A HIERARCHICAL TIME-INDEXED DATABASE FOR MULTI-MODAL DERIVED SENSOR DATA USING GPU-ACCELERATED POSTGRESQL

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

JAMAL SAIED-WALKER

Dr. Grant J. Scott, Thesis Supervisor

DECEMBER 2023

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

A HIERARCHICAL TIME-INDEXED DATABASE FOR MULTI-MODAL DERIVED SENSOR DATA USING GPU-ACCELERATED POSTGRESQL

presented by Jamal Saied-Walker,

a candidate for the degree of Master of Science and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Grant J. Scott

Dr. Alex Hurt

Dr. Yaw Adu-Gyamfi

ACKNOWLEDGMENTS

I want to sincerely thank Dr. Grant J. Scott, my advisor, for his support and direction during my research. Over my master's program, he continuously gave me the resources I needed to complete my research.

Besides my advisor, I would like to thank Dr. Yaw Adu-Gyamfi for his invaluable support during my early years at Mizzou, and Pallavi Gupta for helping me navigate our lab's research.

Finally, I am extremely grateful to my family, especially my parents, Diana Walker and Abraham Saied, for their support. My godmother, Olga Walker, who has left an indelible mark on my life and has been a guiding force. My cousin, Vasco Duke, who has been an incredible mentor during my journey of graduate school. I am also thankful to Maryann Castillo, my partner, whose emotional support has been fundamental during this process of earning my master's degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the vast influx of data captured by ubiquitous sensors, such as those in wearable devices, smart sensors in long-term care facilities, and hydraulic bed sensors, there exists an enormous potential for exploiting these multi-modal data using advanced analytics. However, the ability to effectively utilize these growing datasets is contingent upon high-performance data-handling systems that ensure high data scalability and effortless data accessibility, especially in various fields demanding real-time signal processing like healthcare, artificial intelligence, machine learning, and scientific research.

In this context, this thesis aims to address the challenges associated with the immense volume of sensor data by proposing an efficient database design system. This system utilizes general purpose, high-scalability database systems and integrates them with data analytics focused column stores that exploit hierarchical time indexing, compression, and dense raw numeric data storage. The approach further includes leveraging the capabilities of graphics processing units (GPUs) in conjunction with database management systems (DBMS) using the server programming interface (SPI) with PostgreSQL for accelerated signal processing.

We demonstrate robust design, developments, and techniques of a hierarchical time-indexed database for decision support systems and propose an innovative database system for multi-modal derived time-series featured data (e.g., *respiration*, *restlessness*, and *heart rate*). We also introduce a data-processing pipeline to enable Big Data analytics for multi-modal time-series feature data and compare the performance of CPU and GPU approaches for feature extraction from sensor data.

Our evaluations reveal the performance characteristics and tradeoffs of each component, with special emphasis on data access latencies and storage requirements - vital elements for capacity planning in scalable systems. The proposed database system is demonstrated to be extremely scalable and offers straightforward integration with existing analytic tools via SQL interfaces. Furthermore, we discuss the usability, adaptability, timing metrics, and precision differences between CPU and GPU code in different thread and block configurations, thereby offering a comprehensive view of real-time signal processing systems.

Chapter 1

Introduction

1.1 Background and Related Work

The continuous and increasing collection of data and advancements in healthcare has led to significant efforts to develop scalable and high-performing database management systems (DBMS) for smart sensor environments. DBMS provides a structured and efficient platform to store, manage, and retrieve large amounts of data. This is crucial for long-term care facilities, where there is a need to access time-series data in arbitrary time windows. The ability to access arbitrary sliced time data is especially important for time-series analysis as a monitoring tool and prognosis analytics in long-term care facilities [1]. The use of smart sensors helps track the health deterioration of chronic diseases. Cardiovascular, respiratory, and brain diseases are among the chronic diseases that can be tracked using smart sensors like hydraulic bed sensors [2, 3, 4, 5]. The hydraulic bed sensor data is particularly suited for scalable systems due to the high-frequency nature of the data [6, 7].

While traditional data storage systems such as SQL and NoSQL can accommodate large amounts of raw and computed data, their basic structure limits their efficiency of access for algorithms and advanced analytics. Time-Series Databases (TSDBs) are another traditional solution for time-series data that link sets of time with values. While generally compatible with Internet-of-Things (IoT) devices, the unique characteristics of non-wearable sensors such as hydraulic bed sensors make TSDBs less suitable for integrating this type of data [8, 9, 10].

Moreover, the use of advanced algorithms to extract information from the hydraulic bed sensor signal for heart rate, respiration, and restlessness is crucial for identifying and studying diseases. The ability to perform multi-modal, longitudinal, and predictive health analysis is necessary as health conditions can span weeks to years or even a lifetime. Health tracking is crucial for residents in long-term health facilities, where sedentary behavior has increased in older adults. Smart sensors can provide useful information to help caregivers or researchers identify earlier stages of a disease, enhancing their ability to react to a health episode of the residents [11, 12, 13, 14].

Because of this, High-Performance Computing (HPC) has become increasingly important and much research effort for analyzing and deriving physiological data [15]. Utilizing HPC methods can accelerate the computation of signal-processing algorithms. Graphic Processing Units (GPUs) have become increasingly dominant in many scientific research applications, including artificial intelligence, machine learning, and data mining [16, 17, 18, 19]. Additionally, there has been an increase in the popularity of GPUs in large-scale data center applications. GPUs' increased popularity in many fields has enriched the research for GPU-accelerated DBMS [20, 21, 22].

The integration of GPUs with DBMSs can improve the processing of large amounts of data [23], and as mentioned, GPUs have been gaining the focus of attention for many scientific computing tasks. An example of a GPU-accelerated database extension is presented in [24], where they designed a GPU framework for high-throughput large-scale pattern matching as a PostgreSQL extension [25]. The authors of [26] presented different algorithms for performing common SQL operations such as aggregations, relational queries, and conjunctions on GPUs inside the database. As part of their contribution, they highlight that some of the algorithms performed better than others; demonstrating the important point that not every problem or algorithm is going to benefit from GPU acceleration within databases.

1.2 Contribution

In this thesis, we present a comprehensive solution to address the challenges in handling time-series Big Data by proposing a hierarchical time-indexed database (HTIDB) design for scalable time-series data warehousing. This database design incorporates a hierarchical time-index (HTI) structure, irregular sampling, irregular timespans, and native numeric data in a segmented column-storage layout. The storage of data in compressed dense numerical matrices not only saves space but also enhances scalability. This approach facilitates efficient data access for analytics, algorithms, and the integration of other data types such as relational records and sensor data.

Our proposed data processing pipeline for processing multi-modal feature time-series Big Data effectively supports Big Data analytics and allows researchers to utilize signal-processing algorithms for extracting physiological features from bed sensor data (i.e *restlessness*, *respiration*, and *heart rate*). We demonstrate a novel approach to developing signal-processing algorithms, enabling seamless integration into GPU-accelerated PostgreSQL. This approach is applicable to any algorithm that can benefit from GPU-accelerated databases.

Lastly, we provide an overview of our solution to further foster a better understanding of the design concepts and implementation of the data processing pipeline of multi-modal feature data from hydraulic bed sensors and GPU-accelerated databases. To facilitate ease of implementation, we offer an open-source repository with examples of different GPU kernels, highlighting how to create PostgreSQL extensions using Server Programmable Interface (SPI) in C++ and performing the computations on the GPU. These contributions aim to provide a scalable and efficient solution for handling time-series Big Data in various research and industry applications.

1.3 Outline

This thesis addresses the challenges in handling time-series Big Data by proposing a hierarchical time-indexed database (HTIDB) design for scalable time-series data warehousing, a data processing pipeline for deriving multi-modal feature data from hydraulic bed sensors, and a comprehensive guide on how to create GPU-accelerated databases for performing database operations on the GPU. Chapter 2 introduces the concepts of time-series databases, hierarchical time-indexed databases (HTIDB), the current layout system structure for long-term healthcare facilities, data storage compression methods such as segmented storage and array compression, and the benefits of a DBMS with HTIDB design in healthcare.

Chapter 3 introduces the data processing pipeline and Big Data management for the derivation of multi-modal features from hydraulic bed sensors. Chapter 3 also details how to store such derived features, and storage characteristics, as well as how to construct the data for Machine Learning (ML) applications.

Chapter 4 introduces and delves into how to create database extensions in PostgreSQL with SPIs and GPUs, core SPI concepts, and how to create kernels and applications within the GPU using Compute Unified Device Architecture (CUDA). Furthermore, Chapter 4 details how to design and integrate SPI and CUDA to create PostgreSQL extensions that run computations on the GPU.

Chapter 5 introduces health applications, timing experiments for accessing raw data, multi-modal derived feature table access times, CPU and GPU timing comparisons for *heart rate* estimation from bed sensors, and different thread and block configurations for the *heart rate* kernel. Furthermore, Chapter 5 highlights the results and provides an analysis. Lastly, Chapter 6 explains future directions, summarizes the work, and highlights the accomplishments of this thesis.

Chapter 2

Hierarchical Time Indexed Databases Designs and Implementations

The need to access time-series data in arbitrary time windows has become increasingly important in long-term care facilities, as they continue to adopt more sensor networks and technologies. Arbitrary sliced time data offers an easily accessible mechanism for time-series analysis, which can be employed as a monitoring tool and for prognosis analytics in these facilities. This is particularly relevant when identifying associations between sensor data streams and medical events documented in electronic health records (EHRs), as well as human activity captured by other ambient sensing modalities. Consequently, this chapter aims to highlight the necessity and propose a solution for a DBMS in PostgreSQL design, tailored to accommodate the needs of irregular time-series data, such as hydraulic bed sensor data.

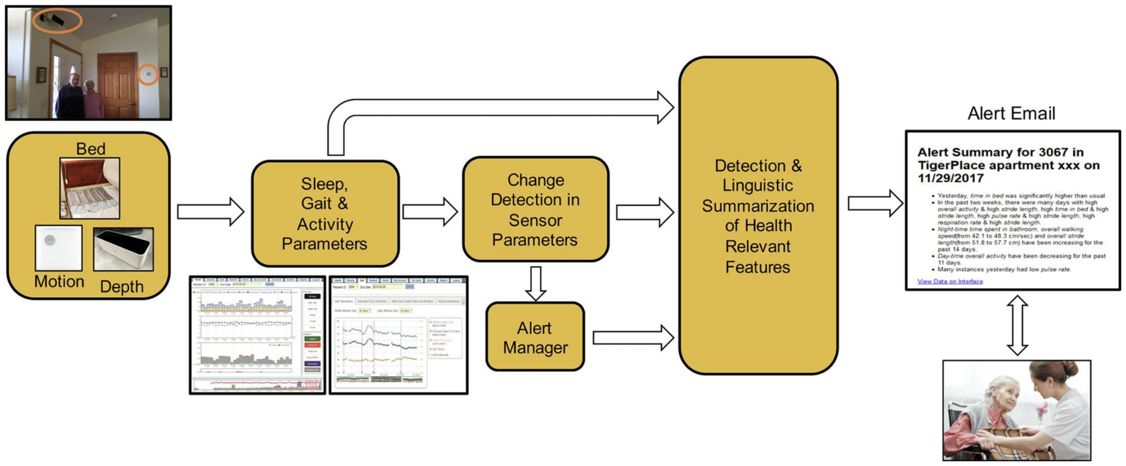
2.1 Sensor Data

The data source used in the entire thesis is from a hydraulic bed sensor that collected data at long-term facilities for older adults. Center for Eldercare and Rehabilitation Technology (CERT) works together with Americare aging-in-place facilities, such as TigerPlace, to advance the creation of innovative smart healthcare technologies designed for elderly care [27]. TigerPlace is an aging-in-place facility for older adults where numerous smart sensors are in place for passive older adult resident monitoring established in 2005 at the University of Missouri.

As shown in Fig. 2.1, TigerPlace consists of a sensor network of both wearable and non-wearable sensors. Figure 2.1 shows how the smart sensors are used for constant monitoring and detection of the residents' activity. Wearable sensors like smartwatches may be challenging for older adults who are unfamiliar with the device and require active management. Therefore, non-wearable sensors like hydraulic bed sensors are preferred in long-term care facilities. Tracking residents' health information via a hydraulic bed sensor is challenging for researchers for two reasons: the amount of continuous data collected per day per person, and the way data produced from hydraulic bed sensors is stored.

The hydraulic bed sensor is a collection of four transducers (flexible water-filled tubes) that are positioned under the resident mattress as seen in Fig. 2.2 [29]. The hydraulic sensors are pressure sensitive to capture bio-physiological changes in bed. These changes in pressure translate into a change in the voltage signal. Moreover, the voltage signals undergo hardware filtering. Hence, for every captured timestamp, each one of the four transducers generates one raw reading (denoted with the letter r) and one filtered reading (denoted with the letter f).

The purpose of using four dispersed transducers is to cover the largest possible surface area on the bed for effective data gathering. As a result, typically at any given moment, only one of the four transducers will have a maximum reading, unless


Bed
Motion
Depth
Sleep, Gait & Activity Parameters
Change Detection in Sensor Parameters
Alert Manager
Detection & Linguistic Summarization of Health Relevant Features
Alert Email

Alert Summary for 3067 in TigerPlace apartment xxx on 11/29/2017

- Yesterday, *time in bed* was significantly higher than usual.
- In the past two weeks, there were many days with high *overall activity* & high *stride length*, high *time in bed* & high *stride length*, high *pulse rate* & high *stride length*, high *respiration rate* & high *stride length*.
- *Night-time time spent in bathroom*, *overall walking speed*(from 42.1 to 48.3 cm/sec) and *overall stride length*(from 51.8 to 57.7 cm) have been increasing for the past 14 days.
- *Day-time overall activity* have been decreasing for the past 11 days.
- Many instances yesterday had low *pulse rate*.

View Data on Interface

Figure 2.1: Sensor Network consists of wearable and non-wearable sensors (bed, gait, motion activity) [28]

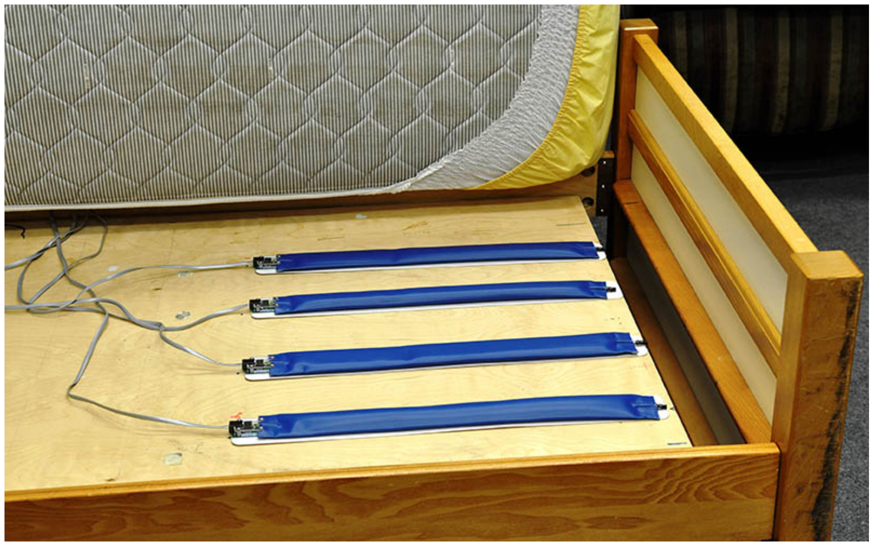


Figure 2.2: Hydraulic bed sensors monitor cardiovascular activity during sleep. Pressure-sensitive hydraulic transducers capture bio-physiological changes in bed

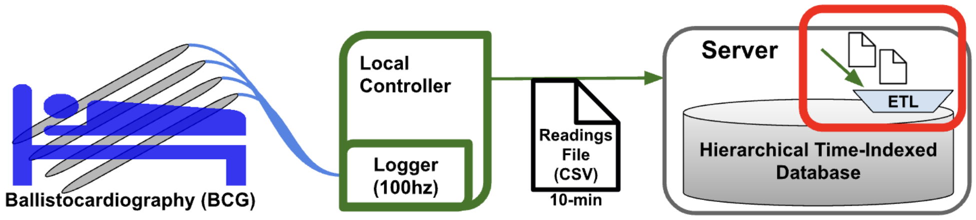


Figure 2.3: Sensor data logger at TigerPlace collects transducer data in 10-minute intervals and saves as compressed CSV files.

the resident shifts their position or turns to their side on the bed, causing the load to transfer to another transducer.

The current data pipeline for storing hydraulic bed sensor data from the long-term health facility, TigerPlace, is using GZIP (compressed) comma separated value (CSV) files as shown in Fig. 2.3. Figure 2.3 is a visual representation of how the data goes from the hydraulic bed sensor to the server. The hydraulic bed sensor takes readings at 100 Hz measurements, saving the eight features every 0.01 s, and saving the data in chunks of ten-minute intervals in the compressed GZIP formats.

Aside from having a good compression ratio, saving data in this flat file storage format is inefficient for researchers to conduct later analysis and any subsequent data processing. Furthermore, since the local controller stores the file directly on the server, each smart sensor has a directory in the file system associated with a resident. This hierarchical file structure of CSV files is depicted in Fig. 2.4.

