].

[22] Newtonsoft. Popular high-performance JSON framework for .NET. http://www.newtonsoft.com/json, 2017. [Online; accessed 2017-09-03].

[23] Inc. Objectivity. InfiniteGraph is an enterprise distributed graph database. http://www.objectivity.com/products/infinitegraph/, 2017. [Online; accessed 2017-01-08].

[24] Salvatore Sanfilippo. Redis is an open-source in-memory database project implementing a distributed, in-memory key-value store with optional durability. https://redis.io/, 2017. [Online; accessed 2017-04-11].

[25] Richard Snodgrass and Ilsoo Ahn. A taxonomy of time databases. *SIGMOD Rec.*, 14(4):236–246, May 1985.

[26] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[27] Richard T. Snodgrass. TSQL2 tutorial. In *The TSQL2 Temporal Query Language*, pages 31–46. Springer, 1995.

[28] Richard Thomas Snodgrass. Temporal databases. *IEEE Computer*, 1986.

[29] Richard Thomas Snodgrass. *Developing Time-oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[30] StackExchange. Dapper - a simple object mapper for .Net. https://github.com/StackExchange/Dapper, 2017. [Online; accessed 2017-11-02].

[31] StackExchange. StackExchange.Redis is a high performance general purpose redis client for .NET languages. https://github.com/StackExchange/StackExchange.Redis, 2017. [Online; accessed 2017-02-14].

[32] Daniel Wertheim. The asynchronous CouchDB client for .NET. https://github.com/danielwertheim/mycouch, 2017. [Online; accessed 2017-10-12].

[33] Nathan Young. xUnit.net is a free, open source, community-focused unit testing tool for the .NET Framework. https://xunit.github.io/, 2017. [Online; accessed 2017-09-20].