ADB: A Time-series Database for Dataflow Programming

Michael Sokolnicki

Master of Engineering

Department of Electrical, Computer and Software Engineering

McGill University

Montreal, Quebec

December 2012

A Thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Engineering

ACKNOWLEDGMENTS

I would like to recognize the particular contribution of certain individuals without whom this work would not have been possible. Andrew Phan, whose project and initial request for help in designing a database schema for time-series data motivated the initial prototyping of adb (originally named AndrewDB). The members of my lab, Olivier St-Martin Cormier, Irina Entin, John Harrison and Prasun Lala whose ideas, criticisms and suggestion helped shape the design and purpose of adb. A special thanks to my father, Christopher Sokolnicki, whose expertise in formal discrete algebra helped me understand and formulate a theoretical temporal framework. Finally, a great thanks to Frank P. Ferrie, whose support and assistance throughout the execution of this project made adb possible.

ABSTRACT

In modern robotics systems, processing input is often the most complex and resource-intensive segment of computation. Today's sensors, especically video cameras, can produe very large data sets in very short periods of time, and it is often necessary to store this information for later processing. Unfortunately, today's database solutions are very ill-suited for a high write throughput scenario. Additionally, most of them have a very limited support for time values, which is often inadequate for precise scientific computations. This thesis attempts to formalize the problem of a time-series value storage and presents the implementation details to our proposed solution, the time-series database adb.

ABRÉGÉ

Dans le cadre des systèmes de robotique modernes, le traitement des données entrantes est souvent la partie la plus complexe et la plus exigeante en terme de ressources. Les capteurs actuels, en particulier les caméras vidéo, peuvent produire de très larges ensembles de données sur de très courtes périodes, et il est souvent nécessaire de stocker ces informations pour un traitement ultérieur. Les logiciels de base de données contemporains ne sont malheureusement pas adaptés à des conditions d'écriture à haut débit. De plus, la plupart d'entre eux ont un support limité pour les valeurs temporelles, et sont souvent inadéquats dans un contexte de calculs scientifiques de précision. Ce rapport présente une base théorique de la gestion et du stockage de données chronologiques et la réalisation du logiciel de base de données temporelles adb.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

ABSTRACT iii

ABRÉGÉ iv

LIST OF TABLES vii

LIST OF FIGURES viii

1 Introduction 1

1.1 The Collaborative Computational Network Model 2
1.2 Historical Dimension in Dataflow Programming 3

2 Data Persistence Solutions 6

2.1 Data Models and Current Database Systems 6
2.1.1 Relational and Object-based Database 6
2.1.2 Relationship-centric Databases 7
2.1.3 Domain-specific Databases 8
2.1.4 Temporal Databases 9
2.1.5 Time-series Databases 10
2.2 Solution Definition 11
2.2.1 Functionality Requirements 11
2.2.2 Other Design Goals 12

3 Theoretical Discussions 13

3.1 The Time-series Database Model 13
3.2 Temporal Algebra 16
3.2.1 Time Instants 17
3.2.2 Time Spans 18
3.2.3 Time Operations 19

3.2.4 Discrete Time . . . 22
3.3 Time Representations . . . 24
3.3.1 Universal Time, Coordinated . . . 24
3.3.2 Terrestrial Time . . . 26
3.4 ACID Properties in the Time-series Model . . . 28
3.5 Data Opacity and Aggregate Operations . . . 28
3.5.1 Arbitrary Type Storage . . . 28
3.5.2 Filter, Map and Reduce . . . 29

4 Implementation and Experiments . . . 32

4.1 Software Architecture and Components . . . 32
4.2 Data Organization and Storage . . . 33
4.2.1 Nested Variable Model . . . 33
4.2.2 Storage Methods . . . 35
4.3 Programming Language Support . . . 38
4.3.1 C API . . . 38
4.3.2 Matlab and Python API . . . 40
4.4 Universal Interface . . . 43
4.4.1 Temporal Description . . . 44
4.4.2 Data Manipulation . . . 47
4.5 Use Case Example . . . 49
4.6 Benchmarks . . . 51
4.6.1 Insertion . . . 52
4.6.2 Query . . . 55

5 Conclusions and Discussions . . . 58

5.1 Performance and Usability . . . 58
5.2 Future Work . . . 59
5.3 Closing Remarks . . . 59

Appendix A . . . 60

A.1 Internal Time Representation . . . 60
A.2 C code listings . . . 62
A.3 Data Manipulation Language . . . 64
A.4 Getting and Installing adb . . . 64

References . . . 66

LIST OF TABLES

Table page

4–1 Execution complexity for the stamped storage mode 35

4–2 Execution complexity for the slotted storage mode 37

4–3 Execution complexity for the text storage mode 37

4–4 Time units and their values 45

A–1 Primitive types and their attributes 64

A–2 List of available operators 65

LIST OF FIGURES

Figure		page
1–1	Collaborative Computational Network example	3
1–2	Blackboard model in image recognition system	5
3–1	Vector representation of temporal concepts	20
4–1	Software architecture of adb	32
4–2	ROS stack with adb	50
4–3	Insert performance comparison for scalar values	54
4–4	Query performance comparison for scalar values	56

CHAPTER 1
Introduction

Robotics systems have evolved tremendously in the past decade, thanks in part to the emergence of new tools and techniques. Beyond pure technological progress in hardware capability and software complexity, it is the collaboration in the development of this technology that is mostly responsible for the great strides the community has been able to accomplish.

Among the many feature-rich frameworks now available, the Robot Operating System (ROS) is currently the most popular, and its networking effect only precipitates its status as the de-facto standard in the industry. Its popularity comes not only from the wealth of already available building components [1] , but also from the elegance of its architecture. In ROS's case, it oddly meets the development organization in principle: that of disjoint, independent computational nodes that pass data to one another at different stages of the problem. For lack of an already established name for this model, we will henceforth refer to it as the Collaborative Computational Network (CCN).

[1] The developers of ROS, the robotics research group Willow Garage, are also the current maintainers of OpenCV, the most popular computer vision toolbox. Its integration with ROS makes the latter a very attractive development framework

1.1 The Collaborative Computational Network Model

A computer vision system (or more generally, a robotics system), can be modeled as a network of nodes exchanging data and applying transformations to them. It is a generalization of a pipeline, since multiple data streams can be available and combined in different ways. Instead of a sequence of nodes, we instead have a graph with no single entry or exit point [35]. Figure 1–1 illustrates a simple computational network.

We can identify three categories of nodes. The first contains all the sensors and probes that acquire data directly from the physical world. They are the multiple inputs of the system, and in software will most often be implemented as drivers of a particular type of data collection hardware. The second category includes all the intermediate steps, algorithms and decisional logic that form the core of the robotics system. Their inputs are most often the outputs of nodes in the previous category, but in more complex systems, they may also share data amongst themselves. The final category is especially relevant to robotics systems, but perhaps less so to our discussion: the output nodes, or nodes that consume data from the system to then directly act on the physical world.

From a data dependency standpoint, the combination of nodes resembles functional composition. However, in a live system, we need to introduce the concept of triggering, i.e. that a computation is initiated when new data it depends on are created. For instance, ROS's publisher-subscriber model ensures that when a node has finished producing some data, all nodes that depend on its output, i.e. that have subscribed to the data channel, are notified and said data

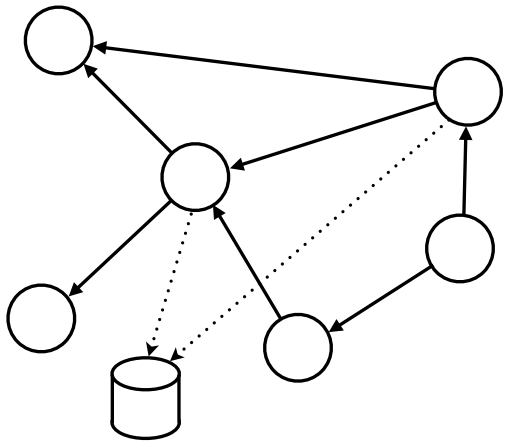

Figure 1–1: Collaborative Computational Network example

are made available to them. When this happens, the subscribers can execute their internal logic and produce output data themselves, which other nodes may have subscribed to. We can say that the first node triggers the second set of nodes, which in turn triggers the third set.

In this form, the CCN deals only with live data: new values flow through the graph with sporadic buffering in cases of asymmetrical computation time, but old data are forgotten. In some cases though, computations require not only knowledge of current values, but past values as well. This thesis presents the theoretical foundation and concrete implementation of a data storage solution that would expand the capabilities of the CCN to preserve and use historical data.