Figure 2.4 demonstrates that for each resident, there is a folder containing multiple sensor folders. These sensor folders are then organized into subsequent folders by year, month, and day, with the GZIP files located beneath them. This means that a record taken at `2017-09-17 21:55:15` would reside in `/<resident_id>/beddata/2017/09/17/2017_09_17_21_50_44.bd3.gz`.

As we can see, analyzing a specific temporal window of hydraulic bed sensor data requires researchers to undertake several inefficient steps. Firstly, they must

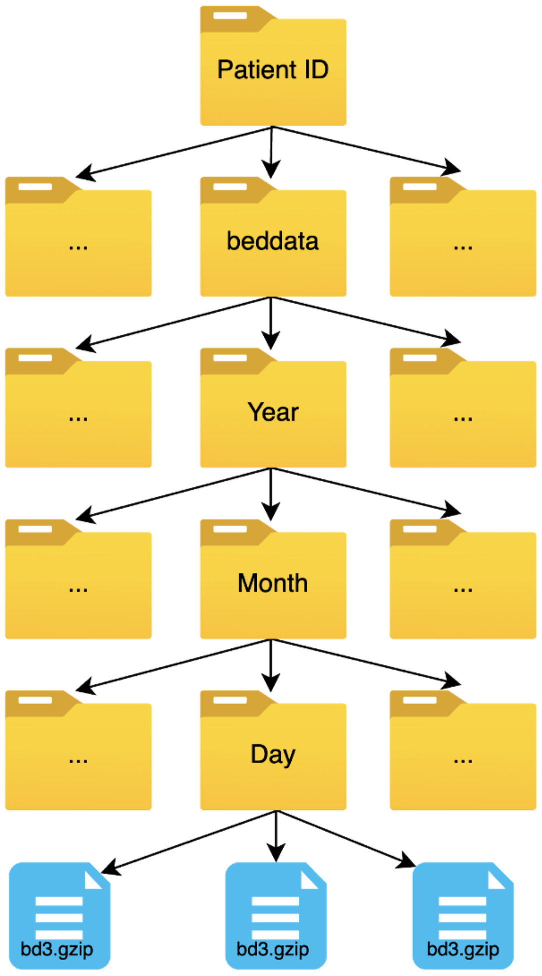


Figure 2.4: Hierarchical file structure of CSV files.

Table 2.1: Example of the contents of each GPZIP file.

tstamp	r1	r2	r3	r4	f1	f2	f3	f4
08-14 10:58:41.795	24	619	31	688	2027	2028	2042	2062
08-14 10:58:41.805	24	619	30	688	2030	2042	2042	2050
08-14 10:58:41.815	24	620	32	688	2032	2037	2044	2049
08-14 10:58:41.885	24	624	31	689	2033	2032	2045	2048
....	..	...	..	...	....	....	....	....
....	..	...	..	...	....	....	....	....
....	..	...	..	...	....	....	....	....
08-14 11:08:44.025	25	619	32	534	2036	2040	2041	2042
08-14 11:08:44.035	26	619	32	530	2033	2038	2045	2041
08-14 11:08:44.045	24	622	32	528	2043	2037	2041	2042
08-14 11:08:44.055	25	619	28	529	2034	2037	2048	2047

convert the temporal window into a path-like string to locate the relevant directories containing the data. Secondly, they must unzip each individual GZIP file and convert it into a numerical representation. Finally, they need to manipulate timestamps for each row to determine the actual timestamp as shown in Table 2.1.

Table 2.1 illustrates the contents of a GZIP file. As can be seen from Table 2.1, the *tstamp* column does not contain valid timestamps. To convert the *tstamp* column into a valid format, the year must be appended to each *tstamp* entry. While this approach may be feasible for narrow temporal windows, it becomes impractical when analyzing data spanning multiple months or years.

2.2 Hierarchical Time-Indexed Database

2.2.1 Design

As mentioned in section 2.1, the current design requires inefficient steps for using the bed sensor data. The required characteristics of an improved data storage and access solution for bed sensors must support the following: a) native (binary) storage of the numerical data; b) indexing mechanisms for arbitrary access to temporal ranges of

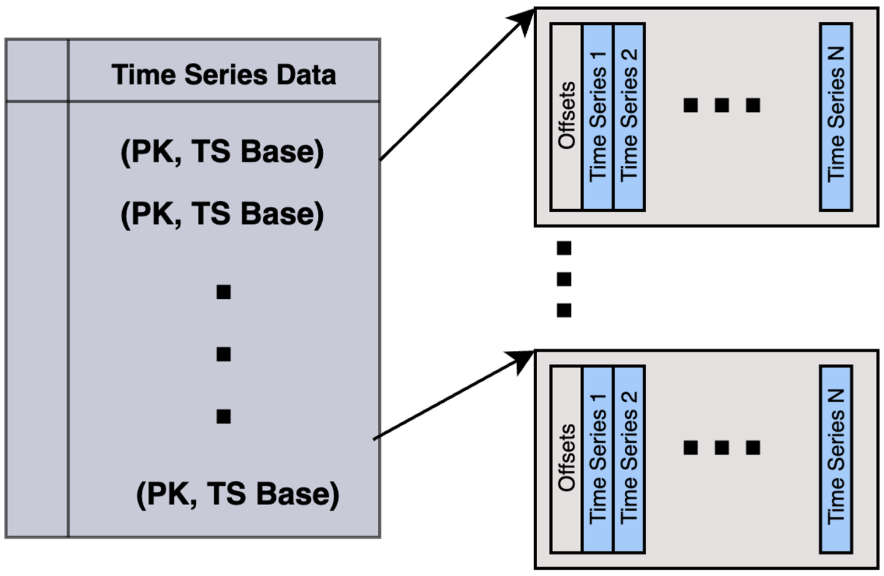


Figure 2.5: Hierarchical Time-Indexing uses blocks of records, with each block having a timestamp base, then each record block is a partial column store of dense numerical data, compressed and indexed by sub-second offsets.

the data; and c) the ability to integrate the data access with other modalities, such as relational data. With this in mind, we have developed a hierarchical time-indexed database (HTIDB) design for scalable time series data.

Figure 2.5 presents a diagram of the HTIDB design. The top-level record structure comprises an index based on non-time-series data, such as sensor ID (i.e., relational *Primary Key*), as well as a timestamp base value. This base value provides an absolute time reference for a time series column store segment with time indexing, measured in partial seconds relative to that base.

Each top-level record contains a column store segment with secondary time indexing associated with the timestamp base. While this secondary indexing introduces storage overhead, it eliminates the need for assumptions regarding measurement sampling rates and continuity, thereby enabling reliable access to irregular time-series data. The column store segments always maintain temporal order; however, the timestamp base does not necessarily follow this order, as they are indexed in both *(temporal order)* and *(Key, temporal order)* at the higher database layer.

Record (1.15 MB)	
Field	**Data Type (Size)**
Enabled Sensor ID	integer (4 B)
Timestamp Base	timestamp (8 B)
Timestamp Offset	float array (234.4 KB)
Bed Sensor Data	dense 2-D array (937.5 KB)
Indices	
Index	**Scope→Unit**
SensorID	Table→Partition
(SensorID, TimeStampBase)	Partition→Record
(Timestamp Offset)	Record→Dense Array

Figure 2.6: Bed sensor data table layout. The *Timestamp Base* is an index into 10-minute segments (records). Each record has 10-minute span column store of the bed sensor data, which is indexed in floating point seconds from the base. DB and column store segment indices accelerate analytics.

Figure 2.6 provides an overview of the record size and breakdown within the HTIDB used for high-frequency bed data. As noted, a record consists of 60,000 time series measurements across eight features, which is packed into 1.15 MB. The majority of this is a) the timestamp offset array, floating point seconds, using 234.4 KB, and b) the 2-D dense numerical array (60,000×8) of measurements, using 937.5 KB. However, these two arrays are compressed in storage, allowing the total data use to be significantly smaller than 1.15 MB per record. The 2-D dense numerical array is organized as a column store segment, whereby retrieval and ordering by the records' timestamp base allows the concatenation of arbitrary-sized time series data.

Finally, one of the key important aspects of the HTIDB is that it is integrated into a traditional DBMS. In this way, the *Key* referenced in Fig. 2.5, or the *Enabled Sensor ID* in Fig. 2.6 example, can be incorporated into traditional join semantics of database queries.

Table 2.2 displays an exemplary layout, with a base time of August 5, 2019, at 2:19:20.210 AM. The first row in the dense numeric 2-D array is indexed at 0.0 seconds, followed by the second row at 0.015 seconds from the timestamp base, the third row at 0.025 seconds, and so on. It is important to note that there is no requirement

Table 2.2: Example of hierarchical raw high-frequency (100 Hz) bed data with internal arrays, set at timestamp (TS) base and temporal indexing offsets in seconds.

TS Base	Offset	Bed Sensor Data (tensors)	
2019-08-05	0.000	27, 816, 24, 994	2031, 2024, 2031, 2041
02:19:28.210	0.015	30, 822, 23, 994	2032, 2034, 2030, 2040
	0.025	30, 819, 24, 999	2030, 2027, 2026, 2037
	0.035	30, 822, 22,1000	2042, 2034, 2027, 2047
	0.045	28, 820, 22, 995	2038, 2018, 2030, 2048
	0.055	28, 818, 21, 999	2040, 2027, 2037, 2045
	0.065	28, 815, 24, 996	2041, 2026, 2026, 2044
	0.075	27, 816, 24, 995	2045, 2013, 2029, 2044
	0.085	30, 818, 24, 993	2044, 2021, 2030, 2034
	0.095	28, 823, 24, 996	2035, 2020, 2026, 2036

Table 2.3: Bed Data Storage Characteristics for one resident across two years.

Schema	**Size**	**Index Size**	**Number Rows**
RAW BD3	8.9 GB	-	182,822,164
GZIP	1.9 GB		4098 (files)
PostgreSQL	16.0 GB	5.5 GB	182,822,164
HTIDB	2.4 GB	144 KB	4098 (recs)

for measurement sampling intervals to be uniform within the *offset indexing* of the column store segment. In the tested implementation, the column store segments consist of 60,000 time series measurements (rows), which is approximately 10 minutes of 100 Hz bed data. For the sake of brevity, we have truncated the data in Table 2.2.

2.2.2 Space Utilization & Characteristics

Table 2.3 shows the characteristic disk utilization for 182 million measurements across the eight features, which are agglomerated from 4098 10-minute bed data files of 100 Hz readings. Most notable are the various trade-offs between the storage paradigms. The raw data in CSV format is nearly 9 GB but requires parsing from row-based text into numerical columns each time data is loaded. Furthermore, to process data spanning multiple 10-minute data files or captures, file concatenation either during or after parsing into memory structures is required.

Table 2.4: Bed Data Storage Characteristics

Residents	Nights	Hours	Readings	Storage (GB)
27	9568	205,655	32.7 Billion	632.17

The GZIP CSV files generated from the sensor system are the most space-efficient but the least efficient for subsequent analysis and processing. Data extraction and processing require decompression, parsing, and concatenation. When the data is staged into a relational database table, the space utilization amounts to 16 GB, but it is easy to access and perform aggregations.

In this scenario, the data is structured into numerical storage; however, row-based access for high-frequency time series is sub-optimal, especially when not all features are needed, such as only the four filtered measures. Nevertheless, the proposed HTIDB (implemented in PostgreSQL) occupies only 2.4 GB and supports both simple access and aggregation analytics. Due to its smaller size, the HTIDB is significantly more efficient for access and analytics from a performance perspective, as discussed in the next subsection. The storage size of a record (60,000 rows in a column store segment) averages 0.6 MB. In the case of both the flat PostgreSQL and the HTIDB, standard DBMS indexing is applied. Although these indices create a space overhead, they drastically improve data access performance.

After loading the GZIP files from 27 residents, the PostgreSQL HTIDB displays the bed storage characteristics as presented in Table 2.4. Table 2.4 demonstrates the bed data storage features for all hydraulic bed sensor tables. Specifically, it surpasses 32 billion measurement points in the time-series and utilizes approximately 632 GB of storage in the HTIDB.

The hydraulic bed sensor tables continuously expand as more residents are added and as existing residents keep sleeping each night. The data collection period varies among residents, ranging from a few months to over two years, spanning from January 2017 to July 2022.

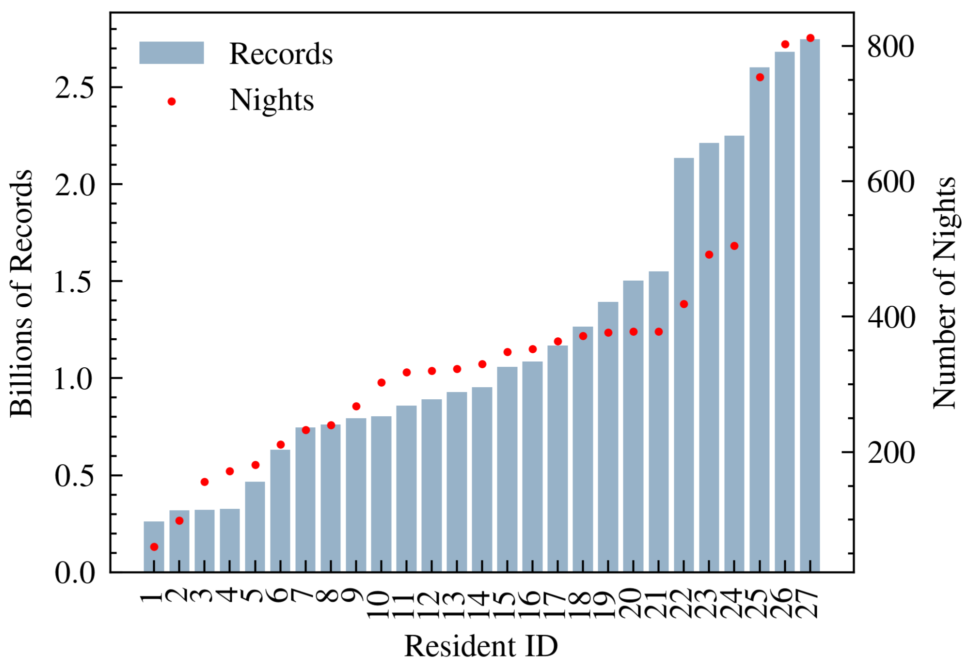


Figure 2.7: Bar chart displaying the number of data records (in billions) for each resident in the database. Each data record consists of four raw and four hardware-filtered measurements from the bed sensor system.

Figure 2.7 displays the number of records taken from each resident's hydraulic bed sensor tables. The red points on Fig. 2.7 show the number of nights of hydraulic bed sensor data for each resident, indicating a close correlation between the number of records and night period counts.

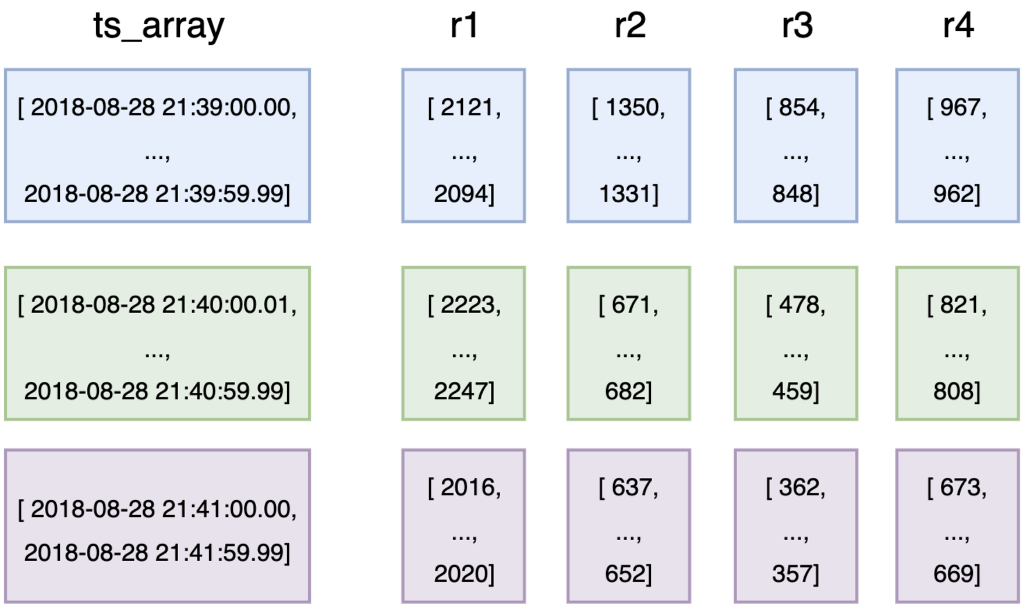
2.3 Summary

This chapter has provided an overview of the existing sensor network layout at TigerPlace. It has explained the concept of hydraulic bed sensors, their importance in research, and the challenges researchers currently face with the existing sensor pipeline. Moreover, this chapter has proposed a design for Hierarchical Time-Indexed Databases (HTIDB). The design supports irregular time sampling, queries, and analytics over arbitrary temporal windows, as well as high-performance linear growth in data storage requirements and query performance. Lastly, we have discussed the various space utilization and storage characteristics of the proposed system.

Chapter 3

Multi-modal derived features from bed sensor data

Hydraulic bed sensors have been instrumental in deriving physiological components such as *respiration*, *restlessness*, and *heart rate* that can be used for the early diagnosis of numerous chronic diseases, including respiratory and heart diseases [30]. For example, abnormal patterns in a patient's *respiration* rate derived from hydraulic bed sensors can be indicative of pulmonary diseases like Chronic Obstructive Pulmonary Disorder (COPD), respiratory diseases, or blood pressure-related diseases [31]. Moreover, early detection of heart failure in older adults can be achieved by monitoring ballistocardiogram (BCG) signals deconvolved from hydraulic bed sensor data [32] [33]. In light of the critical role that multi-modal derived feature data plays in healthcare, it is essential to have this information available for researchers. However, managing and storing such vast amounts of data presents a challenge. This thesis aims to propose a design for the multi-modal feature time-series Big Data management system tailored to hydraulic bed sensor data. Furthermore, the proposed design enables signal-processing algorithms to efficiently extract physiological features from bed sensor data.



ts_array	r1	r2	r3	r4
[2018-08-28 21:39:00.00, ..., 2018-08-28 21:39:59.99]	[2121, ..., 2094]	[1350, ..., 1331]	[854, ..., 848]	[967, ..., 962]
[2018-08-28 21:40:00.01, ..., 2018-08-28 21:40:59.99]	[2223, ..., 2247]	[671, ..., 682]	[478, ..., 459]	[821, ..., 808]
[2018-08-28 21:41:00.00, 2018-08-28 21:41:59.99]	[2016, ..., 2020]	[637, ..., 652]	[362, ..., 357]	[673, ..., 669]

Figure 3.1: Grouped data in 1-minute intervals retrieved from raw bed sensor table with all the raw readings for *restlessness* computation.

3.1 Derived Multi-Modal Feature Data

As mentioned before, from the hydraulic bed sensor data, various physiological signals are extracted that may be associated with resident health conditions. For example, the bed sensor signal is a combination of physical movement, breathing, and cardiovascular activity while the resident is sleeping. Depending on the extracted feature, (*respiration*, *restlessness*, or *heart rate*), different parts of the bed sensor data are extracted and processed for computing derived features from the bed sensor.

3.1.1 Restlessness

The *restlessness* feature is gathered by taking the raw readings from the bed sensor table discussed earlier, using the PostgreSQL procedure shown in Fig. 3.13. Figure 3.13 is a stored PostgreSQL procedure that takes any bed sensor table in a column-storage layout and converts it back into a traditional row layout, which is then ready for grouping. Once the data is in the row layout, it can be grouped into

1-minute intervals by using the stored procedure shown in Fig. 3.14. Figure 3.14 takes in the result from the stored procedure described earlier and truncates the *tstamp* to the minute using the *DATE_TRUNC* built-in PostgreSQL function. Each raw value is then aggregated back into arrays using the *ARRAY_AGG* built-in PostgreSQL function. The result of the procedure in Fig. 3.14 is shown in Fig. 3.1. As we can see from Fig. 3.1, each row represents an array of values for a given minute for all four raw readings from the bed sensor, as well as their corresponding *tstamp* values. Next, a sixth-order bandpass filter is taken and applied to all the raw bed sensor arrays. After applying the filter, a maximum value for every second is computed alongside the mean value, and then the motion values are assigned based on a constant motion threshold (1.5). This constant motion threshold was selected empirically to provide high enough sensitivity for movement detection. After the motion strength has been calculated for each minute interval, the computed matrix is used to extract the start times and end times of restless motion.

Figure 3.2 shows the extracted *restlessness* values for resident 54005 over one night. As we can see from Fig. 3.2, it shows the motion values started at around `22:27`, and the last recorded motion value was recorded at `06:27`.

3.1.2 Respiration

Features related to *respiration* are gathered using the method mentioned in Heise et al. [34]. It begins with the extraction of *respiration* signals from bed sensor data by applying a butter-worth low-pass filter with a cut-off frequency of (0.7 Hz). From *respiration* signals, peak points, valley points, and the associated time stamps are extracted. Before processing the data, the first step is to select the best, i.e., most responsive, transducer for feature extraction. The best transducer can be derived by computing the sum of each of the raw readings and then selecting the filtered reading that aligns to the raw reading signal (*r1* with *f1*, *r2* with *f2*, and so on). Once the

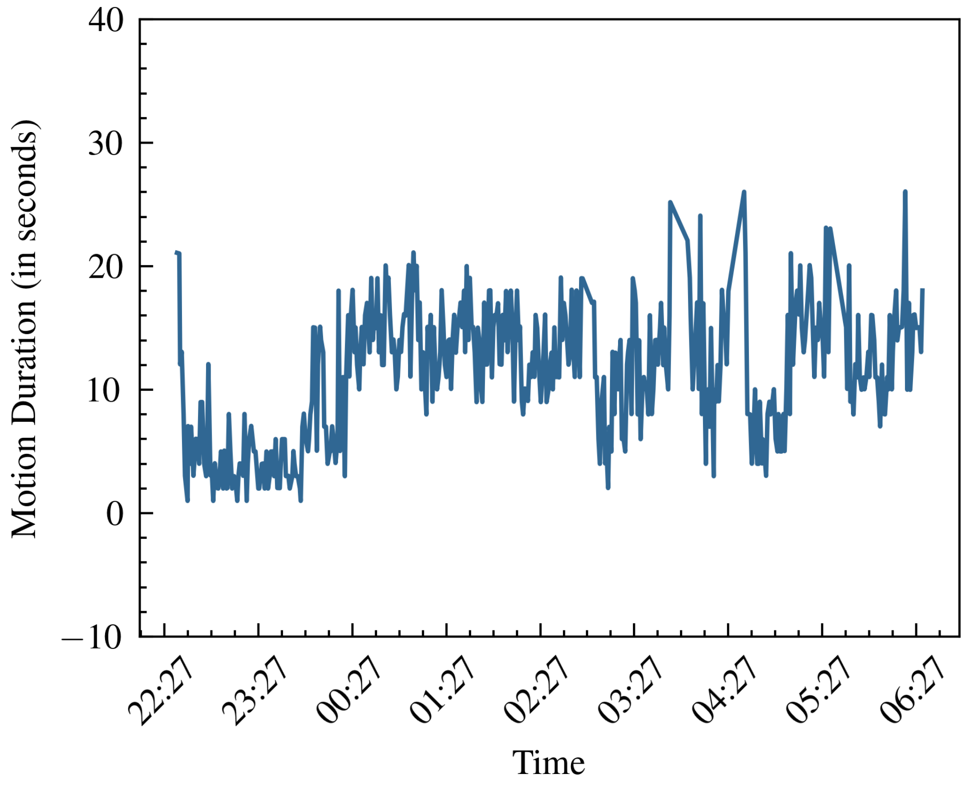


Figure 3.2: Example of extracted multi-modal feature time-series plot over one night. The plot shows the motion in seconds for the entire night.

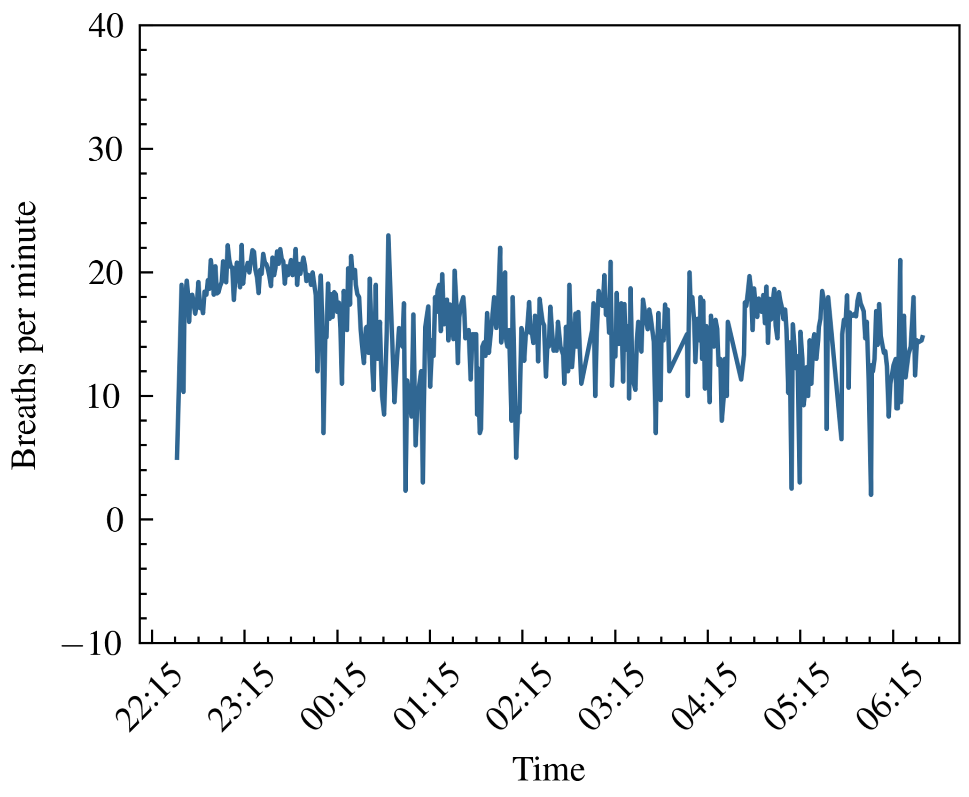


Figure 3.3: Example of extracted multi-modal feature time-series plot over one night. The plot shows the respiration rate for the entire night.

transducer filter reading has been selected and the butter-worth low-pass filter has been applied, the processed data is used to get peak points labeled as normal peak, noisy peak, apnea peak, or hypopnea peak. Noisy peaks are the *respiration* peaks that are outside of normal bounds (and thus likely caused by motion rather than *respiration*). We disregard the noisy peaks, apnea peaks, and hypopnea peaks. We consider only the normal peaks to calculate the *respiration* rate. Due to the computational complexity of the algorithm, the *respiration* cycles had to be partitioned into one-hour intervals for efficient processing. The *respiration* cycles are partitioned into one-hour intervals for efficient processing to accommodate computational complexities.

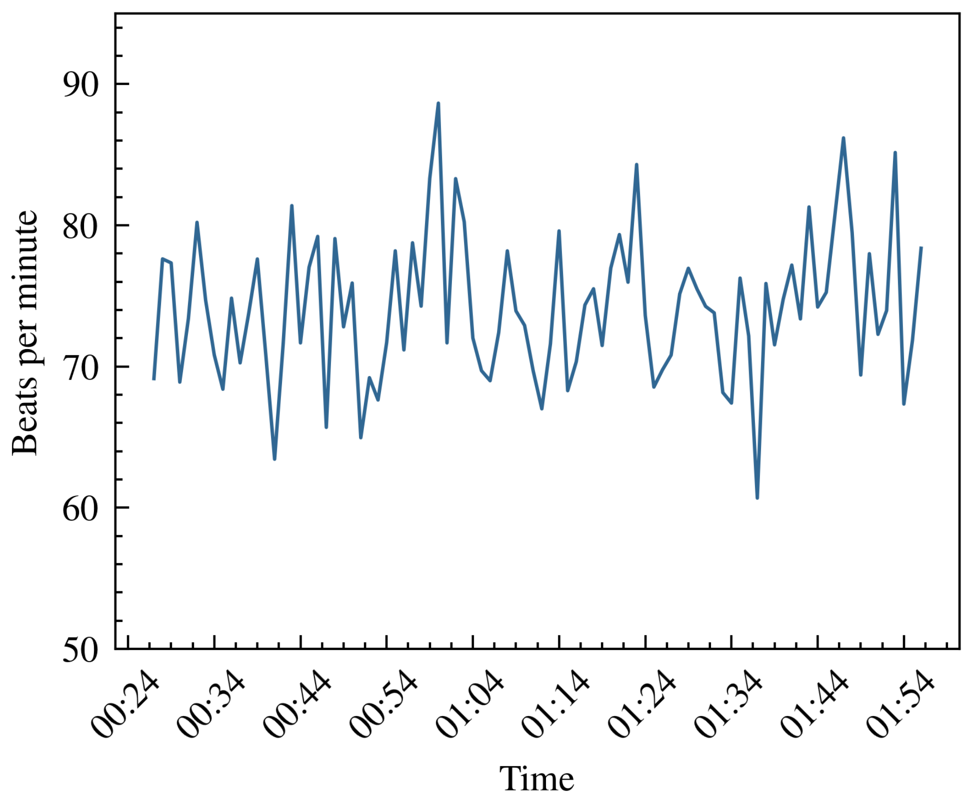


Figure 3.4: Example of extracted multi-modal feature time-series plot over one night. The plot shows the beats per minute (bps) for a two hour period.

3.1.3 Heart Rate

The cardiovascular feature (*heart rate*) component is computed using a short-time energy algorithm [35]. The algorithm first applies the Butterworth low-pass filter with an order of 6 to remove the *respiration* and high-frequency noise component from the transducer signals. The cutoff frequencies are set to 0.7 Hz to 10 Hz. Next, outlier removal is performed and the final estimate is then computed.

3.2 Data Processing Pipeline

Once the extracted feature data has been computed for every resident, the data-processing pipeline shown in Fig. 3.5 can be performed. Figure 3.5 shows the main

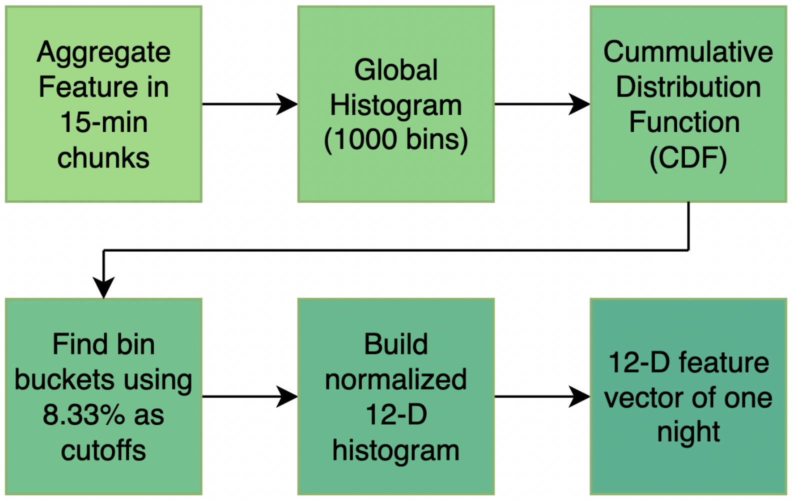

Aggregate Feature in 15-min chunks
Global Histogram (1000 bins)
Cummulative Distribution Function (CDF)
Find bin buckets using 8.33% as cutoffs
Build normalized 12-D histogram
12-D feature vector of one night

Figure 3.5: Data-processing pipeline for creating 12-dimensional feature vectors for each night on every resident.

steps for generating the vectors for each night per resident, per feature space (*respiration* and *restlessness*). This process is done identically for both extracted features. We generate a 12-dimensional feature vector for each resident for each night, and here the choice of 12 dimensions is arbitrary and can be adapted for particular downstream analytical needs. We first determine the global, equal area histogram cut lines for each feature space (based on all residents) to create a normalized nightly 12-dimension histogram per resident. To do this, we fetch all the data for all nights for the extracted feature and group them in 15-minute intervals. Next, a global histogram with 1000 bins is computed from this grouped data. The global histogram is then used to generate a cumulative distribution function (CDF) to find the histogram cuts for the 12 equal-volume buckets for the feature space. The histogram cuts are chosen using cuts of 8.333% steps based on the CDF. Once the bin cuts are defined for the entire feature space, the process can generate individual night histograms per resident using the global bin cuts.

Figure 3.6, 3.7, and 3.8, shows the cumulative distribution for *restlessness*, *respiration*, and *heart rate* for all nights for all residents. Each bin of the *restlessness* global

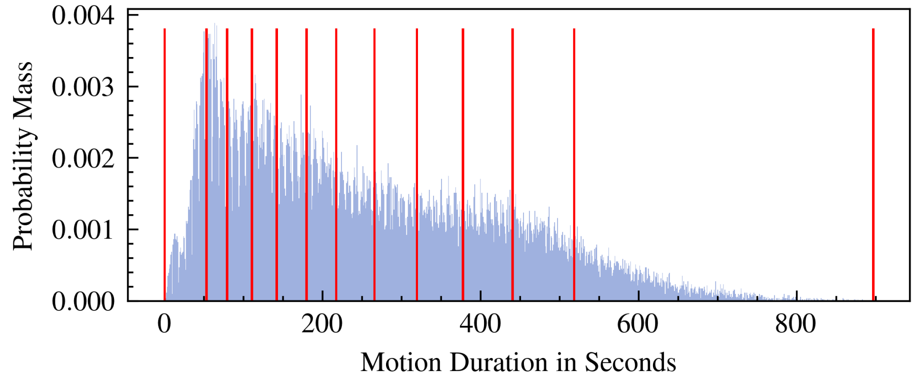


Figure 3.6: Global histograms for *restlessness* using all 27 residents. The histogram is normalized, and 1000 bins are used. Red lines represent the 12 bin cuts from the data-processing pipeline.