1.2 Historical Dimension in Dataflow Programming

The artificial intelligence research efforts of the early seventies faced a conceptually similar issue: they had numerous algorithms that would operate on

different data and at different stages of completion, and needed a way to keep track of intermediate results while certain components were busy. While they weren't strictly dealing with a large amount of historical values, the memory restrictions at the time bore the same problem: not all data could be kept in live memory, and there was a necessity to organize the data in permanent storage. A popular solution was the Blackboard model [10]. The idea behind this strategy was to emulate the real-life scenario of many researchers working on a problem: each would read the information he needed from the blackboard, and write back intermediate steps to arrive at the solution. Other researchers could pick up from these intermediate solutions and move forward with new data, closer to the ultimate problem solution. In software, the blackboard was therefore a generic data storage system that could accommodate multiple stages of data, and the correspondents of the researchers were individual processing units called Knowledge Sources (KS) [22]. For example, in a computer vision scenario, one KS may be a sensor module that produces an image input, a second KS may be doing early vision processing like edge detection on the image, yet a third KS could do image segmentation and finally a fourth KS could recognize patterns and write the solution. Figure 1–2 illustrates this scenario.

The current CCN model is derives some ideas from the Blackboard model. Both distribute work across individual units, are only concerned with inputs and outputs for their particular task, and the data are centrally managed. In systems like ROS, message subscription and publication is handled by a common server responsible for rerouting data to the appropriate pipes. The major difference

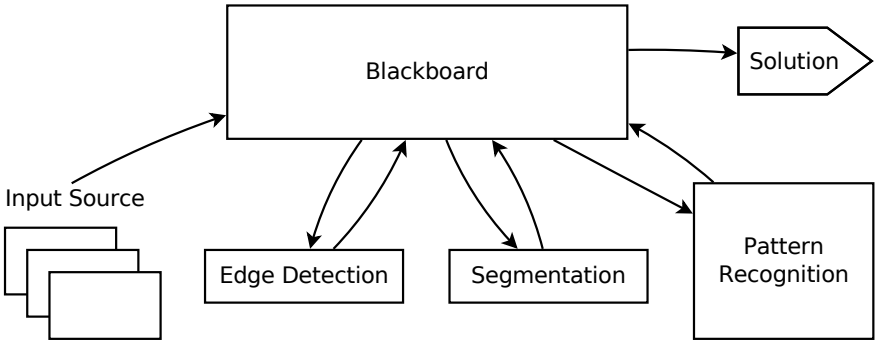

Figure 1–2: Blackboard model in image recognition system

is that in a CCN, data are passed around directly, without any retention. This subscription model works best for live data, but for historical values, an on-demand approach similar to the Blackboard has an advantage. The two methods of acquiring data could very easily coexist, and the historical repository does not need to be part of the existing delivery mechanism. In fact, a historical repository could simply subscribe to live data and progressively store it, and retrieving said data can be obtained through a connection to the storage engine. We are therefore looking for a database solution suitable for the management of historical data.

CHAPTER 2
Data Persistence Solutions

2.1 Data Models and Current Database Systems

2.1.1 Relational and Object-based Database

Since its development in the early seventies [12], the relational model for databases has been the de facto standard for almost all data storage applications. Its success is largely due to its general-purpose nature and closeness to the object-oriented model, which has itself become the standard in software development. Most mainstream object-oriented development environments, including Microsoft's .NET, Oracle's Java and Apple's Cocoa Framework, incorporate a mechanism to offer persistence support for objects to a relational database backend. This type of mechanism is usually called object-relational mapping [5]. Additionally, many web development frameworks, such as Ruby on Rails and Django, feature implicit semantics for storing objects in relational databases.

There is however an intrinsic mismatch between the object model and the relational model, and some corner cases prove to be more difficult to transition to from the live programming environment to the storage. To solve these problems, another class of database systems, known as object databases [25], attempted to deal directly with object-oriented concepts, such as polymorphism and encapsulation. One downside of these systems that has prevented them from displacing

relational databases is that their object model is tightly coupled with the programming language they interface with, which limits interoperability. A hybrid concept, the object-relational database, brings some object-oriented concepts to the relational realm [31]. PostgreSQL is currently one of the most popular such systems.

The data flow model we are faced with can be implemented using a relational or object-relational database [20]; after all, both object-oriented programming and relational databases are demonstratively general enough for almost any problem [28]. However, in our particular case, this approach comes with a significant logical and organizational overhead. First, temporal values and manipulations are supported to the second, rarely millisecond precision. Sub-millisecond is very uncommon, but it may be possible to define a new type for this purpose, along with an appropriate collation. On the other hand, the relational abilities and joins are superfluous in the case of independent variables, so the relational model is both too powerful in some areas, and not powerful enough in others. In other words, it is not a suitable tool to solve this particular problem.

2.1.2 Relationship-centric Databases

Various other, lesser-known database models have been proposed over the years, some to satisfy particular needs and fit very specific information structures and relationships. Where relational database systems allow the expression of a wide range of relations between entities, we may sometimes want to tighten the constraints in cases where the type of relationship is more important than a particular data schema.

Hierarchical databases for instance enforce a parent-child relationship between entities. Most currently used file systems can actually be categorized as hierarchical databases, where the data are files, and the relationship is that of nesting. This method of organization has not found much traction outside of this individual example [9]. A more general model of relation is the Network Model, where entities can have several parents, and therefore form a graph structure [4]. These concepts were implemented in graph databases [18], which are especially well-suited for knowledge bases. For example, in the field of robotics, graph databases can be used to store object characteristics for recognition algorithms.

More recently, the popularity of schema-free markup languages, such as XML and JSON, and the growing importance of web applications motivated the creation of document based databases, commonly referred to as NoSQL. This name was coined to distinguish them from relational databases that almost all comply to and use the SQL standard [26]. The advantage of NoSQL databases are most notable in software development practice: with no rigid schema, changing data formats and expanding the feature set is not as problematic as altering tables in relational databases. This feature, however, is of limited use in the data flow model, and the drawbacks [32] make NoSQL databases a poor candidate.

2.1.3 Domain-specific Databases

All the database models discussed so far are fairly generic in their range of applications, and the type of data they deal with. In some cases, some features of the world data allow for important optimizations and novel ways of organizing data. One such example in robotics and computer vision is spatial data, either

2D or 3D. Spatial databases organize their entities to facilitate proximity searches and spatial slicing, that is selection of elements in a specified region of space [17]. These ideas are somewhat similar to temporal databases and localization in time, which we will discuss hereafter. This characteristic yielded an extension for spatial databases that also includes time as a dimension, called spatiotemporal databases [1].

Spatiotemporal databases are useful at a stage of processing when we already have a higher understanding of scenic composition. At earlier stages, we may only have access to raw sensor information. We would wish for example to store video sequences indexed by time. This is the type of task accomplished by video databases [29], which are perhaps the most specialized data storage system we surveyed so far. Being constrained to only storing images may be too restrictive for our purposes however we wish to retain the temporal indexing mechanism, and look into databases which primarily deal with information with a time component.

2.1.4 Temporal Databases

While both SQL and NoSQL databases have some support for temporal information, they generally are not concerned directly with the temporal dimension of the data itself. They do not have direct support for recording when data is created, updated and destroyed, or provide the ability to carry different values depending on the point in time of interest. In the nineties, these ideas were explored and regrouped under the name "Temporal Databases" [30].

Temporal databases for the most part only extended the properties of relational or object-oriented databases, i.e. they were classical databases with

historical support. Because of their extensive feature set, they often are ill-suited for real-time purposes.

Unfortunately, the academic interest for this category of database has faded after a decade, and most of the software being developed is either proprietary or incomplete [8].

2.1.5 Time-series Databases

The expanded importance of web computing has not only brought demand for dynamic and schema-free data storage solutions, but it has also created a need for ways to track server metrics. While still very niche, a few non-proprietary applications that fit this purpose exist, and they can commonly be referred to as time-series databases. Unfortunately, there has been very little formal and academic study of this new breed of storage software, perhaps partly because of their very restricted use and the novelty of the approach.

Instead of records and tables, time-series databases track the values of particular metrics (or variables) that most commonly have no relationship with one another. The values are stored with the timestamp of their first occurrence, and the history of change is the primary concern of the database. Their semantics are therefore very different from the CRUD [1] -based operations of their SQL and NoSQL cousins. The value history is usually immutable, insertions and updates are coalesced into the same concept (the variable is updated, or new values of that

[1] CRUD: Create, Read, Update Delete. They form the basis of all operations in classical persistent storage applications.

variable are inserted), while queries are mostly aggregates of several values for the purpose of statistical analysis.

The semantics of the time-series database seem to be a good fit for the data flow programming model. We found, however, that the existing solutions lacked some flexibility for our particular purposes. In the next section, we formalize the requirements and desired features of the ideal solution.

2.2 Solution Definition

2.2.1 Functionality Requirements

When appraising existing solutions or developing a new one, some software features are simply indispensable. Perhaps one of the foremost of them is the requirement of an arbitrary data type. In robotic applications, numerous sensors with sometimes unpredictable data formats may be used, and rapidly replaced by newer or more capable hardware. It is therefore crucial that the data store be able to handle any digital format representation and of any size. This point allows us to eliminate a sizable portion of existing software, as most of them keep track of scalar metrics, and can only store values that are formatted as a double-precision floating point number.