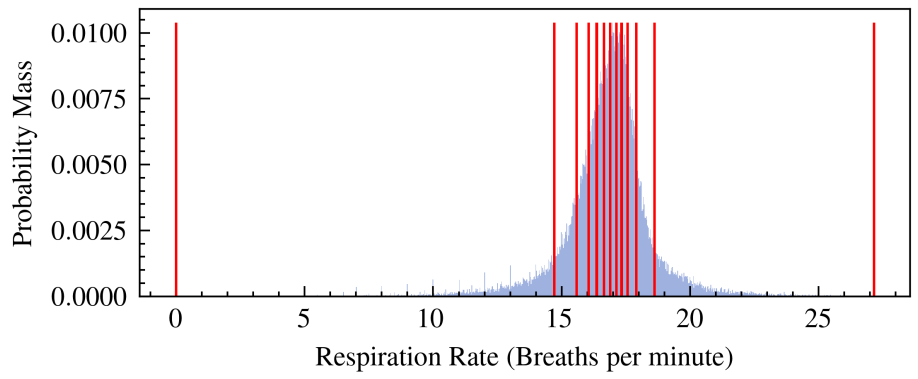


Figure 3.7: Global histograms for *respiration* using all 27 residents. The histogram is normalized, and 1000 bins are used. Red lines represent the 12 bin cuts from the data-processing pipeline.

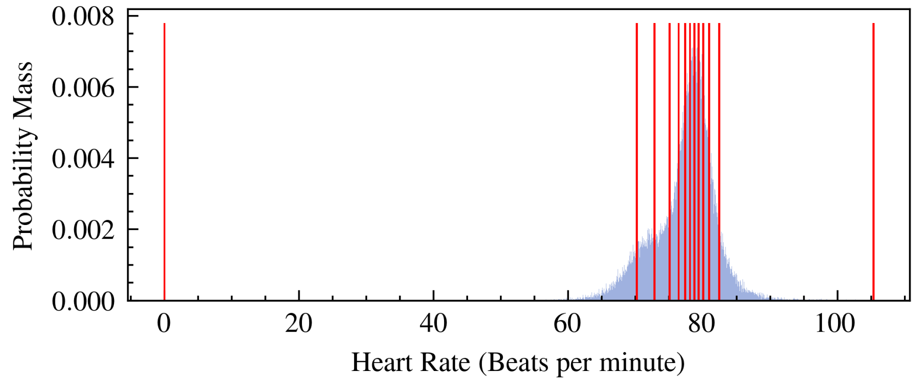


Figure 3.8: Global histograms for *heart rate* using all 27 residents. The histogram is normalized, and 1000 bins are used. Red lines represent the 12 bin cuts from the data-processing pipeline.

histogram represents the accumulated seconds of motion within a 15-minute duration. This histogram in Fig. 3.6 shows a right-skewed distribution with the middle 50% of the extracted feature data lying between 111 and 400 seconds of accumulated motion duration during any 15-minute interval for every night. The median motion duration for any 15-minute interval for all 19 residents was recorded at 218 seconds. The global histogram for *respiration* shown in Fig. 3.7 shows the average *respiration* rate within a 15-minute duration for all 27 residents. The *respiration* global histogram shows a near Gaussian distribution with most of the values lying between 15 and 20 breaths per minute, which closely aligns with the average respiratory range (between 12 to 20 breaths per minute) for a healthy adult during sleep [36]. Understanding "normal" sleeping *respiration* patterns and being able to create derived features that measure *respiration* distribution over a night for an individual is a key capability to detect breathing and respiratory abnormalities in future research. The global histogram for the *heart rate* shown in Fig. 3.8 shows the average beats per minute within a 15-minute duration. The *heart rate* global histogram shows a near Gaussian distribution with most of the values lying between 60 and 96 beats per minute.

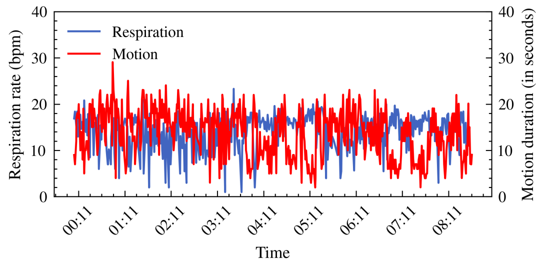
Once the processing pipeline has processed the derived feature values from all residents, a query statement for decompressing and aggregating the data can be applied as shown in Figure 3.9. Figure 3.9 shows how to aggregate the extracted feature data and how we can leverage the Structured Query Language (SQL) for advanced data aggregations and data analytics.

Furthermore, SQL aggregation statements can be performed over one night of data to generate time-series data plots such as breaths per minute or motion duration (Fig. 3.10) of residents. The multi-modal derived features from Fig. 3.10 are taken and converted into the vector feature space (described in Fig. 3.5) to produce the individual night histograms shown in Fig. 3.11.

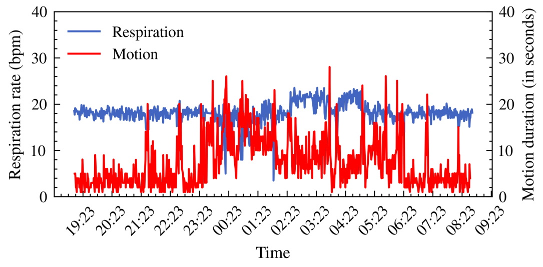
We can use the 12-dimensional vectors under the same time window to further

```
WITH tmp AS (
SELECT
  respiration_rate + unnest(offset[:])
  * INTERVAL '1 second',
  unnest(resp_rate[:])
FROM respiration_table
WHERE participant_id = id AND start_resp
  BETWEEN start_ts AND end_ts)
SELECT
  DATE_TRUNC('hour', start_resp) +
  (((DATE_PART('minute', start_resp) / 15) * 15)
  || 'minutes')::interval AS _15_min_tstmp,
  AVG(respiration_Rate)
FROM tmp
GROUP BY _15_min_tstmp;
```

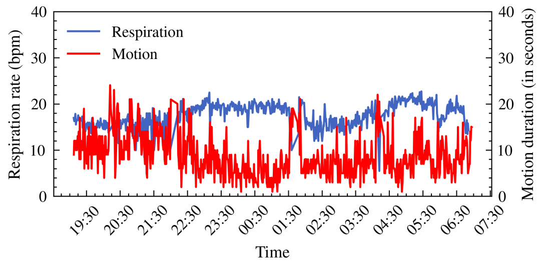
Figure 3.9: SQL query statement aggregating *respiration* data in 15-minute intervals for the specified time window. The initial step is to unnest the compressed column segments into an absolute time-series of measurements (`WITH` clause) then aggregate with traditional SQL syntax. The 900 seconds is used to group into 15-minute intervals.



(a)

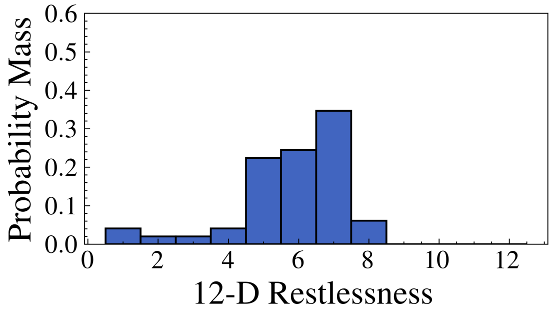


(b)

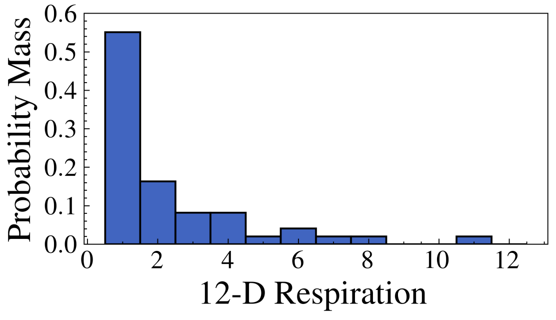


(c)

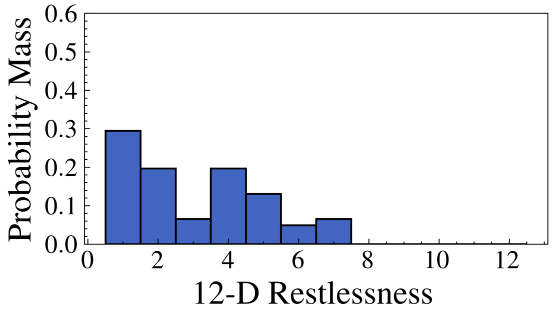
Figure 3.10: Extracted multi-modal feature time-series plots over one night for residents (a) 1, (b) 2, and (c) 3. It can be seen that there are characteristic differences in the time series of measurements and clear distinctions between residents. The aggregation to 12-D feature vectors facilitates comparisons between residents or even different nights of the same resident. We can also see that the time in bed is drastically different between the three residents.



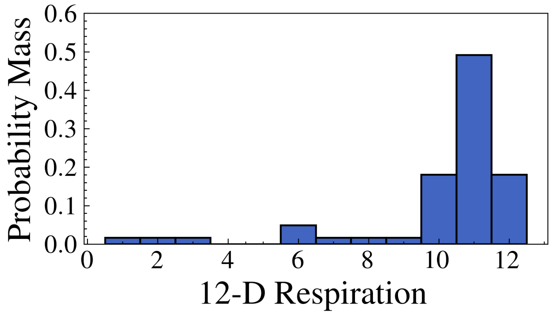
(a)



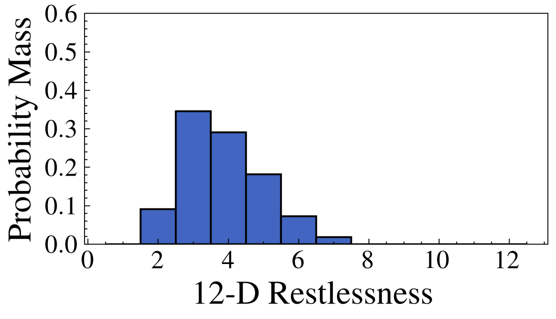
(b)



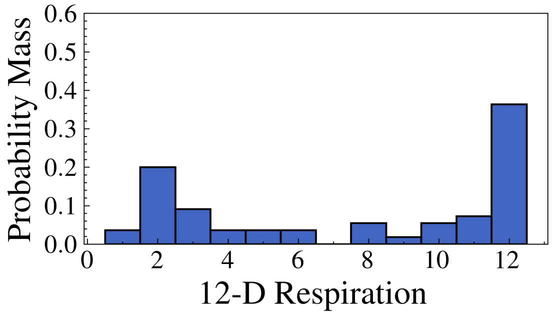
(c)



(d)



(e)



(f)

Figure 3.11: Feature space histograms for residents 1, 2, and 3. (a) & (c) & (e) are the *restlessness* one-night histograms of residents 1, 2, and 3. (b) & (d) & (f) are the *respiration* one-night histograms of residents 1, 2, and 3. The histograms are created using the individual 12-dimensional vector of one night for each extracted feature for each of the residents. We can see clear signature differences between motion from *restlessness* (a) & (c) & (e) and breathing patterns (b) & (d) & (f) among each of the residents, as well as differences between residents.

study whether a correlation between the *respiration* and *restlessness* features exists. Figure 3.11 shows the 12-dimensional feature space vector for both extracted features over one night for the same three residents. Resident 1 presents higher motion throughout the recorded night as seen in Fig. 3.11a and a low-valued *respiration* histogram (Fig. 3.11b) describing lower respiration values for that night compared to the global distribution of respiration in the residents. As seen in Fig. 3.11c, resident 2 presents an overall low-value *restlessness* histogram describing a night with low motion. Moreover, the *respiration* 12-dimensional vector in Fig. 3.11d shows a high-valued feature vector representing higher breaths per minute over the same window night. In resident 3, we see *restlessness* that is more Gaussian-shaped, and the *respiration* is clearly bimodal. This preliminary work leads us to believe that we will be able to mine these feature spaces for multi-modal physiological correlations related to resident health. While we have illustrated just one particular night for only three residents, we expect to be able to mine patterns from the data to, for example, discover anomalous nights for particular residents or discover associations between sleeping *respiration*, *restlessness*, and *heart rate* associated health conditions. Additionally, due to the performance and scalability of the system, researchers will be empowered to accelerate such research.

3.3 Feature Big Data Database

The design for all derived feature data follows similar patterns to the current way the raw bed sensor data are stored (see extended details in [37] or in Chapter 2). Likewise, all the extracted features discussed are to be stored in a column-storage segment layout, as shown in Fig. 3.12. The feature data table design shown in Fig. 3.12 is a logical representation of a resident feature data structure for a specific row, which holds a segment of time-series measurements at any time interval (e.g., both regular

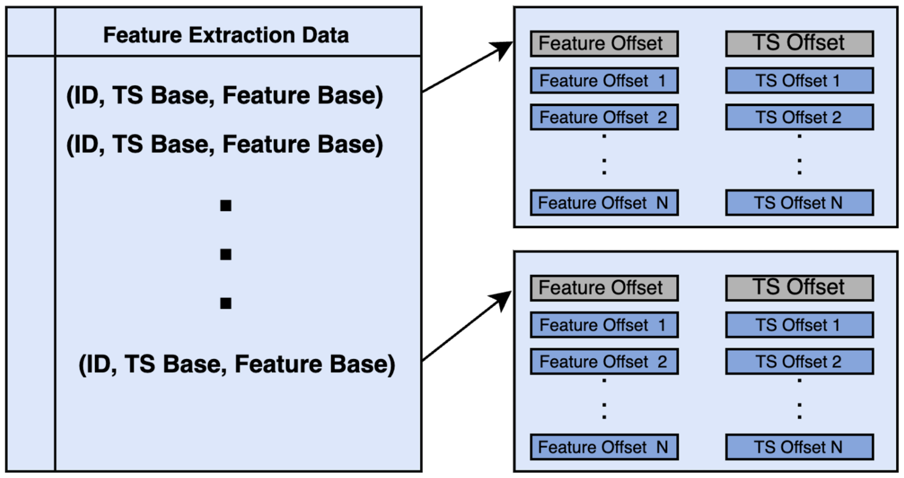


Figure 3.12: HTI having a block of feature data (*respiration*, *restlessness*, or *heart rate*) and each individual record having a start motion offset and their corresponding feature value for the specified offset.

and irregular). Each table row represents the derived feature data computed for one night. Each row is uniquely identified by a composite primary key with a residents' *id*, *start time*, and *end time*. All the derived feature values are stored in a dense array as a column-store segment, with each value inside the dense array representing the feature value at a given range specified by the start time and time offset. The column-storage segment layout allows the storage of the extracted features in dense arrays, which allows the PostgreSQL DBMS to compress the data efficiently. For all feature extraction tables (*respiration*, *restlessness*, and *heart rate*), for each resident, each feature is calculated in one-minute intervals, with the start and end of when the extracted feature was detected. The aggregated minutes are grouped by night for every resident starting from `19:00` to `10:00` of the following day, (up to 15 hours of data). After processing the nightly data aggregations, the timestamp offsets are calculated given the first recorded timestamp. With this layout in place, every row for every resident is one night of the specified feature data.

Table 3.1 is an example of how two rows would look for the *restlessness* feature sensor data table. The first row values for Table 3.1 have the *timestamp base* value

Table 3.1: Example of hierarchical restlessness feature data with dense feature arrays, a base timestamp, and temporal indexing offsets in seconds.