The second aspect of the nature of the data is its frequency of update. A standard camera, for instance, updates the value of the image 30 times a second. Other sensors that measure acoustic data, motor rotation or lever pressure may update at even faster rates. It is therefore necessary to not only store this information at least as fast as it arrives, but represent a high degree of temporal precision, preferably sub-millisecond. Here too, this requirement may readily

disqualify a number of existing solutions, whose timestamps sometimes have a resolution of one millisecond, but typically only go down to one second.

It is the unavailability of any open-source solution that fits these requirements which prompted the development of our own piece of software, adb.

2.2.2 Other Design Goals

Beyond the functional requirements, there are particular features that are of secondary importance, but should be kept in mind during the selection or development of the solution. These goals are not strict but simply guidelines that will affect design choices and effort allocation.

The first such goal is minimal software complexity. The use cases are simple but rather broad, so it is in our interest to identify a small set of functionalities, and leave out any support for specific scenarios but allow for their inclusion at higher levels. This will also save implementation time, but might instead increase the difficulty in planning of the architecture.

Natural results of restricted functionality and complexity are various software size metrics: binary size, memory footprint, and especially number of software dependencies. Since adb is meant to be used in robotics, it is desirable to make it easily deployable on even the most rudimentary platform. In this specific case, we expect the core implementation to require a basic file system, a *NIX kernel (Linux or BSD), and a C runtime.

Finally, for the benefit of other researchers and to facilitate the extension of this platform, all its components should be made available under a liberal, open-source license.

CHAPTER 3
Theoretical Discussions

Before we begin a deeper analysis of the different issues pertaining to this specific database model, a few formalizations and definitions need to be set. As time-series databases can be classified as a subset of temporal databases, we will rely on vocabulary already agreed upon and defined in the consensus glossary of temporal database concepts [23]. For instance, we will refer to an arbitrary representation of time as a temporal element. Other concepts will be defined as necessary when they first appear in the discussion.

We will also introduce concepts specific to time-series database, and clarify the time-series model, which is comparatively more constrained, and therefore simpler than the more widespread relational model.

3.1 The Time-series Database Model

A time-series database enables one to store values of one or more observed processes as they evolve with time, in a sense capturing in a dynamic fashion the temporal dimension of these processes. Mathematically, a process is modeled as a variable, and we use this name to designate the collective information pertaining to that process in the database. Since the processes in question may be of any nature and produce any type of value, we wish to impose no constraints on their interpretation by the database. Formally, the possible values are simply elements of a set, with no other constraint imposed.

A time-series variable correlates process values with the time at which these values occurred. This pairing between a temporal element and a value is called a time relation, and in the specific case when the time models the world, it is called a valid-time relation [23].

An alternative model to pairs of time and value is that of a mapping from the time domain to the process' value domain. We will call the time domain T, the value domain V, and this mapping p (Equation 3.1).

$$p : T \to V \tag{3.1}$$

In this fashion, to find the value of the variable at a particular point in time t, one simply needs to apply the function p to it. This operation is the simplest form of a query. In relational database systems, a query is understood as any operation we can request from the system, sometimes only restricted to operations that do not alter the data. In our time-series context, we define a query as an operation that retrieves a variable's values from the database.

In its general form, a query takes a set of temporal elements, and returns a set of values. It is therefore another function, this time one that maps values from the power set of T to the power set of V. The power set of a set, denoted $\mathcal{P}$, is the set of all subsets of the given set. In other words, we are considering here the domain of all possible arrangements of temporal elements, the domain of all possible arrangements of values, and defining the query as a mapping between the two. Equation 3.2 expresses the formal mathematical definition of such a mapping.

$$p : \mathcal{P}(T) \to \mathcal{P}(V) \tag{3.2}$$

The database is not unique, so the query function must also have knowledge of the relations and we should pass it as an argument to the function as well. As Equation 3.3 indicates, the result of the query are all values satisfying the variable time-to-value mapping for all given temporal elements.

$$\begin{aligned} q &: (T \to V) \times \mathcal{P}(T) \to \mathcal{P}(V) \\ q(p, T') &= \{p(t) : t \in T'\} \quad \text{where} \quad T' \subseteq T \end{aligned} \tag{3.3}$$

The other main operation on time series data is the introduction of new time and value relations. Compared to relational databases, this operation is similar to an insertion from the implementation's point of view, but from a semantics perspective, it resembles an update. To avoid confusion between these previous concepts, we shall call it "augmentation", but will refer to the derived action in the database implementation as an insertion in later sections.

An augmentation is the creation of a new relation between some temporal elements to some value. Ordinarily, we want to enforce the conditions that the temporal elements be not part of an existing relation, and that all of them be strictly greater than all related temporal elements. In this fashion, we do not change history, and no information is destroyed. While it is rather easy to see how a query can be formalized as a function, an augmentation has side-effects and modifies the set it is executed on. It can also be represented as a function,

but one that transforms one state of a database to an new state which features the new relations. Since we expressed the relations themselves as a function p, an augmentation can be thought of as a mapping from a relation set, a temporal set and a value to a new relation set. Equation 3.4 illustrates this idea.

$$
\begin{aligned}
r &: (T \to V) \times \mathcal{P}(T) \times V \to (T \to V) \\
r(p, T', v) &= p' \quad \text{such that} \\
p'(t) &= \begin{cases} v & \text{if } t \in T' \\ p(t) & \text{otherwise} \end{cases}
\end{aligned} \tag{3.4}
$$

A noteworthy observation is that only a single element v from V is associated with many potential t elements in the subset T' of T. We shall see in the next section that while we must keep V general, the specific characteristics of time and temporal elements allow us to build an algebra around the set T and give us useful mechanisms for query expressions.

3.2 Temporal Algebra

In the previous section, we formalized the time-series database semantics, leaving out the specifics of the time and value domains. In this section, we will develop the mathematical framework which allows us to manipulate and reason about time. A formalization of these concepts becomes necessary when building a reliable and expressive query engine for a temporal database.

3.2.1 Time Instants

Humans experience time as one continuous dimension: from "now", we can only go forward or backward, and there is an infinite number of "points in time" between "now" and "then". Time, in this frame of understanding, is much like the other physical dimension in the sense that it can be mathematically expressed with the help of real numbers, and many temporal concepts have equivalents in real algebra.

The points in time referred to earlier will be called instants, as per the definition in [23]. Instants are elements of a totally ordered set, i.e. that given two distinct instants, one will be strictly greater than the other. We can therefore define the comparison relation, ">" on set T.

This ordering permits us to define time intervals, which are semantically similar to real intervals. Thanks to this property, we can reuse much of the concepts developed in the field of interval algebra [33] and have a direct way of reasoning about the temporal intervals. Their formal definition is given by Equation 3.5.

$$\begin{aligned} [t_1, t_2) &: t_1, t_2 \in T, t_2 > t_1 \\ [t_1, t_2) &\equiv \{t : t \in T, t_1 \leq t < t_2\} \\ [t_1, t_2) &\subseteq T \end{aligned} \tag{3.5}$$

To simplify our discussion, we shall only consider left-half-closed intervals, i.e. those that only include the left bound but not the right bound, and denoted here

by an opening square bracket and a closing round bracket. This is consistent with our understanding of event occurrence: an event happening from time t_1 to time t_2 is most likely understood to have finished, therefore not to be defined anymore at time t_2. Time intervals are subsets of T.

3.2.2 Time Spans

Intervals are a natural way of expressing events that are not instantaneous (do not happen at a single instant), as most events are not happening at a set of disjoint instants but rather at continuous subdivisions of the set T. A useful measure of an interval is its duration, which can be understood as the distance between its bounds. Equation 3.6 defines the time difference operator, $\boxminus$ and Equation 3.7 gives the definition of the duration of an interval. In [23], the span is defined as the directed duration of time. We shall reuse this definition here, and keep the term duration to designate only non-directed durations, or the property of an interval.

$$\boxminus \quad : \quad T \times T \to S \tag{3.6}$$

$$\mathit{duration}([t_1, t_2)) \quad \equiv \quad t_2 \boxminus t_1 \tag{3.7}$$

The duration of an interval is a span, an element of a set S, different from T and with its own semantics. Intuitively, we reason that spans can be combined through addition and subtraction, and if we think of span as a signed duration, it possesses an additive inverse relation. S is closed under all these operations, but as we see from Equation 3.6, its elements can also be synthesized from elements of T.

But we can also use elements of S to create new elements in T from other elements in T. These operations, called time shifts, are represented by the operators $\oplus$ and $\ominus$ and defined in Equations 3.8 and 3.9. As spans are directed, a negative shift by a span is the same as a positive shift by the inverse of that span, as expressed in Equation 3.10. Finally, Equation 3.11 expresses the dual relationship between the time difference $\boxminus$ and the time shift $\ominus$.

$$\oplus \quad : \quad T \times S \to T \tag{3.8}$$

$$\ominus \quad : \quad T \times S \to T \tag{3.9}$$

$$t \ominus s \quad \equiv \quad t \oplus (-s) \tag{3.10}$$

$$t_1 \boxminus t_2 = s \quad \iff \quad t_1 \ominus s = t_2 \tag{3.11}$$

We observe that an interval can be defined with either its two bounds or with a bound and a span (typically the lower bound and the interval's duration). Thanks to the two operators, we also can define instants relative to other instants, a semantic particularly useful in queries.

3.2.3 Time Operations

The sets S and T and their related operators form the basis of our temporal algebra, and we can use it to reason about time slices in query operations. Indeed, we want to be able to express an arbitrary subset of times at which we are interested in the value of a particular variable. The algebra will allow us to express these slices, and potentially simplify them in the query engine. We can establish

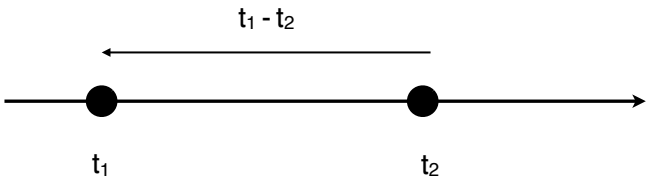

Figure 3–1: Vector representation of temporal concepts

four additional equalities that can be used for the algebraic manipulation of temporal values.

As is the case for the difference operation on real numbers, the time difference on instants is anti-commutative (Equation 3.12). The intuition behind this law is that a time difference produces a span, which is directed, and can be thought of as a one-dimensional vector in time. Swapping the operators reverses the direction of this vector. In Figure 3–1, instants t_1 and t_2 are points on the one-dimensional timeline, and the span $t_1 - t_2$ is a vector connecting them.

A few other laws have their correspondents in one dimensional vector algebra. Equation 3.13 is similar to the rule for summing vectors, and we can again intuitively compare spans to vectors and instants to points. Similarly, in Equation 3.14, we can think of vectors and imagine that displacing an instant by two spans in sequence is the same as displacing it once by the sum of those two spans.

Finally, Equation 3.15 states a rearrangement law for instants and spans. It embodies concepts of associativity and distributivity. We have two instants, t_1 and t_2, each displaced by a span, and we are looking for the time difference between the two. Once again, we can geometrically show that the result is the same as the time difference of the two instants displaced by the difference of the two spans.

$$-(t_1 \boxminus t_2) = t_2 \boxminus t_1 \tag{3.12}$$

$$(t_1 \boxminus t_2) + (t_2 \boxminus t_3) = t_1 \boxminus t_3 \tag{3.13}$$

$$(t \oplus s_1) \oplus s_2 = t \oplus (s_1 + s_2) \tag{3.14}$$

$$(t_1 \oplus s_1) \boxminus (t_2 \oplus s_2) = (t_1 \boxminus t_2) \oplus (s_1 - s_2) \tag{3.15}$$

These four rules demonstrate that there exists a correspondence between the operators $\ominus$, $\boxminus$ and $-$. Indeed, if we use a numerical representation for both instants and spans, the algebra becomes trivial, but it is useful to separate these concepts and keep thinking about elements of distinct domains.

To complete our understanding of this temporal algebra, we can state a few rules regarding intervals as well.

Equation 3.16 states that an interval whose bounds are identical should be treated as empty. This should make sense intuitively, since the interval's duration is clearly 0. We could have also decided this interval to be equal to a set with the single instant t, but since we are dealing here with left-closed, right-open intervals only, we want to preserve the constraint that the right bound is never part of the interval.

We introduce two rules about interval composition through set unions. The first, stated in Equation 3.17, shows that an instant will be "absorbed" by an interval during a union if it is located between its bounds. The second concerns partially overlapping intervals, and predicts that the union of two such intervals

will be bounded by the earliest instant on the left and the latest instant on the right.

$$[t, t) = \varnothing \tag{3.16}$$

$$[t_1, t_2) \cup \{t_3\} = [t_1, t_2) \iff t_1 \leq t_3 < t_2 \tag{3.17}$$

$$[t_1, t_2) \cup [t_3, t_4) = [\min(t_1, t_3), \max(t_2, t_4)) \iff t_1 < t_4, t_3 < t_2 \tag{3.18}$$

3.2.4 Discrete Time

The previous theoretical analysis assumes continuous time and processes, but recording the value of these processes digitally will inevitably produce a discrete representation of time. This reality somewhat alters the interval algebra used in previous sections, and new concepts need to be introduced.

Discrete time means we have a minimum distance between two distinct temporal elements, and also a smallest time span we can express. As defined in [23], the granularity is the measure of coarseness of a discrete representation, and the chronon is the smallest unit representable. In other words, in a system with a 5 milliseconds long chronon, we can achieve granularities down to 5 milliseconds.

Time spans are in this case composed of an integer number of instants, each separated by at least one chronon. In general, a sequence of instants separated by an arbitrary number of chronons is what we call a time-series. If the number is constant between two instants we introduce the concept of a granular interval.

Equation 3.19 gives its representation and mathematical definition, where g is the granularity of the interval.

$$[t_1, t_2)_g \equiv \left\{ t_1 + n \times g : n \in \mathbb{N}, n < \frac{t_2 - t_1}{g} \right\} \tag{3.19}$$

Granular intervals behave somewhat differently from regular intervals. Equation 3.17 does not hold, and Equation 3.18 works only for intervals with the same granularity.

The discrete time model also affects the query operator defined in Equation 3.3. We are dealing with a series of key-value pairs where the keys are discrete time instants, and it is now possible that the precise requested instants are not present in the series. The assumption we must make is that the values are valid between these discrete times. More specifically, each recorded instant marks a change in the value, and the process keeps the value recorded at that instant until the next recording. Equation 3.20 gives the formalization of this principle.

$$\begin{aligned} r &= \{(t_1, v_1), (t_2, v_2), ..., (t_n, v_n)\} \\ p_r(t) &= v_i \quad \forall t \in [t_i, t_{i+1}) \end{aligned} \tag{3.20}$$

Using both instants and spans, we can finally formulate the last definitions regarding queries on discrete time:

$$q_r(\{t\}) \equiv \{p_r(t)\}, \tag{3.21}$$

$$q_r([t_1, t_2)_g) \equiv \{p_r(t_j) : t_j \in [t_1, t_2), t_{j+1} - t_j \geq g\}, \tag{3.22}$$

$$q_r(T \cup T') \equiv q_r(T) \cup q_r(T'). \tag{3.23}$$

3.3 Time Representations

Time is, according to our modern understanding of physics, one of the dimensions of space-time, and supposedly not that different from the other three dimensions. Its particularity, that we experience it by linearly traversing it in one direction, which in turn limits our ways of effectively measuring it, has compelled us to treat, represent and reason about it in very different terms than almost every other dimension we know of.

The immediate observation we can make is that, unlike dimensions like length, mass or energy, time does not have one dominant, widely accepted and used standard of measurement. While the SI has defined standard metrics, useful in common life as well as in a scientific setting for most dimensions, time has no less than seven standardizations, most of which are not universal for all possible temporal representations, and none that has absolute advantages over all others [15].

3.3.1 Universal Time, Coordinated

The predominant one is UTC, or Coordinated Universal Time. It forms the basis of most of civil time keeping and is used in many operating systems, the network time protocol, radio, television and cellphone time systems. It is based on the usual way of defining a year as a complete rotation around the Sun, a day as a complete self-revolution of the Earth, and the usual integral subdivision of the day in hours, minutes and seconds. In fact, most time standards started from

these definitions, but the increase in precision of time keeping equipment revealed these few simple rules to be contradictory. The ratio of the Earth's revolution around the Sun to its revolution on itself is not an integer, it is not even constant. Neither are the revolution durations themselves. When two contradictory sets of definitions conflict, one must be relaxed for a standard to be well-defined. In the case of UTC, it is the number of days in a year, and the number of seconds in an hour that had to be kept non-constant. Indeed, besides leap years compensating for the fractional ratio of year to day, UTC introduces the concept of leap seconds to further compensate for smaller variations. Leap seconds are seconds periodically either added to or subtracted from an hour to insure a UTC year is still within a second of being the exact time it takes for a complete solar revolution, and that a day stays within a second of a complete Earth revolution. Unfortunately, indeterminate processes and uncertainty dictate when these leap seconds are necessary, and so they cannot be predicted in advance. Thus, it is a standards body, the International Earth Rotation and Reference Systems Service, that dictates, as time progresses, when these leap seconds should be introduced.