TS Base	Offset	Motion duration in seconds
2017-09-16	0.00	1.35
20:49:00	4.21	2.47
	23.56	10.12
	132.39	16.74
	435.33	32.91
2017-09-17	0.00	5.02
21:55:15	10.05	3.01
	22.07	1.13
	29.11	5.21
	42.14	1.00
2017-09-18	0.00	7.02
22:27:06	8.02	2.03
	11.03	3.00
	15.06	5.02
	26.10	4.01

at `2017-09-16 20:49:00`, meaning for this row, the initial motion was initially encountered on September 16, 2017, at 8:49 p.m. In the second column, array values of the time range are calculated by adding the timestamp base with the second offset (4.21 s), the third time range with the third offset (23.56 s), and so on. The respective motion duration value for the n^{th} range is going to be located at the n^{th} position in the motion array. This means for the first-time range (`20:49:00`), the motion duration value is 1.35 s. The second row starts at `2017-09-17 21:55:15`, the third row `2017-09-18 22:27:06`, and both rows follow the same pattern. Even though Table 3.1 is a specific example of two rows for the *restlessness* table, the *respiration* feature follows the same design patterns but with different dense array values. Additionally, the dense array length within the rows can vary greatly from row to row, depending on the original bed sensor data and the physiological features that are extracted.

```
CREATE OR REPLACE FUNCTION view_sensor_data(
    table_name TEXT,
    start_time TIMESTAMP,
    end_time TIMESTAMP
)
RETURNS SETOF sensor_data AS $$
DECLARE
    query TEXT;
BEGIN
    query := format('
        SELECT
            tstamp_base + unnest(ts_offset_seconds[:]) * INTERVAL ''1 second'' as tstamp,
            unnest(data[:][1:1]) as r1,
            unnest(data[:][2:2]) as r2,
            unnest(data[:][3:3]) as r3,
            unnest(data[:][4:4]) as r4
        FROM
            sensor.%s
        WHERE
            tstamp_base BETWEEN $1::timestamp AND $2::timestamp
        ORDER BY
            tstamp_base
    ', table_name);
    RETURN QUERY EXECUTE query USING start_time, end_time;
END;
$$ LANGUAGE plpgsql;
```

Figure 3.13: SQL view decompressing the raw bed sensor table with all the raw readings for computation.

```
CREATE OR REPLACE FUNCTION get_minute_data(
    table_name CHARACTER VARYING, start_time CHARACTER VARYING, end_time CHARACTER VARYING)
RETURNS SETOF minute_sensor_data AS $$
DECLARE
    v_sql_dynamic TEXT;v_record RECORD;
BEGIN
    v_sql_dynamic := format('
        WITH r AS (
            SELECT * FROM view_sensor_data(''%s'', ''%s''::timestamp, ''%s''::timestamp)
        ),
        agg_data AS (
            SELECT
                DATE_TRUNC(''minute'', r.tstamp) AS minute_tstamp,
                MAX(ARRAY[r.tstamp]) AS end_tstamp,
                MIN(ARRAY[r.tstamp]) AS start_tstamp,
                ARRAY_AGG(r1) AS r1_agg, ARRAY_AGG(r2) AS r2_agg,
                ARRAY_AGG(r3) AS r3_agg, ARRAY_AGG(r4) AS r4_agg,
                ARRAY_AGG(r.tstamp) AS tstamp_array
            FROM r
            GROUP BY DATE_TRUNC(''minute'', r.tstamp)
        )
        SELECT * FROM agg_data', table_name, start_time, end_time);
    FOR v_record IN EXECUTE v_sql_dynamic
    LOOP
        start_tstamp := v_record.start_tstamp;
        end_tstamp := v_record.end_tstamp;
        tstamp_array := v_record.tstamp_array;
        r1 := v_record.r1_agg; r2 := v_record.r2_agg;
        r3 := v_record.r3_agg; r4 := v_record.r4_agg;
        RETURN NEXT;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

Figure 3.14: Grouped data in 1-minute intervals retrieved from raw bed sensor table with all the raw readings for *restlessness* computation.

3.4 Summary

This chapter has provided an overview of the data processing pipeline for multi-modal feature extraction from hydraulic bed sensor data. Specifically, this chapter has explained in depth the feature derivation of *respiration*, *restlessness*, and *heart rate*. Moreover, this chapter has proposed a database design greatly influenced by the HTIDB from Chapter 2. The data processing pipeline facilitates machine learning research such as pattern mining for cohort discovery within the resident population, and temporal analysis of evolving sleep and health patterns in residents. Lastly, this chapter has discussed various examples of multi-modal features across multiple residents.

Chapter 4

PostgreSQL Extensions with SPI and CUDA

With the exponential growth in computational tasks across all areas of computer science, such as ML and AI, the necessity to customize DBMS to enable GPU acceleration has become increasingly significant. In this light, PostgreSQL stands out due to its capacity to adapt and evolve. This powerful DBMS has the ability to incorporate user-defined functions written in the C programming language and execute SQL commands within these functions or procedures. In this chapter, we will delve into how we can leverage the Server Programmable Interface (SPI) and Compute Unified Device Architecture (CUDA) extensions to improve PostgreSQL's performance and handle more intensive computation tasks efficiently. Moreover, we will present a set of CUDA wrappers to easily develop PostgreSQL extensions utilizing GPU acceleration all through SPI.

4.1 SPI in PostgreSQL

PostgreSQL SPI allows C/C++ code to interact with PostgreSQL DBMS [38]. This interface is important because it extends the functionality of PostgreSQL by creating custom C/C++ functions that can be called directly from SQL functions or triggers. Moreover, the SPI provides various functions, including executing SQL statements, retrieving data from the database, managing transactions, access to the parser, planner, executor, and more. Another benefit of using the SPI is PostgreSQL's ability to allocate memory within memory contexts, which offers a practical way of managing allocations made in different places with different scopes inside the code. All the memory allocated in a context is released when the context is destroyed. Hence, the SPI reduces the individual objects needed to be freed, avoiding memory leaks; instead, only a few contexts need to be manually managed.

4.2 CUDA integration with SPI

In this research, we use the SPI to enable the integration of GPUs inside DBMS by using CUDA within PostgreSQL. CUDA is a general-purpose GPU (GPGPU) framework and programming model developed by NVIDIA to solve complex computational problems more efficiently than on a CPU in C/C++ [39]. By utilizing PostgreSQL SPI, CUDA code can be called from within PostgreSQL functions and triggers, enabling GPU-accelerated computations to be performed directly inside the DBMS. We must first use the SPI connect function to establish a connection to the SPI. This function creates and opens a connection to the SPI manager, creates memory contexts for the current procedure, enters the newly created stack level, initializes the SPI executor, and connects to the PostgreSQL database. We must create a Portal to call the SPI cursor to execute different SQL queries inside the SPI. A Portal is an abstraction that represents the execution state of a running or runnable query.

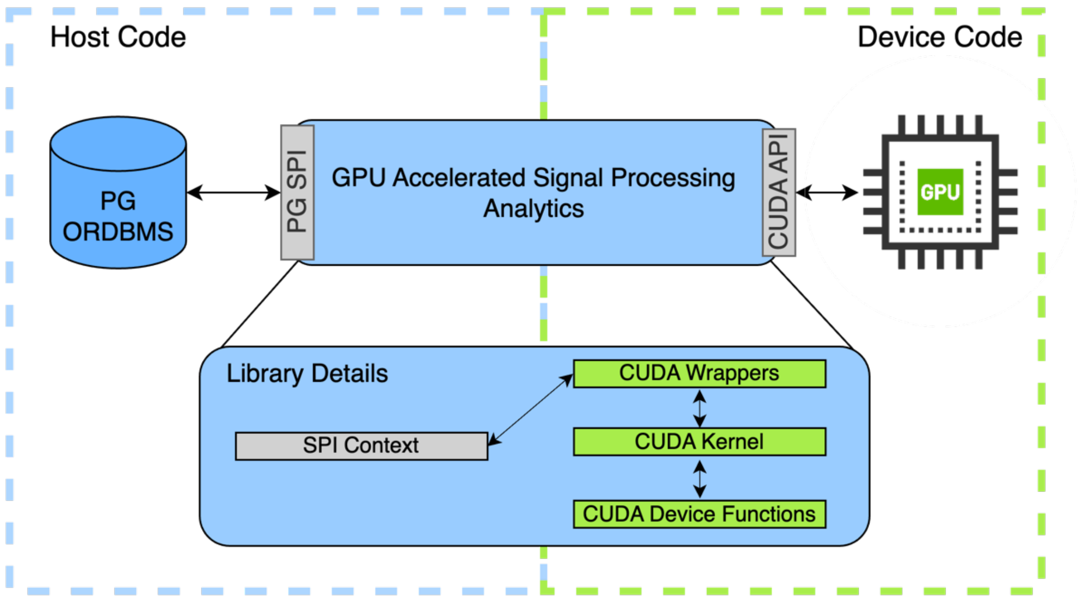


Figure 4.1: High-Level diagram of the general integration of a DBMS accelerated procedure using CUDA.

Finally, we fetch and process each result to compute the final results. Independent of the type of PostgreSQL server extensions built with the SPI, the same process must be followed to interact with the SPI and PostgreSQL successfully.

Figure 4.1 presents a high-level diagram illustrating the integration of the GPU into the DBMS within PostgreSQL using SPI. The code is segregated into two parts: host code and device code. The CPU compiles and executes the host code, while the device code is first compiled into Parallel Thread Execution (PTX). CUDA uses PTX as an intermediary assembly code in its compilation process [40]. The PTX code is then optimized and translated into the actual machine code for the targeted GPU architecture. The host and device codes are treated differently and kept separate, as shown in Fig. 4.1 where the gray and green boxes in the library details section join them. The SPI Context and CUDA connection operates by linking the two within a CUDA wrapper intermediary header file. This header file holds the function declarations and macros that can be compiled and read by regular C/C++ code. The CUDA wrapper functions defined in the intermediary header file are responsible

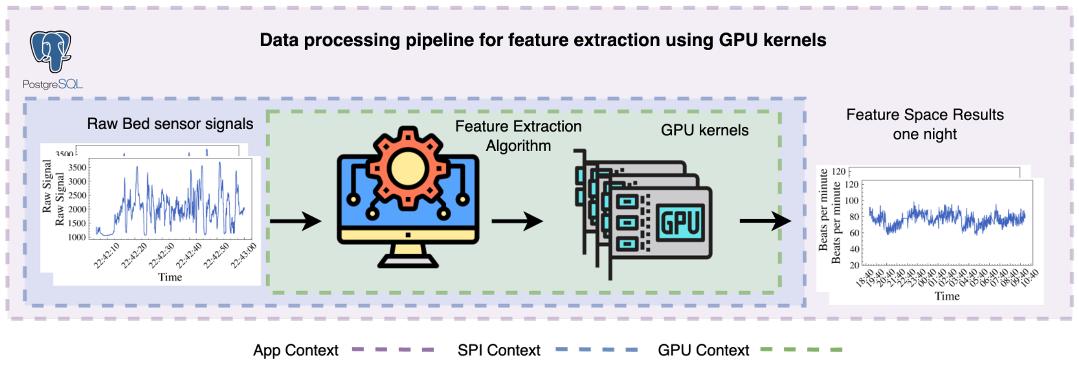

Data processing pipeline for feature extraction using GPU kernels
PostgreSQL
Raw Bed sensor signals
Feature Extraction Algorithm
GPU kernels
Feature Space Results one night
Raw Signal
Time
Beats per minute
App Context
SPI Context
GPU Context

Figure 4.2: Data processing pipeline for physiological feature extraction. Data is taken from the raw tables, specific feature extraction algorithms are applied, and features are generated.

for allocating resources on the device, transferring data from host memory to device memory, calling the kernel, and collecting the results from device memory to host memory. The following and final link is the kernel's device functions, which are only linked to the CUDA kernel files, as shown in Fig. 4.1.

4.3 Heart Rate Estimation using SPI and CUDA

The *heart rate* estimation algorithm, as previously described, is derived from the transducer signal measurements and computed within PostgreSQL using SPI. The SPI session and context manager abstract much of the work involved in working with C/C++ code. The specified algorithm for a physiological feature is computed, and the algorithm results are populated and sent back to the application. This process constitutes an improvement from the prior feature extraction methods conducted in Python and Matlab, which exhibited decreased processing times due to the computational overhead of high-level language calls.

Figure 4.2 shows the data processing pipeline for *heart rate* feature extraction from hydraulic bed sensor data. The pipeline starts with selecting the best transducer signal from the four f readings by calculating the cumulative sum of the amplified

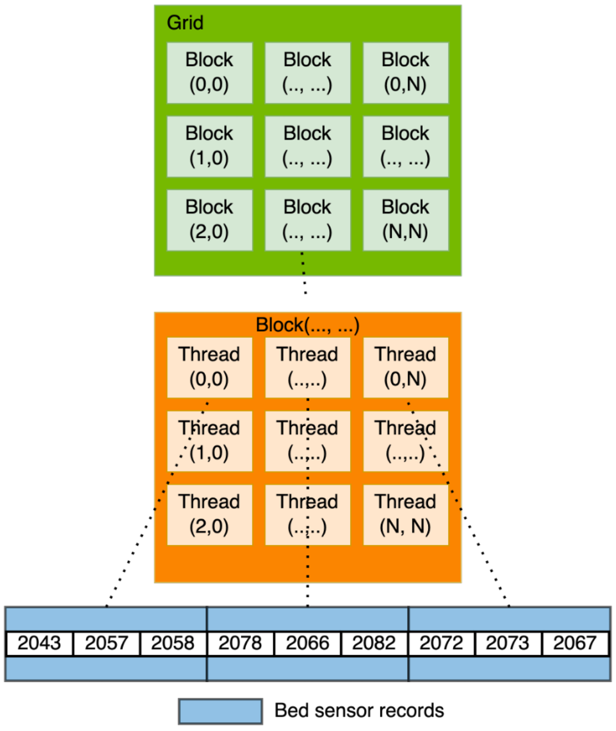


Figure 4.3: CUDA grid/block configuration for deriving and computing the different features. Each block computes the feature values of an entire minute.

readings (r) over a one-minute period. The transducer (1-4) with the highest summated count of readings is the best signal representative. The next step is to apply the feature extraction algorithm to the bed sensor data. For the *heart rate* algorithm, a short-time energy algorithm is used [35]. This algorithm applies a 6 order Butterworth band-pass filter to remove the respiration and high-frequency noise components from the transducer signals, with the cutoff frequencies set to 0.7 Hz to 10 Hz. Outlier removal is performed, followed by a 1-D convolution to smooth the signal. Finally, the estimate for the *heart rate* is computed. Now that we have established a data processing pipeline that builds signal processing algorithms inside the SPI, a solution using a GPGPU can be described and constructed more efficiently. When designing the algorithm inside a GPU, the main steps are clearly defining the thread and block configuration, data layouts, memory configuration, and thread synchronization, among others.

The thread and block configuration of the CUDA kernel is shown in Fig. 4.3. Each block in the CUDA kernel can consist of 64, 128, 256, 512, or 1024 threads, with a dynamic number of blocks per grid based on the number of minutes needed to compute. Each block estimates the *heart rate* of an individual minute, with threads inside a block communicating and dividing the work using shared memory and synchronization barriers, such as *syncthreads*. The shared memory sizes will vary based on the graphics card model. For this particular study, an NVIDIA RTX 2080 GPU was utilized. Due to the shared memory size limitations per block, only a single minute of bed sensor data can be processed for each block, given that a minute of bed sensor data is composed of 6000 elements with a limit of 49 Kbytes of shared memory. Therefore, each block is only responsible for one minute of *heart rate* estimation. Figure 4.3 illustrates how each thread is responsible for computing a subset of the signal array.

4.4 A Deep Dive into SPI Extensions with CUDA

We have discussed in previous sections, the overall benefits of building PostgreSQL extensions using SPI and CUDA. In this section, we will dive into how to create such extensions with actual code examples and pseudo-code.

When a user-defined function from a specific loadable object file is called for the first time, PostgreSQL's dynamic loader pulls that object file into memory. This ensures the function is accessible and ready to be executed. The `CREATE FUNCTION` statement is used to define a user-defined C function and requires two pieces of essential information:

- The name of the loadable object file.
- The C name (link symbol) represents the specific function within the aforementioned object file.

To work with PostgreSQL extensions, we need to use specific data types and macros. Two of the most common are **Datum** and **PG_FUNCTION_ARGS**. 'Datum' is a PostgreSQL generic data type that can represent any value, be it an integer, a float, an array, or even a complex data type like a text string or a row. PostgreSQL uses this unified approach to handle function arguments and return values in a consistent manner, regardless of their actual data type. **PG_FUNCTION_ARGS** is a macro used in the declaration of user-defined functions in PostgreSQL extensions. It represents the arguments passed to the function. It simplifies the function signature and ensures that the function can seamlessly accept any number or type of arguments, as PostgreSQL handles the actual argument extraction internally.

The standard procedure to create a PostgreSQL extension involves a particular calling convention, primarily depending on macro calls. For instance, the C declaration for defining an extension function is expressed as:
`Datum funcname(PG_FUNCTION_ARGS)` (as shown on line 15 in Listing 4.1).

Listing 4.1 shows a basic template to create a simple vector addition extension using SPI. In lines 17 and 18, the function calls the `PG_GETARG_ARRAYTYPE_P` macro, which allows you to retrieve the actual array value from the function's argument list and work with it in your C-language-based PostgreSQL extension. The macro `PG_RETURN_ARRAYTYPE_P` on line 24 is used to return an array type value from a C-language function. Now that there is a basic understating of how to create extensions, we can dive in on how to build SPI extensions with CUDA.

The pseudo-code in Algorithm 1 outlines an SPI/CUDA procedure for integrating the GPU into the DBMS. The procedure starts by creating a *TupleDesc* object (line 2), which describes the tuple format returned by the function. If it is the first call to the function, the current function context is saved, and the procedure switches to a multi-call memory context. One-time setup code is executed, such as initializing the Set-Returning Function (SRF) for the first call, connecting to the SPI, and preparing

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"
#include "utils/geo_decls.h"
#include "funcapi.h"
#include "utils/array.h"
#include "vector_add_funcs.h"

#define GET_PARAMETER_AT_INDEX(n) \
    PG_ARGISNULL(n) ? NULL : PG_GETARG_ARRAYTYPE_P(n)

PG_MODULE_MAGIC;
PG_FUNCTION_INFO_V1(vector_addition);

Datum vector_addition(PG_FUNCTION_ARGS)
{
    ArrayType *pg_signature1 = GET_PARAMETER_AT_INDEX(0);
    ArrayType *pg_signature2 = GET_PARAMETER_AT_INDEX(1);

    // Get regular float[] references
    float *a = (float *)ARR_DATA_PTR(pg_signature1);
    float *b = (float *)ARR_DATA_PTR(pg_signature2);
    int *dmns = ARR_DIMS(pg_signature1);
    PG_RETURN_ARRAYTYPE_P(perform_vector_addition(a, b, dmns));
}
```

Listing 4.1: SPI extension example for vector addition on the CPU

Algorithm 1 SPI/CUDA procedure(PG_FUNCTION_ARGS)

```
1: procedure CUDA PROCEDURE(PG_FUNCTION_ARGS)
2:     $TupleDesc \leftarrow$ create TupleDesc
3:     if SRF FIRST CALL then
4:         $oldContext \leftarrow$ current function context
5:         switch to multi call memory Context
6:         /* One-time setup code appears here: */
7:         SRF first call init
8:         SPI Connect
9:         prepare function context
10:        prepare Cursor
11:        $currentContext \leftarrow oldContext$
12:    end if
13:    $funcCtxContext \leftarrow$ current function context
14:    if SPI Processed = 0 then
15:        SPI Finish
16:        clear context
17:        SRF_RETURN_DONE
18:    else
19:        fetch results /* compute results with CUDA */
20:        $HeapTuple \leftarrow results$
21:        $Datum \leftarrow HeapTuple$
22:        SRF_RETURN_NEXT
23:    end if
24: end procedure
```

the cursor. Once the one-time setup is complete, the procedure returns to the old context (lines 3-12). For subsequent calls to the function, the current function context is retrieved (line 13) and checks if the procedure has finished processing. If it has, the procedure cleans up the context and returns done (SRF_RETURN_DONE). Otherwise, the procedure fetches results computed in the CUDA functions. It then creates a *HeapTuple* object (line 20) from the results, converts the *HeapTuple* object into their respective *Datum* representation (line 21), and returns (SRF_RETURN_NEXT) for further processing. An important aspect of the SPI/CUDA procedure described in Algorithm 1 is the utilization of the CUDA wrapper (line 19). The CUDA wrapper allows the code to remain dynamic on the SPI side, enabling the use of the same SPI structure, whether retrieving the results from the CPU or GPU. The CUDA wrapper acts as an intermediary layer that abstracts away the complexities of GPU integration with SPI. Developers can then write SPI code as if they were only targeting the CPU while also writing their GPU kernels as if they were not running them inside SPI. Using the CUDA wrapper, developers can create highly optimized GPU-accelerated applications that integrate seamlessly with PostgreSQL [41].