We require a representation that is primarily used in a scientific and computational context. Further, the particular requirement of the time-series model, i.e. that all events have valid times increase monotonically should also affect our choice of representation. UTC seems inadequate here. First, the presence of leap seconds, and therefore of minutes that can be 59, 60 or 61 seconds long would force us to use lookup tables to compute precisely the time difference between two values. Since our desired granularity can go down to a microsecond, an uncertainty of a

million times that unit is not acceptable. The use of lookup tables, while giving the right answer, would needlessly slow down computations that will be very common in a temporal database. Second, leap seconds make some time representations ambiguous (inside positive leap second) and others not corresponding to physical time (inside a negative leap second). This violates monotonicity, and imposes the establishment of corrective measures when one of these edge cases is encountered. But the most problematic issue remains that of computing future times. Under UTC, it is simply impossible for us to get the exact number of seconds that will elapse from now to a date and time 10 years from today, since leap seconds are yet to be announced for that period, and can't be predicted. UTC may be an adequate system for civilian time, and applications that don't require precisions exceeding a second, but for our real-time systems, it is simply unusable.

3.3.2 Terrestrial Time

In the context of very precise time keeping such as our time-series database, it seems necessary to look into time standards that are used in the scientific community. Terrestrial Time, often abbreviated TT, is the current standard used by the International Astronomical Union for observations from the Earth's surface. Defined in 1976, and originally called Dynamic Terrestrial Time, TT is very similar to UTC but, perhaps ironically for an astronomical time standard, it is not based on the Earth's rotation. Instead, it is based on the SI second, which is derived from a specific frequency of the caesium atom. This definition of the second was selected because of its closeness to the Ephemeris second, the empirically calculated mean of the duration of the second as one 86400^{th} of the

Earth's revolution, and its measurement through the use of atomic clocks. The International Atomic Time, or TAI, is a time standard that uses hundreds of such clocks in coordination, and in practice, TT is simply computed as an offset from TAI. UTC is also computed from TAI, but the offset changes as leap seconds are added. TT, to insure linearity, does not feature these leap seconds.

TT does not suffer from the drawbacks we identified in UTC: it is strictly monotonic, predictable, the non-contextual [1] units from the Gregorian calendar (minute, hour, day and week) are constant multiples of the base unit, which is also constant. The exact difference between any two values expressed in TT can be easily obtained with an arithmetic operation. TT has only been defined from 1976 and TAI only been kept since 1958, but due to the deterministic nature of a linear time scale, TT can be easily extended to prior dates. This hints at an easy and natural encoding of TT-represented instants: an integer count of chronons (we chose our chronon to be a microsecond), from an epoch, a specific instant in History. The specifics of the implementation, the choice of the epoch and ways to reconcile this system with the internal representation of time on commodity hardware is discussed in the Appendix.

[1] The month and year in the Gregorian calendar are inherently "contextual", or variable, since their exact length depends on when in time they happen. A month's length depends on which month we refer to, and a year's length depends on whether it is a leap year or not. The span of a minute is always defined as 60 seconds. This variability is different from the leap seconds introduced in UTC: leap seconds don't belong to a specific minute, but are placed where convenient, and their computation is non-deterministic.

3.4 ACID Properties in the Time-series Model

A very important requirement of relational database systems is to satisfy the ACID properties: Atomicity, Consistency, Isolation and Durability. These aspects are mainly relevant in the context of transactions, i.e a composition of several primitive operations. Taken together, these principles ensure that the whole transaction is performed or not at all (atomicity) and that as a result, the database cannot be in an inconsistent state (consistency). Furthermore, all transactions are independent and should not affect one another (isolation), and all these properties hold true even in the case of system failure such as power loss (durability) [19]. In a time-series model, guaranteeing these properties is relatively easy: since history is immutable, values can only be modified at the tail of a series, so writers need to take their turn and are therefore isolated. Transactions can only be multiple sequential writes, so atomicity can be satisfied by recording how many writes are requested, and rolling back on failure. Consistency and durability are likewise much easier, as checks need only be carried out at the tail of the time-series, and one can assume that history can never be corrupted, since it is immutable.

3.5 Data Opacity and Aggregate Operations

3.5.1 Arbitrary Type Storage

Most database systems provide a set of primitive data types to represent the information they are storing, along with accompanying operations on these types. More complex types are built as aggregates (collections or compositions) of these primitives. In relational databases, a collection can be represented as a table,

whereas a composition is a row with multiple fields. In NoSQL databases, most often the specifications of the underlying XML document define the possible types. Even in the time-series databases we have studied, there exists type information, although it is limited to only one representable type: the key is obviously a temporal type, while the value is always a double-precision floating point number. Although generic enough to allow the user to exploit them to his or her own needs, these types systems reflect their primary problem domains of intended use: business applications, web applications and diagnostic systems.

One of our requirements is to allow arbitrary data to be stored. Most of the systems mentioned before do allow for completely arbitrary data in the form of a specific binary array type, often called "blob". However, this is a case of "making the uncommon case possible", and the blob type does not have the same facilities available as the other primitive data types. Our problem deals primarily with numerical data, but also data of completely unknown structure. It is with a concern for simplicity that we chose to simply not worry about type information at all in the core implementation, and instead opt to offer the means to users of the database to construct their own type systems.

3.5.2 Filter, Map and Reduce

Being aware of the type of the data stored allows a DBMS to perform manipulation operations useful in queries. The choice to not have type information would therefore appear to remove that facility and seriously restrict the richness and potential of queries to only partitioning based on time. While this is sufficient in the context of a low-level API, a self-contained query system would require

the ability to express every type of query. These queries would be constructed in a query language capable of "asking" anything from the database, without necessarily being Turing compete.

As time-series databases' models are that of an ordered set of key-value pairs, a universal query language would be computationally equivalent to a list manipulation algebra. Fortunately, semantics of list manipulations are well understood, and are often a core component of functional programming languages. For the purposes of data queries, we are particularly interested in three basic list manipulations: filtering, reduction and mapping [6].

Filtering is the process of selecting elements from a list based on a predicate, the predicate being a function that maps an element of the list to either true or false. A filtering operation transforms a list into a new list containing only the elements of the first list that satisfy the predicate. It is a rough correspondent of the selection operation in relational algebra, or the WHERE clause in SQL. Equation 3.24 gives a formal mathematical definition of the filter operation, represented by the symbol "$\triangleleft$".

$$pred \triangleleft L = \{l : l \in L, pred(l) = true\} \tag{3.24}$$

The second most useful operation for querying is the reduction. A reduction is an operation that transforms a list to a single number, obtained by the successive application of some binary operator, or binary function, to the elements of the list. In practice, this is usually achieved by assuming the operator is associative, and the evaluation is executed by feeding the result of the previous operation

as an argument to the next. This argument is called the accumulator, since it accumulates the partial result of the reduction.

Reductions are useful to obtain aggregate results, such as the sum of all elements. In relational databases, aggregate functions perform reduction, but they are usually restricted to a predefined set. A formal definition is given in Equation 3.25, where "/" is the reduction operator, and $\oplus$ is an arbitrary binary operator used in the reduction.

$$\oplus/[l_1, l_2, ..., l_n] = l_1 \oplus l_2 \oplus ... \oplus l_n \tag{3.25}$$

Finally, the map operation completes the basic set of list operations. It transforms every element in a list to some new value according to a given mapping function, as shown in Equation 3.26.

$$fn * L = \{fn(l) : l \in L\} \tag{3.26}$$

These three operations can be used to transform the relation set, but their usefulness extends beyond this first level. Indeed, as we mentioned previously, the only constraint on the value domain is for it to be representable in computer memory, and we can treat it as a list of bytes, where the mapping, filtering and reduction are also defined. We therefore have two levels of list manipulations, unified under one semantic basis. This model gives us simultaneously the expressiveness that we require and the simplicity that we aimed for.

CHAPTER 4
Implementation and Experiments

4.1 Software Architecture and Components

As per the software requirements established earlier, the adb system needs to accommodate a wide variety of use cases and platforms. It needs to scale from very constrained embedded environments with limited memory, restricted operating system and a frugal set of libraries and tools to a network of commodity personal computers with distributed computation requirements. Such constraints would be unreasonable for a single piece of software, but may be achieved using a layered model of increasingly complex components depending on one another. In fact, the approach in building adb was to start with a simple component adequate for use in an embedded environment with a minimal set of primitive operations which could be reused in higher-level systems.

The core of adb, referred hereafter as adbCore, is the first layer of functionality. It is entirely written in the C programming language, and depends only on the C Standard library and POSIX libraries. Since most processor manufacturers

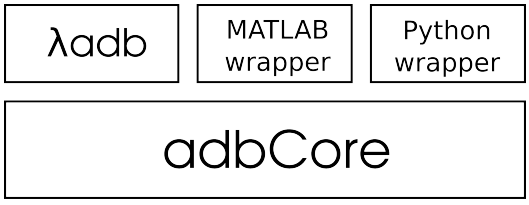

Figure 4–1: Software architecture of adb

ship their product with a standard but restricted C toolchain, porting adbCore to new environments should be trivial. As a rule, if a functional version of the BSD or Linux kernel can be ported to the new architecture, adb is guaranteed to work as well.

Since adb's primary purpose is to serve as a tool for researchers, an intermediate level API is provided on top of adbCore to integrate adb with numerical computation frameworks, currently MATLAB and NumPy through the Python programming language. This component should provide researchers who prototype in these environments with a quick solution for the persistence of their time series data.

Finally, at the highest level, λadb provides a command-line and network interface to the adb system through a data manipulation language. It offers additional organization and maintenance tools, allows for complex queries and data manipulations, and can be used to quickly analyze the data offline. λadb is written in the Haskell programming language.

4.2 Data Organization and Storage

4.2.1 Nested Variable Model

Variables stored in a time-series database are typically not related to one another, and can be thought of as completely independent entities. However, conceptually, it is often desirable to lump some variables under one category, and some under another category. For instance, we may have visual and range data under a 'sensor' category, and several motor rotation signals under a 'control' category. It is also possible to monitor separate but similar systems that share signals.

Still within the context of robotics, it is possible to think of two identical robots, each with its own sensors and control signals. The categories, or namespaces, can be nested.

An organization such as this already exists within each hierarchical filesystem, and in an effort to reduce software complexity, we can directly use a nested file organization within our database. We chose to use this method for a higher level of organization, within λadb. For adbCore, we simply forgo the concept of a database completely, and treat variables independently, designating them simply by their filesystem path.

Data Blocks Partitioning

We can use the available filesystem to further subdivide a variable into data blocks. Other database systems usually feature their own custom block organization within one file of a filesystem, for increased efficiency. However, to limit dependencies and reduce software complexity, it should be sufficient to use a regular filesystem to manage data blocks, especially since historical data is immutable, which limits writes to a single block.

At the low level, the user can specify if a new block should be created when they open a variable for writing. This gives optimal control over data partitioning. When choosing block sizes, the most space conservative strategy is to have blocks correspond to an integer number of filesystem blocks, but such a scheme may not be the most efficient when performing insertions or queries. In λadb, the block size is chosen automatically, but this choice can be changed in local settings.

Table 4–1: Execution complexity for the stamped storage mode

Operation	Execution complexity
Insertion	O(1)
Removal (undo last insertion)	O(1)
In-block search	O(log N)

4.2.2 Storage Methods

Time-series databases by their very nature grow quickly over time until some sort of data compacting is performed. This behaviour is very different from other types of databases that support destructive updates: the size of the database grows with the complexity and size of the model being stored. In this second situation, the concerns for space and speed efficiency are a matter of scale, while in the case of time-series data, they are more related to the lifetime of the application. Fortunately, the nature of a flat, time-series database structure and the physical variables it models allows for some constraints that simplify and improve the efficiency of storage, but these constraints are not always applicable. Consequently, we need more flexibility in choosing the storage modes for particular variables. The adb system at its current stage supports three such modes, and lets the user decide which is best for a particular application.

The Timestamped Storage Mode

The default storage mode used by adb is called "timestamped mode", or "stamped mode". It associates exactly one 8 byte long timestamp value to each data value, also of fixed size. Thanks to the constraint of monotonically increasing keys (the timestamps), and fixed size of keys and values, a very simple, sequential storage strategy similar to early systems like dBase II can be used.

Table 4–1 lists the computational complexity, in Big-O notation, of the primitive database operations. Insertion and removal always happen at the end of the series, so they complete in constant time and this behavior is true for all forms of storage. Within a block, a search is done using the keys, which are already ordered, so the execution complexity is logarithmic.

The Time-slotted Storage Mode

When storing data produced by a sensor, it is often possible to assume a constant production rate, and therefore a constant time separation between successive values. Under such circumstances, it is no longer necessary to keep each timestamp and we could thus save 8 bytes per entry in storage space. The "time-slotted mode", or simply "slotted mode" works in this fashion. Conceptually, the timeline is divided in equally spaced slots, each of which holds a value. This model allows not only size but also execution optimizations. It is particularly useful for small data values (with respect to the size of the timestamp), and should be used whenever the assumption of constant production rate holds, with possibilities of data steam interruptions. Like the stamped mode, data size needs to be constant in length. Table 4–2 shows the computational complexity for the slotted mode. The only difference from the stamped mode is that searches are now constant, since there is a mathematical relationship between the key and the offset of the data within a block.

A Mode for Data of Variable Length

In the primary use cases of adb, the fixed size of the data is a valid assumption. Most sensors measure variables that are typically either scalars or vectors of

Table 4–2: Execution complexity for the slotted storage mode

Operation	Execution complexity
Insertion	O(1)
Removal (undo last insertion)	O(1)
In-block search	O(1)

Table 4–3: Execution complexity for the text storage mode

Operation	Execution complexity
Insertion	O(1)
Removal (undo last insertion)	O(1)
In-block search	O(log N)

values whose range is bounded. It is possible however that these assumptions do not hold, especially if we generalize the purpose of adb. For example, a message logging system can be thought of as a time-series of string values, yet there isn't any consistency in the length of the string, and while it is possible to impose large enough upper-bounds and treat these strings as fixed-size values, it would be done at the expense of space.

A third storage mode will be made available in adb: the string or text mode, which allows for the storage of variable-size data. Since one of the main directions of evolution in databases in the last decade has been towards a less constrained data structure, often expressed in a data description language such as XML or JSON, the support for a text-based storage mode seems highly beneficial. Areas of application include storage of data expressed in these formats, as well as textual diagnostics and logging systems. Large sparse matrices may also be stored more efficiently in this mode.

4.3 Programming Language Support

As our target demographic is primarily confined to the domain of scientific computation, the programming language and library support should reflect the current tools used in this sector. For prototyping, the MATLAB environment is the most widely used toolset. However, the Python programming language and its numerical libraries, SciPy and NumPy, are gaining traction as well.

4.3.1 C API

The lowest level component of adb, adbCore, is implemented in C. The higher layers must access its functionalities through some API. In some contexts, such as that of embedded systems, or development where the main application is also implemented in C or C++, the developer might also want to directly access these low-level features. The API of adbCore reflects these two needs, and prioritizes simplicity and small API count over control and expressiveness.

Currently, the C API of adbCore is composed of nine functions, five custom types and about twenty macros, all declared in one header file. This simplicity provides enough flexibility to accomplish the basic database tasks (insertion, removal and query) while being straightforward to learn.

The functions provided all operate on one variable. To create a variable, one must first create a directory on the filesystem, then create an `adb_var` structure, store appropriate values in its fields, and finally invoke `adb_open`. One of the fields of `adb_var` is `opmode`, which specifies one of the three modes of operation: write mode, read mode and clean mode. A new variable is created in the write mode, which also allows for insertions of new records. Read mode is used to perform

queries on the data, while clean mode is used for compacting. Each mode gives access to different functions and treats the time-series as a specific data structure. When all desired operations are completed, `adb_close` should be invoked on the variable.

In write mode, only the `adb_insert` and `adb_remove` functions are available, and the time-series behaves like a stack: `adb_insert` appends a new value at the end of the series, `adb_remove` deletes the last inserted value. Before `adb_close` is invoked, there is no guarantee the data is committed to disk. In addition, only one write mode handle can be open at one time, so it is recommended to use burst writes: new values should be added to a buffer, and a background thread should open, insert and close the variable whenever the buffer becomes full.

In read mode, the available functions are `adb_next`, `adb_prev`, `adb_key` and `adb_value`. The time-series behaves like a doubly linked-list: a starting timestamp, or key is given to the open function through a field in `adb_var`, which places a cursor at that key. The `adb_next` and `adb_prev` functions can then move the cursor forward and backward in time, while `adb_key` and `adb_value` can be used to retrieve the key and value at the cursor respectively.

In clean mode, the same functions are available as in read mode, except for `adb_prev`, which is replaced by `adb_remove`. The data structure comparison resembles a singly linked-list this time: the cursor cannot move back, but instead, the remove function can be used to discard the entry at the cursor and moves the cursor to the next entry. No changes are committed before a call to `adb_close`, which can only proceed when all other handles have been closed.

4.3.2 Matlab and Python API

MATLAB and Python are programming languages and environments that differ considerably from C: they are both dynamically-typed, memory-managed, have higher-level constructs and libraries for dealing with vector and matrices and can be Just-In-Time compiled for a virtual machine or directly interpreted for rapid prototyping. These properties make them more suited for academic work, where researchers often have little experience in building software systems. Given adb's target demographic, it is almost essential these two platforms get full programming support.