Listing 4.2 displays the wrapper header file used in the main SPI file where it's compiled using PostgreSQL's internal libraries. As observed, there are no internal CUDA API macros or data types present. This is a crucial consideration when linking SPI with CUDA.

Listing 4.3 provides an in-depth overview into the CUDA code responsible for the *heart rate* estimation, where it actually invokes the GPU kernel. This CUDA wrapper starts by translating host vectors into `thrust::device_vector`, allowing for efficient memory management on the GPU. The function then launches the actual GPU kernel to perform computations using CUDA's parallel processing capabilities. After the kernel execution, the function ensures synchronization with `cudaDeviceSynchronize`. Before exiting, the function copies the results from the device memory back to the host

```
#ifndef CUDA_WRAPPERS_H /* Include guard */
#define CUDA_WRAPPERS_H
#include <vector>

void cuda_wrapper_heart_rate_estimation(
    std::vector<std::vector<unsigned short int>>
       selected_filter_vector,
    float *heart_rate,
    int heart_rate_size,
    int num_of_threads,
    float *time_elapsed);

#endif // CUDA_WRAPPERS_H
```

Listing 4.2: CUDA wrapper header example for *heart rate* estimation.

using `thrust::copy`. From Listing 4.3, we see the GPU memory management, kernel invocation, and error handling in CUDA. Additionally, it underscores the utility of the thrust library, a powerful C++ template library for CUDA built on the Standard Template Library (STL), in simplifying GPU-related operations.

```
void cuda_wrapper_heart_rate_estimation(
    std::vector<std::vector<unsigned short int>>
       selected_filter_vector,
    float *heart_rate,
    int heart_rate_size,
    int num_of_threads,
    float *time_elapsed
)
{
    thrust::device_vector<int> d_offset_vector = h_offset_vector
       ;
    thrust::device_vector<T> d_heart_rate(heart_rate_size);

    //  === CUDA ===
    hr_kernel<T, 6000, 500> <<<blocks, num_of_threads >>>(
        d_selected_filter_vector_ptr,
        d_offset_vector_ptr,
        d_global_output_with_padding_ptr,
        d_heart_rate_ptr
    );
    cudaDeviceSynchronize();

    // get last CUDA error
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess){
        std::cerr << "Error: " << cudaGetErrorString(err) <<  "
           Name: " << cudaGetErrorName(err) << std::endl;
    }

    // Copy the result back to the host
    thrust::copy(d_heart_rate.begin(), d_heart_rate.end(),
       heart_rate);
}
```

Listing 4.3: CUDA wrapper example for *heart rate* estimation.

4.5 Summary

This chapter has provided an in-depth explanation of the integration of GPU within DBMSs like PostgreSQL using the Server Programming Interface (SPI). It has specifically detailed the process and presented a real-world scenario, showcasing how GPU acceleration can be harnessed in a database setting to derive critical insights. As an example, we have examined *heart rate* estimation based on hydraulic bed sensor data, a topic explored in Chapters 2 and 3. By enabling GPU acceleration within a DBMS, we can facilitate computation-intensive research in areas such as machine learning, significantly improving processing speeds and overall efficiency. Finally, this chapter delineated the necessary steps and configurations required to create a GPU-accelerated PostgreSQL extension. These steps include preparing the development environment, modifying the PostgreSQL configuration, and implementing the extension. In doing so, we have set the stage for more advanced applications, pushing the boundaries of what PostgreSQL, coupled with GPU acceleration, can accomplish.

Chapter 5

Applications

In the context of a data warehouse solution capable of supporting robust analytics, several primary factors dictate its viability and efficiency. These determinants include usability, ease of incorporation into existing workflows, efficient data access, scalability, and the capacity to handle high-volume data analytics.

The HTIDB, a design implemented in PostgreSQL DBMS and utilizing structured query language (SQL) for data access and analytics, provides a user-friendly and scalable solution. The design extends to incorporate the extracted feature data from the raw bed sensor data, creating an effective data structure for enhanced usability. Furthermore, efficient data access is critical, especially for data-intense computations and Big Data analytics, a capability demonstrated by HTIDB.

The system's scalability characteristics and stable, predictable performance over increasing spans of data are demonstrated through a series of results. Our evaluations involved running particular SQL statements 50× in all timing-based experiments to facilitate statistical analysis. We collected numerous timing statistics to illustrate the different performance characteristics essential to assessing the viability of the data solution.

A more detailed analysis of the system's performance involves the evaluation of

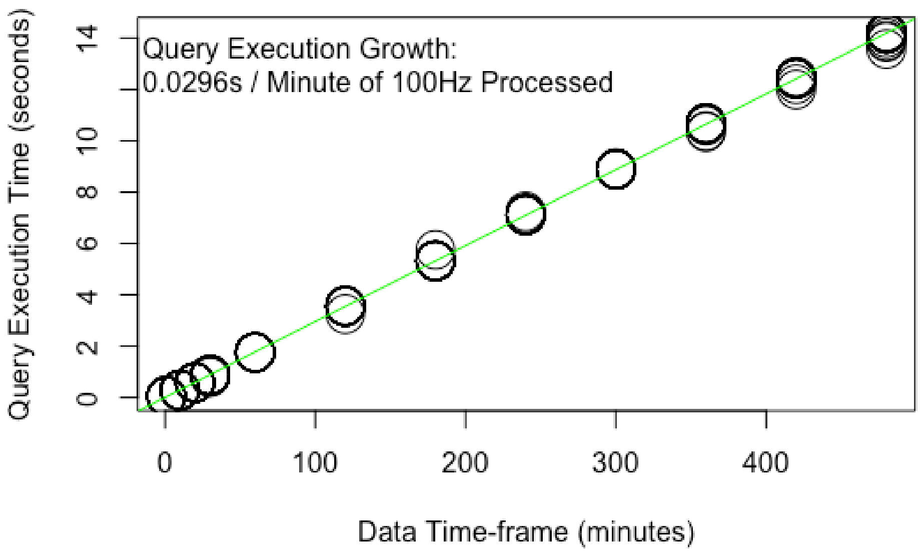


Figure 5.1: The system demonstrates consistent linear growth with increasing time-frame data access. Growth rate is modeled to 0.0296 seconds of query execution time per minute of 100 Hz of high frequency data.

both CPU and GPU approaches for computing the *heart rate* feature from hydraulic bed sensor data within PostgreSQL. This examination sheds light on the performance enhancements achieved through GPU acceleration for PostgreSQL with SPI, showing its ability to efficiently process large datasets, such as bed sensor data.

All timing experiments were conducted on an AMD Radeon 5600X CPU operating at 3.8 GHz with 32 GB of DDR4 RAM, utilizing the Ubuntu 20.04 operating system. Notably, the GPU is an NVIDIA RTX 2080 running with CUDA 11.4. The PostgreSQL version used was 14.1. Together, these results provide a comprehensive overview of our data warehouse solution, demonstrating its robustness, efficiency, and scalability for various applications.

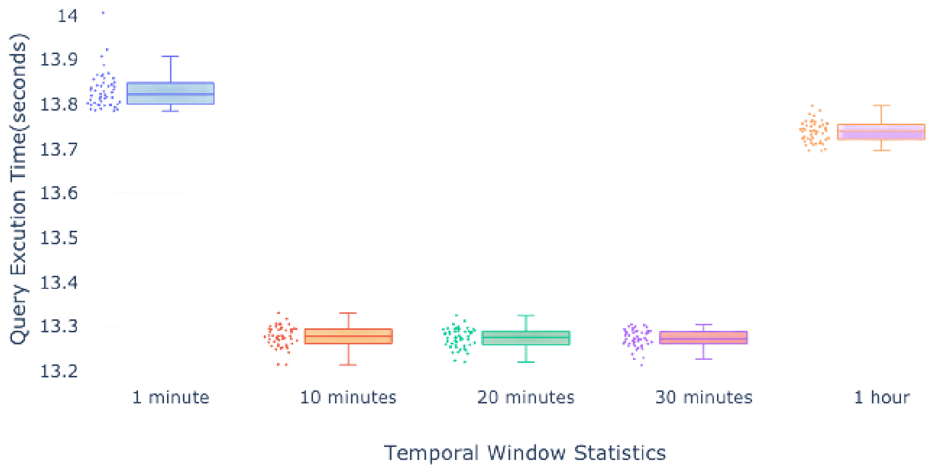


Figure 5.2: In-database analytics performed over various temporal windows. All queries are aggregating statistics from one full day of HFBD for a single resident.

5.1 Experiments

5.1.1 Timing experiments for HTIDB access

Generally, the data access time is dominated by the network transfer time of the data, and therefore the query result size. We conducted data access timing experiments, retrieving successively larger time series data. Each query pulled all 8 features, at full 100 Hz measurement frequency, for an increasing number of minutes. This ranged from 1 minute, up to 500 minutes of data, and Fig. 5.1 shows the linear growth access time to pull data from the HTIDB. The experimentation showed a linear growth rate of query time versus data requested, such that the typical performance can be expected at 0.0296 s per minute of 100 Hz data (8 features). That is, 6000 rows of eight time series in 0.03 s.

Figure 5.2 shows the distribution of query execution time for in-database analytics. In these experiments, the *minimum*, *maximum*, *average*, and *variance* of the four filtered measurements were computed simultaneously using built-in, standard SQL aggregations. We see from the box and whisker plots for five different time intervals (1-min, 10-min, 20-min, and 1-hour) the query execution time of aggregating over

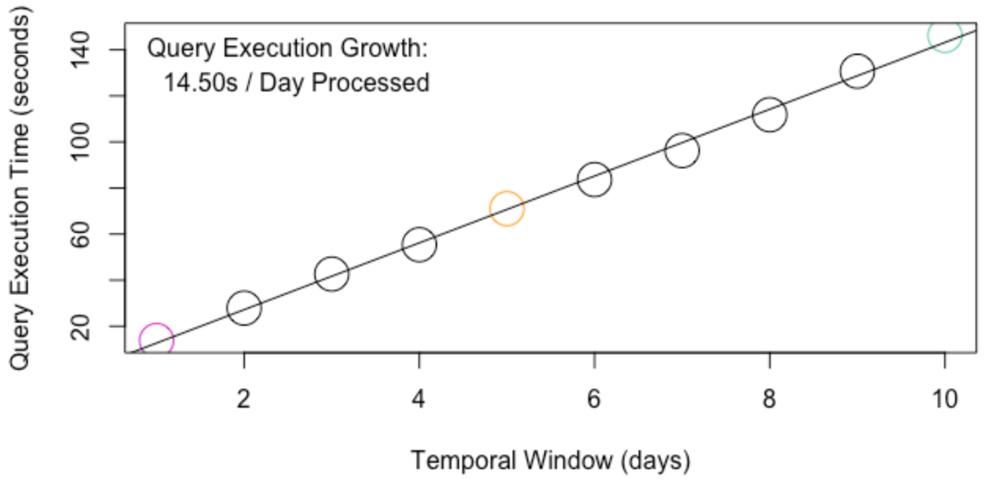


Figure 5.3: In-database analytics (aggregate averages, min, max, and variance) performed over 1-hour temporal windows, spanning differing number of days. All queries are aggregating statistics of HFBD for a single resident.

one day. That is, the 1-minute window (first column) is the distribution of query execution time for simultaneously computing 1-minute *minimum*, *maximum*, *average*, and *variance* of 100 Hz bed sensor readings within the database over the course of a day for a particular resident. This corresponds to a standard SQL *Group By* statement with aggregation. The range of each query execution is very minimal, usually less than 0.2 seconds. For this set of query timing tests, the range among all query execution times is $[13.2, 14.1]$. This tight range across particular analytic tasks indicates that the system is quite consistent when it comes to the query performance and indicates characteristics of stability and scalability.

These experiments were extended from single-day analysis to multi-day, in-database statistics. In Fig. 5.3, the timing of in-database 1-hour time interval aggregates, performed over 1-day to 10-days. This scatter plot depicts the query execution time for the 1-hour in-database aggregates over a set of one to ten days. The range of each day-span's query timing exhibited low variability, within a range of 0.12 seconds for 1-day, then 1.0 seconds for 10-days. It should be noted, in both cases the range is less than 1% as a ratio of the median time for the respective test. It is also important to note the linear query execution growth in time. This query execution growth pattern

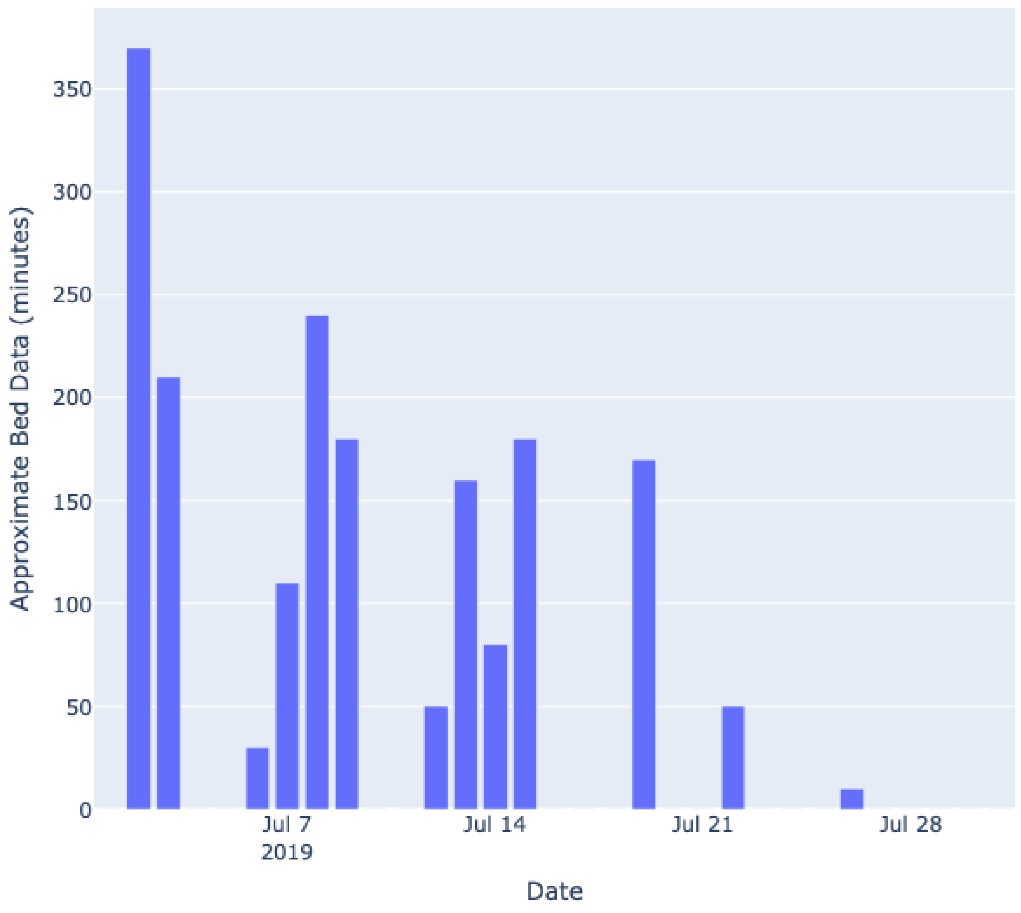


Figure 5.4: Daily *in bed* minutes approximated as sensor data for rapid snapshot of data discovery task or synoptic view of health trend.

fits a linear model with a coefficient of 14.50 s, $p-$value $< 5.461e^{-12}$ and R^2 value of 0.9979. Meaning, the rate of query time growth closely follows 14.50 seconds of query execution for each day worth of 100 Hz bed data processed. The fitted linear regression model demonstrates a strong linear relationship between the query execution time and the amount of data aggregated within the database. This strong linear relationship reinforces the evidence that our system is, indeed, not only consistent, but also highly scalable for query performance.