For adbCore, one of the requirements was the availability of the system on embedded devices with limited memory, processing power and library support. In the case of MATLAB and Python, we are reversing these assumptions: both have large libraries and dependency requirements, have to run on powerful machines, and for numerical computations, it is already assumed large amounts of memory and processing power will be required. This reality, along with the expressiveness of both languages, allow for a different approach to an API and for further simplifications.

```
>> A = [1 2 3;4 5 6];
>> A(2,1)
ans =
     4
>> A(1,2:3)
ans =
     2     3
>> A(2,1) = 10
A =
     1     2     3
    10     5     6
```

Listing 4.1: MATLAB array definition, indexing and slicing

```
>>> A = array([[1,2,3],[4,5,6]])
>>> A[1,0]
4
>>> A[0,1:3]
array([2, 3])
>>> A[1,0] = 10
>>> A
array([[ 1,  2,  3],
       [10,  5,  6]])
```

Listing 4.2: NumPy array definition, indexing and slicing

Both MATLAB and NumPy arrays have in-language support for easy indexing and slicing. Slicing is the technique of providing an indexing range to extract a sub-array from a given array. Indexing and slicing can be used to retrieve values or assign them, thus modifying parts of the original array. Listings 4.1 and 4.2

show the syntax used for these operations. [1] Arrays can have multiple dimensions, so it becomes natural to think of a time-series of arrays as one array with an additional temporal dimension. Indexing and slicing along that dimension become the substitute of the normal query, and insertion of new records can be achieved by setting values at new indexes. Listings 4.3 and 4.4 show a usage example, with both insertion and query.

```
>> T = adb_open('path/to/variable');
>> T('2012-09-04T17:00.000000') = eye(1,3);
>> T(1346778001000000) = ones(1,3);
>> [keys,values] = T(:)
keys =
     1346778000000000
     1346778001000000

values =
     1     0     0
     1     1     1
```

Listing 4.3: MATLAB time-series manipulations

[1] MATLAB starts indexing at 1 while Python, and therefore NumPy start indexing at 0. Furthermore, ranges are inclusive in MATLAB, but right-exclusive in Python. For example, `1:3` corresponds to `1,2,3` in MATLAB but `1,2` in Python. These semantics differences account for the index discrepancies in the examples given.

```
>>> T = adb.open('path/to/variable')
>>> T['2012-09-04T17:00.000000'] = eye(1,3)
>>> T[1346778001000000] = ones(3)
>>> keys,values = T[:]
>>> keys
array([[1346778000000000],
       [1346778001000000]])
>>> values
array([[ 1.,  0.,  0.],
       [ 1.,  1.,  1.]])
```

Listing 4.4: Python time-series manipulations

The selected approach has the advantage of reusing concepts and syntax that users of these environments are familiar with. Its simplicity has one main limitation: more refined queries, for example limiting results based on the values of array elements, need to be expressed in the language after all values have been retrieved, and thus, a larger amount of data than necessary may be read from the database, only to be discarded later by filtering. We justify this drawback by the assumption that systems written in MATLAB or NumPy, where large datasets need to be processed, have an expectation of high memory usage, and that familiarity and ease of use are of greater importance for researchers.

4.4 Universal Interface

The previous sections detail the most common way we believe adb's users will choose to interact with it. There are, however, a multitude of other scenarios where languages other than C, C++, Matlab or Python are used, or simply which require greater flexibility in the type of queries that need performing. In those

cases, a universal, high-level textual interface seems to be the most appropriate. Similar to relational databases and their Structured Query Language, adb also provides a data definition and manipulation language. It is similar in spirit to SQL but differs substantially in its semantics due to the specific nature of a time-series database.

4.4.1 Temporal Description

We strive to offer the user a means of expressing time values for use in queries and insertions in a natural way. Section 3.2 presented the foundation of our temporal algebra; in this section, its actual syntax in the universal interface is presented.

A very useful means of expressing time instants is to do so relative to the current time. The keyword `now` is used to designate the current time, and relative times are constructed by adding or subtracting spans to it, using the operators $\oplus$ and $\ominus$. These are simply written as $+$ and $-$ respectively in the language. Spans themselves are denoted as a composition of integers and time units. Since we are using the Terrestrial Time standard, all the common time units have durations that are integer multiple of others, from the microsecond to the week. Table 4–4 lists the value of units with respect to one another.

Spans are composed by juxtaposing the integers and time unit pairs, separated by spaces. Listing 4.5 gives an example of spans and instants.

Table 4–4: Time units and their values

Unit	Abbreviation	Value
millisecond	ms	1000 μs
second	s	1000 ms
minute	m	60 s
hour	h	60 m
day	d	24 h
week	w	7 d

```
2w 5d
now - 12h 45m
now + 1h 10s 50us
```

Listing 4.5: Spans and instants

We may also wish to use absolute instants. There are two ways to do so: using the ISO 8601 format or by specifying the integer representation of the Terrestrial Time encoding. Listing 4.6 shows the different formats understood by adb and it's values.

```
2012-09-01T14:28:42.123456
2012-09-01
T14:28:42
T14:28:42.123456
0x0000239834812635
```

Listing 4.6: Examples of instant literals

Intervals are defined using two instants, with an additional granularity if needed. We use an arrow operator (`->`) between two instants to specify an interval. A modified ternary arrow operator, composed of a double dashes and an arrow (`--` and `->`), can be used when one wants to specify the granularity. In this form, the operands are given in this order: lower bound, granularity, upper bound. Visually, it looks like an elongated arrow with the granularity in the middle. Both the binary and ternary form of the operator have lower precedence than addition and subtraction, so parentheses are usually not needed, but can be added for clarity. Listing 4.7 gives an example of a natural interval and a granular interval.

```
2012-09-01T14:28:42.123456 -> 2013-09-11T23:17:56.987654
now - 3h -- 10 ms -> now
```

Listing 4.7: Examples of intervals

For advanced temporal operations, it may be necessary to specify disjoint intervals, a set of instants, or any combination thereof. Both instants and spans are treated as time units, and can be combined with a comma operator. Mathematically, this operation corresponds to set union, where individual instants are implicitly wrapped around a set containing only them. The comma operator has the lowest precedence. Listing 4.8 illustrates the use of the comma.

```
now, now - 1s, now - 1m, now - 1h
2012-09-01T14:28:42 -> 2013-09-11T23:17:56, now
now - 24h -- 1h -> now - 1h, now -1h -- 1s -> now
```

Listing 4.8: Examples of combinations

With the help of these primitives, it is possible to easily describe the most common time values that one would like to retrieve in a query.

4.4.2 Data Manipulation

As a primarily time-oriented database system, adb does not support advanced data operations like other more traditional databases. Values stored are treated as a string of bytes and a query therefore must contain knowledge about the specific representation of the data. This low-level approach is simple and expressive, but it may make complex queries rather verbose. Since most database systems interfacing with data applications are known to have their queries generated by scripts, and only rarely written by hand by humans, we found it to be an acceptable trade-off.

In Section 3.5.2, we mentioned that queries can be expressed with the help of filtering, mapping and reduction operations. These operations take as arguments the value of the relation and a function. The data manipulation language must therefore be flexible enough to express a wide range of these functions. A filtering function for instance must return a boolean value, whereas a reduction function must be allowed to take two arguments: the current value, and the accumulated value, which is the result of the previous computation of the reduction function.

A detailed overview of the data manipulation semantics is outside the scope of this thesis. It should however be sufficient to describe its main principles and syntax. As list manipulations are its core mechanism, slicing and concatenation are some of its primary tools. This is done using indexing and concatenation operators. The following example takes two slices and concatenates them, which is equivalent of taking the value as is:

```
val[0:4] ++ val[4:end]
```

Listing 4.9: Example of slicing and concatenation

Another key operation is type casting. By default, the value is treated as an array of unsigned 8-bit integers (bytes), but it can be cast to other types to change the results of arithmetic and comparison operators. In the following example, we compare the first half of the value to the second half, where they were encoded as integer and real:

```
(val[0:4] :: int32) > (val[4:end] :: float64)
```

Listing 4.10: Example of type casting

Finally, for vector data, list operations are also very useful and powerful. Listing 4.11 shows how to keep all elements greater than 10, add 3 to all elements and sum all elements, respectively.

```
(>10) |> (val :: uint32)
(+3) *> (val :: float64)
(+) /> (val :: int64)
```

Listing 4.11: Example of filter, map and reduce

A list of types and operators is given in the Appendix.

4.5 Use Case Example

In this section, we will demonstrate how adb can be integrated in a ROS stack in order to enhance it with a capacity to access historical data. ROS uses a publisher subscriber model, with a centralized repository of all the channels through which data flows. Our strategy is to tap to a particular channel and store the values as they come, so they can be retrieved by some module at a later time. To stay true to the topic of robotics and computer vision, our example shall feature grayscale image at VGA resolution. In practice, since adb doesn't have knowledge of the type of data stored, and can store values of arbitrary size, we could have just as well used scalar values, or high-definition color images. Figure 4–2 shows a block diagram of the node layout of our system.