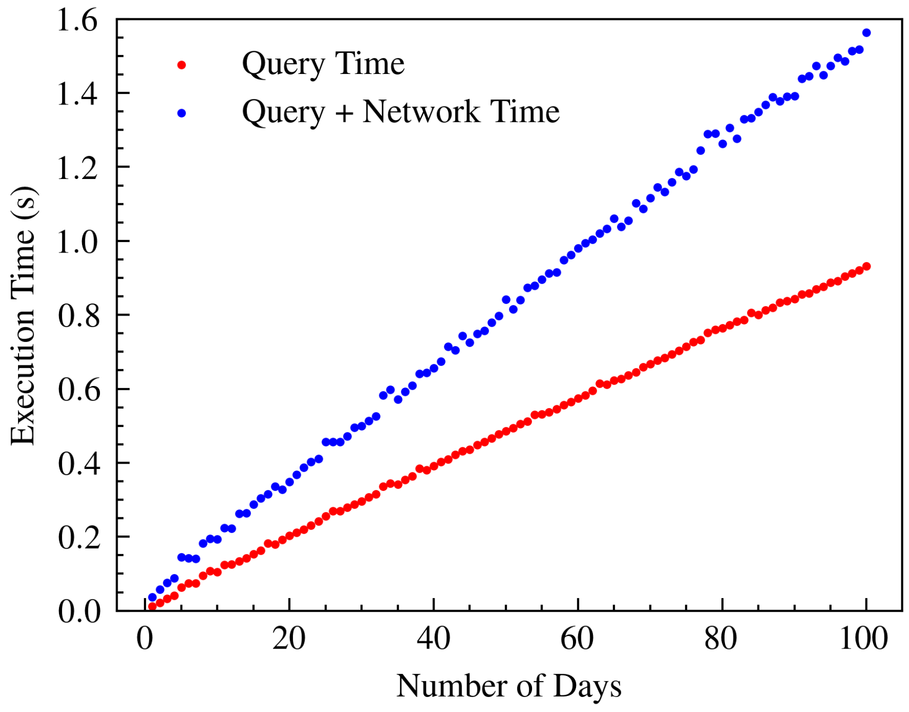
One of the most critical needs of researchers and clinicians in a connected health setting is data discovery. Figure 5.4 shows an example of a data discovery rendering for SQL query results, whereby an approximate number of minutes of bed sensor per day for a resident is rendered. This facilitates a quick monthly snapshot of the resident, and the associated queries are extremely efficient at leveraging the first-level indexing within the database. That is, the column store segments do not need to be accessed for this analysis of the relational data. The query efficiency for this plot ranges is in [0.0011,0.0024] s, and an average of 0.0014 s.

5.1.2 Timing experiments for Multi-modal derived feature tables access

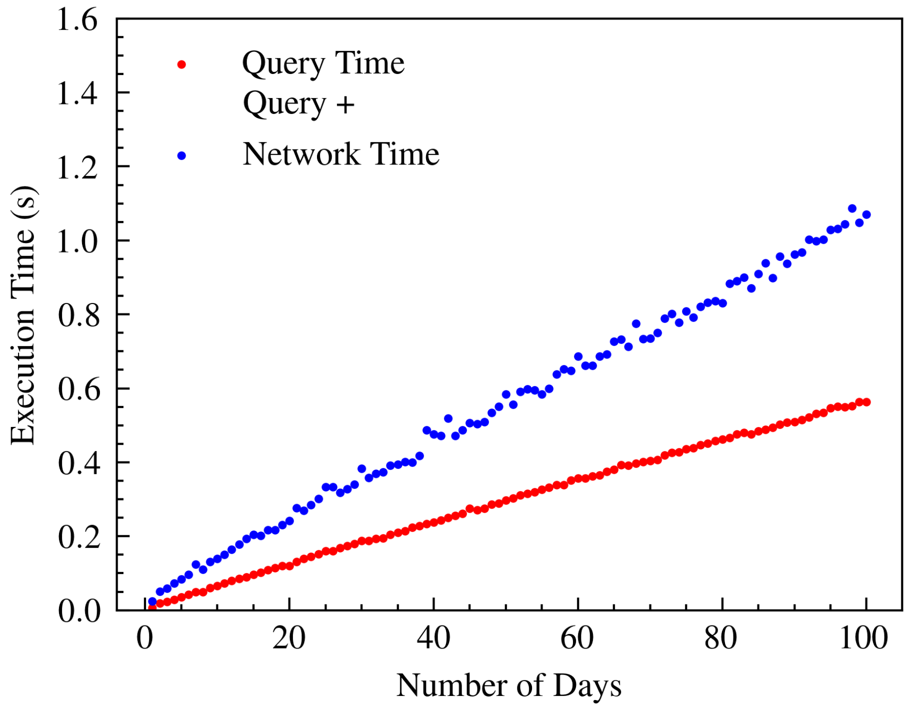
Some determining factors like usability and data access time can ultimately decide whether a data solution is viable or not. The HTI structure design for the extracted feature data achieves usability by extending the existing design of the raw bed sensor data that utilizes the HTIDB. The data solution must also provide efficient data access times for data-intense computing tasks and Big Data analytics. We have collected multiple timing statistics to show the different performance characteristics for this important factor.

Network transfer is often the bottleneck when querying data from database systems. In Fig. 5.5a, we perform timing experiments, retrieving *respiration* data for increasing time intervals. All the timing experiments performed are done using a Dell Precision-5820-Tower-X-Series with an Intel Processor i9-10980XE CPU running at 3.8 GHz with 64 GB of DDR4 RAM using the operating system Ubuntu 22.04. The data transfer throughput is limited to the workstation network card, which is rated at 1 Gbps. Moreover, the PostgreSQL version for storing and retrieving the data is v14.

In-database statistics are performed over 1-day to 100-days of *respiration* data. The timing analysis depicted in Fig. 5.5a shows a linear growth rate of execution time. A regression analysis of *query + network transfer time* grows linearly (Fig. 5.5a blue dots) at a rate of 15.17 ms per day of aggregated data, with a regression fit of p-



(a)



(b)

Figure 5.5: Growth of the query execution time over different time windows, e.g., pulling 1–100 days' worth of feature data. (a) *respiration* timings & (b) *restlessness* timings. The red line represents the query completion time. The blue line represents the total execution time (Query Time + Network Time).

value $< 1.04e^{-141}$ and R^2 value of 0.9986. Fig. 5.5a red dots are the plot of just the *query* timing, showing an in-database aggregation rate of 9.23 ms per day of data aggregated, with a regression fit of p-value $< 2.646e^{-159}$ and R^2 value of 0.9993. The data pulled was all the *respiration* values of an entire night of a resident (around 1500 data records), aggregated in 15-minute intervals, with the average breaths per minute computed.

The same timing experiment was performed on the *restlessness* data, shown in Fig. 5.5b. The *query + network transfer time* grows linearly (Fig. 5.5b blue dots) at a rate of 10.24 ms per day of aggregated *restlessness* data, with a regression fit of p-value $< 1.09e^{-124}$ and R^2 value of 0.9968. Moreover, the red dots in Fig. 5.5b show an aggregation rate of 5.59 ms per day of aggregated *restlessness* data, with a regression fit of p-value $< 4.18e^{-154}$ and R^2 value of 0.9992. An additional timing test was performed on the multi-modal feature data, randomly accessing 1000 different full days of data, showing a mean execution time of 14.3 ms (standard deviation 1.6 ms), and a median of 14.4 ms. Based on these timing and regression analyses, we can expect the system to scale sufficiently for very Big Data, maintain query performance across different temporal windows, and generally, perform as necessary for a scalable Big Data system.

5.1.3 Timing experiments for PostgreSQL extension of CPU and GPU

This section provides a comprehensive overview of the performance metrics used to evaluate the efficiency of both CPU and GPU approaches for computing the *heart rate* from hydraulic bed sensor data inside PostgreSQL. The findings presented in this section provide a detailed analysis of the performance improvements achieved by using GPU acceleration for PostgreSQL with SPI. They highlight its potential to process complex datasets, such as bed sensor data, efficiently. All timing experiments

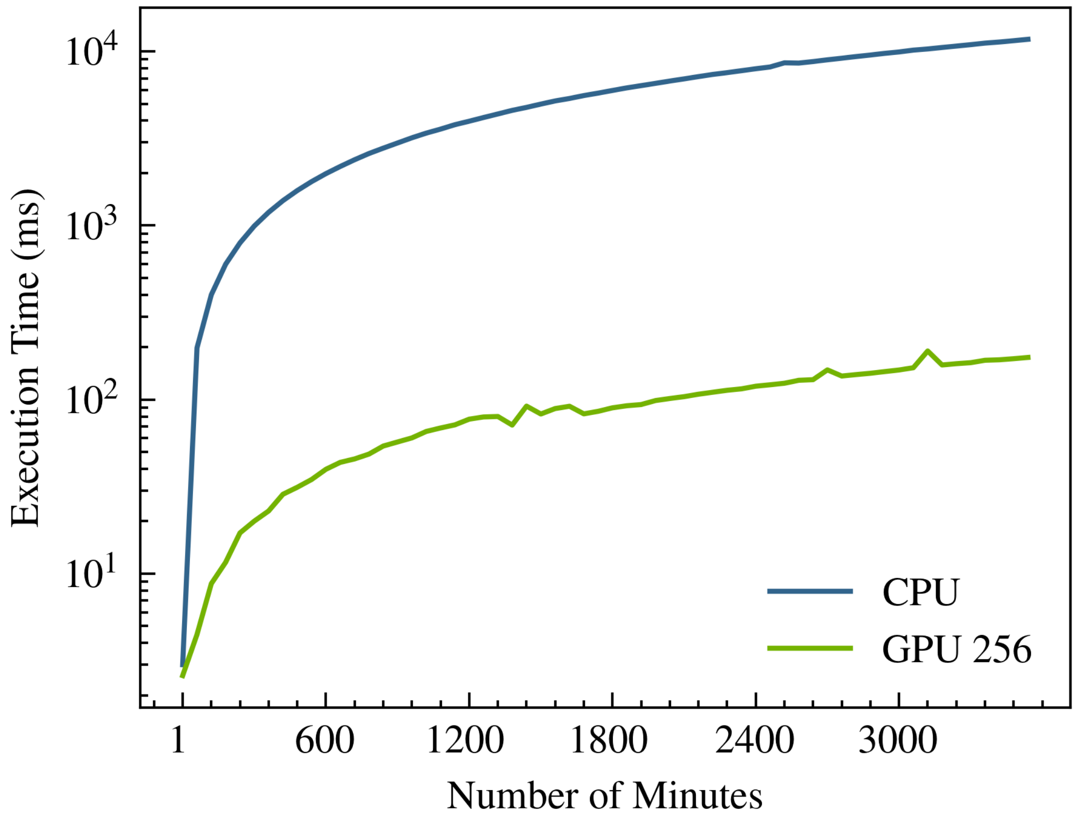


Figure 5.6: Growth of the query execution time over different growing time windows, e.g., pulling 1–3600 minutes of bed sensor data (note Log-Y scale). CPU vs 256 threads per block GPU.

were performed on an AMD Radeon 5600X CPU running at 3.8 GHz with 32 GB of DDR4 RAM, using the Ubuntu 20.04 operating system. As mentioned, the GPU is an NVIDIA RTX 2080 running with CUDA 11.4. Our PostgreSQL version was 14.1. In-database statistics compare the *heart rate* estimation execution time from bed sensor data over growing time windows, ranging from 1 to 3600 minutes. The timing analysis in Fig. 5.6 shows that the execution time increases at a constant rate for both the CPU and GPU versions. A regression analysis of GPU using 256 threads, as seen in Fig. 5.6 (green line), demonstrates a linear growth rate of 2.83 ms for every 60 minutes of bed sensor data. The regression fit has a p-value of $< 1.530e^{-51}$ and an R^2 value of 0.98088. In contrast, the CPU timing, depicted by the blue line in Fig. 5.6, exhibits a rate of 198.84 ms for every 60 minutes of bed sensor data. The regression fit for the CPU timing has a p-value of $< 5.7534e^{-119}$ and an R^2 value of 0.9999. The regression analysis indicates that the CPU version has a significantly higher growth rate of execution time, 70 times higher than the GPU version using 256 threads. This confirms our hypothesis that using GPUs in the DBMS can significantly enhance the performance of in-database statistics for processing large amounts of bed sensor data. Additionally, it is important to highlight the percentage difference between the CPU and GPU versions. When using an L1-norm to calculate correctness, the GPU version produced a difference of 0.01% compared to the 'gold standard' CPU version. This slight difference is attributed to the fact that the CPU and GPU versions are executed on distinct hardware platforms with varying precision. The difference, however, is negligible and does not affect the overall output of the *heart rate* estimation algorithm.

In Figure 5.7, the same experiment is conducted to compare the execution time for *heart rate* estimation from bed sensor data using different thread configurations. The experiment is performed over different growing time windows ranging from 1 to 3600 minutes. Despite slight differences in execution time, the figure demonstrates

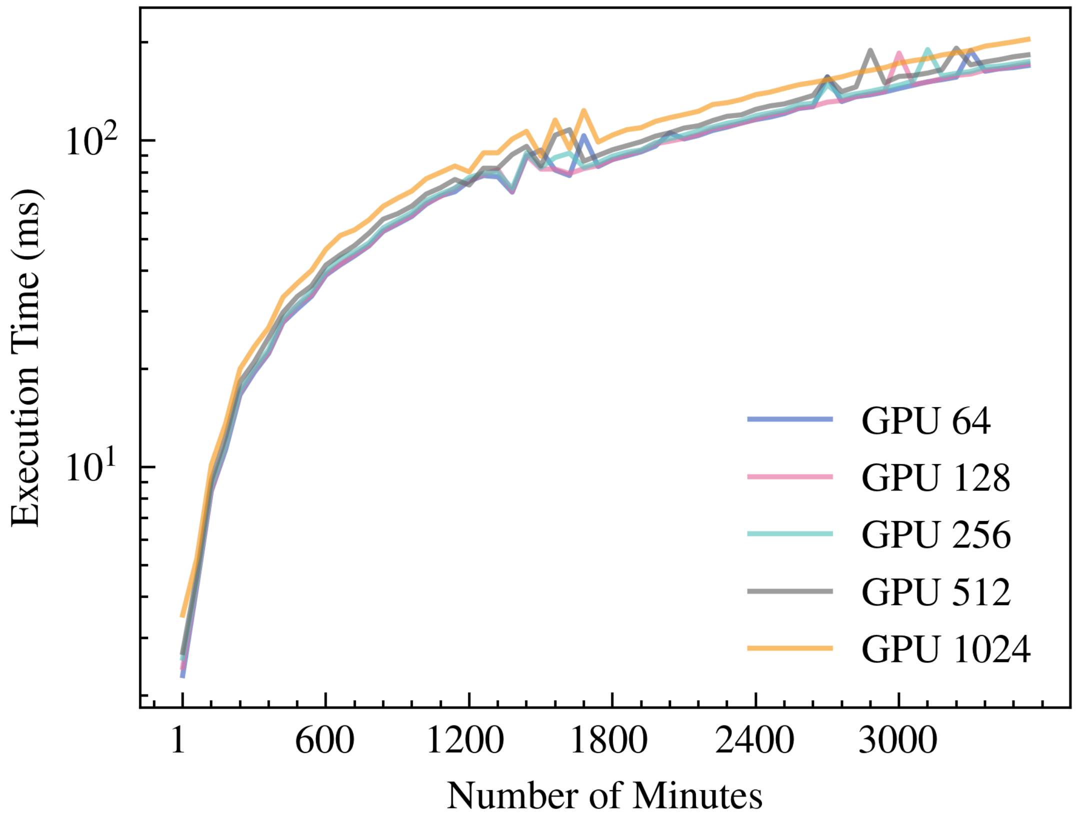


Figure 5.7: Growth of the query execution time over different growing time windows, e.g., pulling 1–3600 minutes' worth of bed sensor data. The Graph compares the execution time across multiple threads configurations.

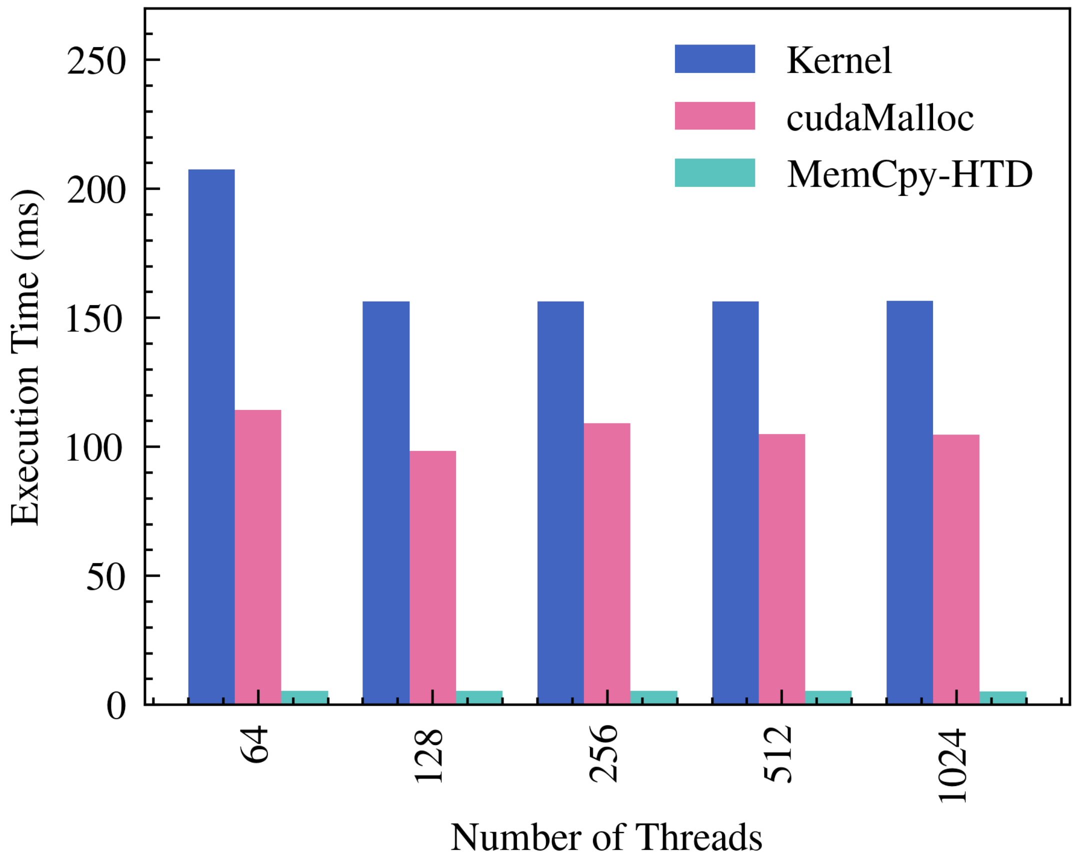
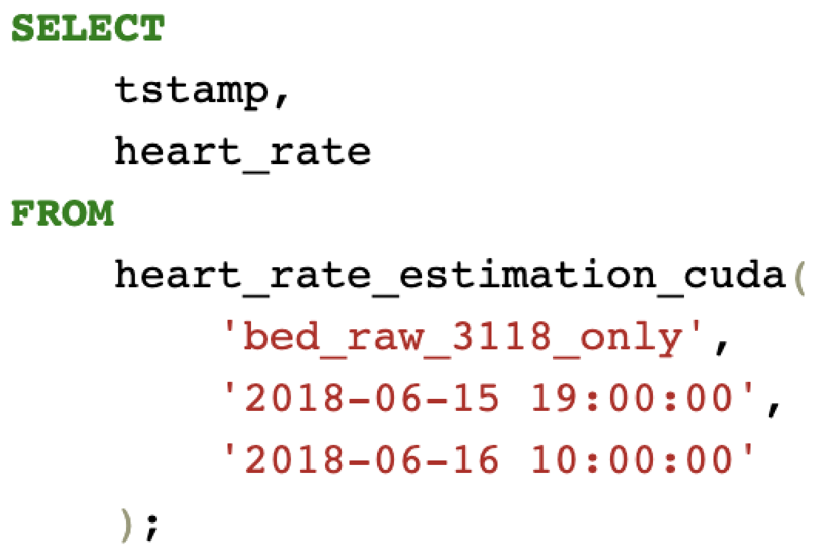


Figure 5.8: Stacked bar chart comparing different timing metrics across different threads configurations.

that the various configurations yield similar results. Notably, the configuration using 256 threads is slightly faster.

Figure 5.8 provides further insight into the execution times metrics across thread configurations. The chart depicts the performance of the GPU when computing 3600 minutes (4 nights) of *heart rate* estimation from bed sensor data. The kernel bar shows the time it took for the kernel to execute from launch to end. The cudaMalloc bar represents the aggregated time for all GPU memory allocations. In contrast, the MemCpy-HTD bar represents the aggregated time for all memory API calls to copy data from host to device memory. By comparing the timings of these various metrics across different thread configurations, we can better understand the best thread and block configuration for the specific use case.



```sql
SELECT
    tstamp,
    heart_rate
FROM
    heart_rate_estimation_cuda(
        'bed_raw_3118_only',
        '2018-06-15 19:00:00',
        '2018-06-16 10:00:00'
    );
```

Figure 5.9: SQL procedure statement returning average respiration for each minute of an entire night.

An example SQL query statement for calling the SPI procedure to estimate the *heart rate* from the resident's hydraulic bed sensor data using SPI-wrapped CUDA kernels is shown in Fig. 5.9. Although the given function is written in C/C++ using SPI and CUDA, it is called as a regular SQL procedure to retrieve the algorithm derived *heart rate* measurements. In this particular example, the SQL procedure retrieves the *timestamp* and the average *heart rate* value for every minute for the entire night from `2018-06-15 19:00:00` to `2018-06-16 10:00:00` for resident 3118.

5.2 Summary

This chapter offered an in-depth look into various applications and timing experiments associated with HTIDB, multi-modal feature access, and the SPI extension timing experiment for both CPU and GPU. The advantages of utilizing an HTIDB are evident in terms of access time, space utilization, and query efficiency. Sub-section 5.1.2 showcases the usability of multi-modal derived feature tables and their respective access times.

Furthermore, the emphasis on how different computational platforms can impact performance is a highlight of this chapter. Sub-section 5.1.3 presents performance metrics, contrasting the efficiency of CPU and GPU extensions when computing the *heart rate* feature from hydraulic bed sensor data. In this analysis, the GPU showcased its capability to significantly outperform the CPU. Specifically, by computing the *heart rate* feature on the GPU, we showed a decrease in execution time — more than 70 times faster than its CPU counterpart. This finding underscores the potential of GPUs in processing and analyzing data swiftly, offering researchers a channel for faster results and more efficient computations.

Chapter 6

Conclusion

6.1 Summary

This thesis presents a method for creating a highly scalable DBMS designed for storing, querying, and analyzing hydraulic bed sensor data. Furthermore, it demonstrates efficient techniques for computing and extracting physiological features from this data. We developed a GPU-Accelerated DBMS that leverages GPUs, achieving a performance rate 70 times faster than CPUs in extracting these physiological features. For data storage and querying, we implemented HTIDB (based on PostgreSQL). This system supports irregular time sampling, SQL-based queries, data compression, analytics over arbitrary temporal windows, and exhibits linear growth in data storage requirements and query performance. Additionally, we introduced a data processing pipeline tailored for multi-modal feature extraction from hydraulic bed sensor data. We also proposed a GPU-accelerated DBMS utilizing SPI and CUDA to create PostgreSQL extensions. This facilitates the extraction of features from hydraulic bed sensor data directly on the GPU. The methodology presented paves the way for easier integration of future signal processing algorithms that capitalize on GPU capabilities within DBMS.

6.2 Future Work

The results from our various research areas point to several promising directions for further exploration and improvement.

1. **Database Scalability and Performance:** The design of hierarchical time-indexed databases (HTIDB) has shown promise in addressing irregular time sampling and analytics over arbitrary temporal windows. Implementing the HTIDB design into NoSQL databases like MongoDB could be explored for further scalability. Additionally, the HTIDB and GPU-accelerated signal processing can be integrated into Kubernetes clusters, such as the NSF Nautilus hyper-cluster, to facilitate an open-source data warehouse solution for broader research communities outside the University of Missouri.

2. **Feature Expansion and Refinement:** The application of HTI and column-storage segment layouts to store multi-modal derived feature data has been beneficial. Extending this work would include expanding multi-modal feature data by extracting the *restlessness* and *respiration* component from the hydraulic bed sensor data. This would assist in the identification of heart-related diseases.

3. **GPU Acceleration and Optimization:** Our research demonstrated significant acceleration in the computation of physiological features through GPU integration with PostgreSQL extensions using SPI and CUDA. We plan to expand our signal-processing library using GPUs to include multi-modal features like *restlessness* and *respiration*.

Bibliography

[1] W. O. de Morais, J. Lundström, and N. Wickström. Active In-Database Processing to Support Ambient Assisted Living Systems. *Sensors (Basel, Switzerland)*, 14:14765–14785, 2014.

[2] P. Gupta, O. Ibrahim, M. Skubic, and G. J. Scott. Leveraging Unsupervised Machine Learning to Discover Patterns in Linguistic Health Summaries for Eldercare. In: *2021 43rd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*. 2021, 2180–2185.

[3] E. E. Stone and M. Skubic. Fall Detection in Homes of Older Adults Using the Microsoft Kinect. *IEEE Journal of Biomedical and Health Informatics*, 19(1):290–301, 2015.

[4] P. Heidenreich, J. Trogdon, O. Khavjou, J. Butler, K. Dracup, M. Ezekowitz, E. Finkelstein, Y. Hong, s. C. Johnston, A. Khera, D. Lloyd-Jones, S. Nelson, G. Nichol, D. Orenstein, P. Wilson, and Y. Woo. Forecasting the Future of Cardiovascular Disease in the United States: A Policy Statement From the American Heart Association. *Circulation*, 123:933–44, Feb. 2011.

[5] D. Yach, C. Hawkes, C. L. Gould, and K. J. Hofman. The global burden of chronic diseases: overcoming impediments to prevention and control. *JAMA*, 291(21):2616–2622, June 2004.

[6] L. A. Despins, G. Guidoboni, M. Skubic, L. Sala, M. Enayati, M. Popescu, and C. B. Deroche. Using sensor signals in the early detection of heart failure: A case study. *Journal of Gerontological Nursing*, 46(7):41–46, 2020.

[7] I. Sadek, J. Biswas, and B. Abdulrazak. Ballistocardiogram signal processing: a review. *Health information science and systems*, 7(1):1–23, 2019.

[8] M. Last, Y. Klein, and A. Kandel. Knowledge discovery in time series databases. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 31(1):160–169, 2001.

[9] L. Deri, S. Mainardi, and F. Fusco. tsdb: A Compressed Database for Time Series. In: *Traffic Monitoring and Analysis*. 2012.

[10] I. García-Magariño, R. Lacuesta, and J. Lloret. Agent-Based Simulation of Smart Beds With Internet-of-Things for Exploring Big Data Analytics. *IEEE Access*, 6:366–379, 2018.

[11] L. J. Phillips, C. B. DeRoche, M. Rantz, G. L. Alexander, M. Skubic, L. Despins, C. Abbott, B. H. Harris, C. Galambos, and R. J. Koopman. Using embedded sensors in independent living to predict gait changes and falls. *Western journal of nursing research*, 39(1):78–94, 2017.

[12] M. Skubic, R. D. Guevara, and M. Rantz. Automated Health Alerts Using In-Home Sensor Data for Embedded Health Assessment. *IEEE Journal of Translational Engineering in Health and Medicine*, 3:1–11, 2015.

[13] E. E. Stone and M. Skubic. Unobtrusive, Continuous, In-Home Gait Measurement Using the Microsoft Kinect. *IEEE Transactions on Biomedical Engineering*, 60(10):2925–2932, 2013.

[14] T. Banerjee, M. Yefimova, J. M. Keller, M. Skubic, D. L. Woods, and M. Rantz. Exploratory analysis of older adults' sedentary behavior in the primary living

area using kinect depth data. *Journal of Ambient Intelligence and Smart Environments*, 9(2):163–179, 2017.

[15] D. Sierra-Sosa, B. Garcia-Zapirain, C. Castillo, I. Oleagordia, R. Nuño-Solinis, M. Urtaran-Laresgoiti, and A. Elmaghraby. Scalable Healthcare Assessment for Diabetic Patients Using Deep Learning on Multiple GPUs. *IEEE Transactions on Industrial Informatics*, 15(10):5682–5689, 2019.

[16] D. Steinkraus, I. Buck, and P. Simard. Using GPUs for machine learning algorithms. In: Oct. 2005, 1115–1120 Vol. 2.

[17] N. Bandi, C. Sun, D. Agrawal, and A. Abbadi. Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations, Oct. 2004.

[18] P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In: GPGPU-3. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, 94–103. ISBN: 9781605589350. URL: `https://doi.org/10.1145/1735688.1735706`.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *J. Parallel Distrib. Comput.*, 68:1370–1380, Oct. 2008.

[20] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. In: vol. 8920. Dec. 2014, 1–35. ISBN: 978-3-662-45760-3.

[21] S. Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[22] A. Meister, S. Breß, and G. Saake. Toward GPU-accelerated Database Optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015.

[23] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[24] G. Scott, M. England, K. Melkowski, Z. Fields, and D. T. Anderson. GPU-Based PostgreSQL Extensions for Scalable High-Throughput Pattern Matching. In: *2014 22nd International Conference on Pattern Recognition*. 2014, 1880–1885.

[25] *PostgreSQL Manual*. `https://www.postgresql.org/docs/current/intro-whatis.html`. Accessed: 2023-05-18.

[26] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast Computation of Database Operations Using Graphics Processors. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, 215–226. ISBN: 1581138598. URL: `https://doi.org/10.1145/1007568.1007594`.

[27] M. J. Rantz, K. D. Marek, M. A. Aud, R. A. Johnson, D. Otto, and R. Porter. TigerPlace: a new future for older adults. *J Nurs Care Qual*, 20(1):1–4, 2005.

[28] B. Y. Su, M. Enayati, K. C. Ho, M. Skubic, L. Despins, J. Keller, M. Popescu, G. Guidoboni, and M. Rantz. Monitoring the Relative Blood Pressure Using a Hydraulic Bed Sensor System. *IEEE Transactions on Biomedical Engineering*, 66(3):740–748, 2019.

[29] D. Heise and M. Skubic. Monitoring pulse and respiration with a non-invasive hydraulic bed sensor. *Annu Int Conf IEEE Eng Med Biol Soc*, 2010:2119–2123, 2010.

[30] W. Wu, J. M. Keller, M. Skubic, M. Popescu, and K. R. Lane. Early Detection of Health Changes in the Elderly Using In-Home Multi-Sensor Data Streams. *ACM Trans. Comput. Healthcare*, 2(3), July 2021. ISSN: 2691-1957. URL: `https://doi.org/10.1145/3448671`.

[31] B. Y. Su, M. Enayati, K. Ho, M. Skubic, L. Despins, J. Keller, M. Popescu, G. Guidoboni, and M. Rantz. Monitoring the relative blood pressure using a hydraulic bed sensor system. *IEEE Transactions on Biomedical Engineering*, 66(3):740–748, 2018.

[32] L. Rosales, B. Y. Su, M. Skubic, and K. Ho. Heart rate monitoring using hydraulic bed sensor ballistocardiogram 1. *Journal of Ambient Intelligence and Smart Environments*, 9(2):193–207, 2017.

[33] L. Despins, G. Guidoboni, M. Skubic, L. Sala, M. Enayati, M. Popescu, and C. Deroche. Using Sensor Signals in the Early Detection of Heart Failure: A Case Study. *Journal of gerontological nursing*, 46:41–46, July 2020.

[34] D. Heise, R. Yi, and L. Despins. Unobtrusively Detecting Apnea and Hypopnea Events via a Hydraulic Bed Sensor. In: *2021 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*. 2021, 1–6.

[35] K. Lydon, B. Y. Su, L. Rosales, M. Enayati, K. C. Ho, M. Rantz, and M. Skubic. Robust heartbeat detection from in-home ballistocardiogram signals of older adults using a bed sensor. In: *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2015, 7175–7179.

[36] A. Rodríguez-Molinero, L. Narvaiza, J. Ruiz, and C. Gálvez-Barrón. Normal Respiratory Rate and Peripheral Blood Oxygen Saturation in the Elderly Population. *Journal of the American Geriatrics Society*, 61:2238–2240, Dec. 2013.

[37] G. J. Scott, J. Saied-Walker, N. Marchal, H. Yu, and M. Skubic. HTIDB: Hierarchical Time-Indexed Database for Efficient Storage and Access to Irregular Time-series Health Sensor Data. In: *2022 44th Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*. 2022, 2972–2975.

[38] E. Geschwinde and H.-J. Schönig. *PostgreSQL developer's handbook*. Sams Publishing, 2002.

[39] C. Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[40] L. F. Cupertino, C. P. Silva, D. M. Dias, M. A. C. Pacheco, and C. Bentes. Evolving CUDA PTX programs by quantum inspired linear genetic programming. In: *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. 2011, 399–406.

[41] J. Saied-Walker. *CUDA integration with PostgreSQL*. `https://github.com/MU-HPDI/postgres-extensions-with-cuda`. 2023.