ROS nodes can be written in both C++ and Python. For this example, we wish to demonstrate adbCore's API, and will write the nodes in C++. For the recording node, we modified a simple subscriber node provided in ROS's tutorial page. The implementation is straightforward, and consists of two steps. First, we set up the database variable that will contain the image values: we create an `adb_var` structure, and populate it with information about storage location and

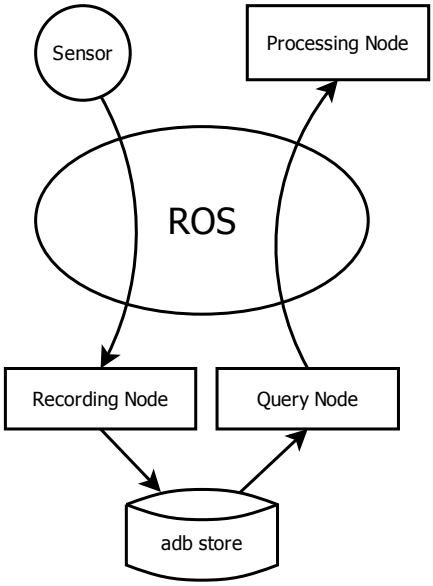

Figure 4–2: ROS stack with adb

variable type. Then in the callback, we open the variable, insert a record with a timestamp corresponding to the time of retrieval, and finally close the variable. The source code is available in Listing A–2.

The query node could be implemented in a number of ways: it could listen on a channel for specific timestamps, or a span and in response, extract the corresponding values from the database and publish them on another channel. It could also be an automated node that publishes historical values according to some rule. The code to achieve this behavior is similar to that of the recording node: a variable setup, followed by the retrieval of values that consists of an open-read-close. Together with the recording node, they can thus add a historical dimension to the live dataflow of ROS messages.

4.6 Benchmarks

We wished to compare adb to existing database solutions in terms of raw performance for both writing and reading. As mentioned in the introduction, there are, to the author's knowledge, no freely available systems which directly satisfy all our requirements, but we still needed a basis of comparison upon which to judge the capabilities of adb in the context of numerical computing. We chose, for this comparison, three other systems: SQLite, the ubiquitous embedded general-purpose relational database system, the RRD tool, a simple round-robin time-series storage solution, and TempoDB, a cloud-based time-series database.

All the systems tested are designed to store data directly on permanent storage like a hard disk drive. Since reading and writing to such a device are very slow operations, each system uses a caching mechanism to keep some of the records in memory. SQLite and the RRD tool use their own caching methods, tailored to their storage layout, and we can guess TempoDB's servers are configure with one as well. The caching systems we chose for adb is very simple: for writing, we let the C standard library use its own flushing schedule, whereas for reading, we map a block file to memory and let the operating system perform its own block caching. The following benchmarks are therefore primarily a way to compare the performance of these different caching strategies.

It should be noted that to increase reading and writing speeds, SQLite, the RRD tool and adb can keep their data in memory, but would then lose their permanent storage capabilities. It suffices to configure the file system to treat a portion of memory as a virtual hard drive, usually called a RAM disk. This

method is not optimal, as certain strategies will ultimately force the operating system to perform copies between two memory locations, but in adb's case, the memory mapping of files should allow the OS to access the data faster. These in-memory databases can be useful for very high-volume data, but are quickly limited by the computer's RAM capacity. For example, high-definition uncompressed 1080p color video takes up about 178 MB per second of footage, which means it takes just over a minute and a half of such footage to fill up 16 GB of RAM.

We required a testing strategy that would adequately compare all four systems, and since each uses a very unique approach, this task proved to be somewhat of a challenge. We could not compare in-RAM and disk-stored performance, since that would have left out TempoDB, whose storage is inaccessible. We already expect TempoDB to perform the worst, possible by several orders of magnitude, since its data access is limited by the network latency and throughput. Furthermore, both the RRD tool and TempoDB can only store floating-point values, so any comparison in storing vector values would be impossible. We therefore had to restrict our testing set to double-precision IEEE 784 numbers to compare all four database systems.

4.6.1 Insertion

For data stream captures, a particularly important point is the speed of insertion. We performed a value insertion stress test comparing our four database systems by measuring the execution time of a bulk insert command using different numbers of items. The particular method varied slightly from system to system, according to the available APIs, but we generally attempted to produce the values

as fast as the interface would allow it. In all cases, the data values were randomly generated.

In the case of adb, the function `adb_insert`, which performs a single update, was called in a loop, without delay. We used the result from a call to `adb_now` for the timestamp value. The TempoDB API allows bulk inserts directly, so for this system, a set of key-value pairs was constructed and stored in a data structure in memory, which was then passed to the insert command.

SQLite, being a general-purpose relational database with loose typing, does not directly support working with timestamps. It features time manipulation functions, but they operate on date-time values stored as strings. Since string parsing is usually a costly operation compared to integer arithmetic, and we are only interested in storing and retrieving values and not modifying them, we opted to store timestamp values as integers, follow the same representation adb uses internally. We also made them into an ascending primary key, to give SQLite the best conditions for performance. Like for adb, the insert statements were executed in a loop, without delay.

Finally, for the RRD tool, we constructed command strings with 10 key-value pairs at once, and these strings were executed against the `rrdtool` binary, also in a loop, without delay. In the particular case of the RRD tool, the timestamp values did not represent the actual time of insertion: the RRD tool does not support a resolution under one second, and updating the value of a variable sooner results in an error from the system. This can be circumvented with an additional software layer that interprets timestamps differently: they are, after all, integers,

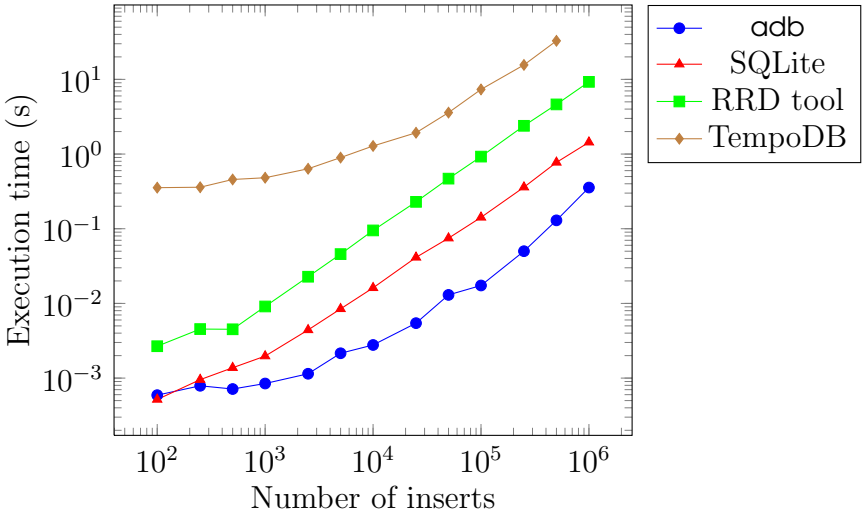

Figure 4–3: Insert performance comparison for scalar values

which granularity can be interpreted at a higher level. It is important to note this additional step needed, and the fact that the RRD tool does not offer this support directly.

The tests were performed on present commodity hardware, namely a desktop computer with an Intel Core i7 860 quad-core processor clocked at 2.80GHz, 4GB of DDR3 memory clocked at 1333MHz and a 500GB hard drive. We picked the number of inserts to be approximately evenly spaced on a logarithmic scale. The results of this benchmark are presented in Figure 4–3.

As predicted, we see that TempoDB is several orders of magnitude slower than the other three contenders, though the difference shrinks as the volume grows. Another unsurprising result is the mostly linear increase of the execution time. We observe that adb and SQLite perform comparably for a few hundred inserts, but adb becomes faster as the volume of inserts reaches the thousands. On the other

hand, the RRD tool, although advantaged by its constant size storage, performed worse than both SQLite and adb. We hypothesize this could be due to the nature of the RRD tool's interface: inserts need to be executed by individual function calls, and the calls take in strings as arguments. The additional calling and string parsing overhead would explain its relatively slow speed. It is also clear that the RRD tool was not designed with handling large volumes of data.

The finest possible granularity of both adb, and the representation we used for SQLite, is of one microsecond, or 10^{-6} seconds. We can see from Figure 4–3 that, given a high enough volume of inserts, both SQLite and adb are able to write records at this minimal granularity. The performance of adb is even more impressive, with an insert time between 6 and 0.3 microseconds per value, which translates to an insertion rate to up to 3 MHz. This speed should satisfy even the fastest capturing scenarios.

4.6.2 Query

Query time is of secondary importance in our particular problem domain, but in most other areas of application, it is actually the main metric the software is optimized for. In these cases however, the data one queries for is usually scattered around the storage data structures, and it is through clever organization and retrieval algorithms that high performance is achieved. We are mainly interested in queries that involve time slices, so the access to data that is already organized sequentially in time should only be limited by IO operations. Thanks to its simplicity, we expect adb to perform equally well.

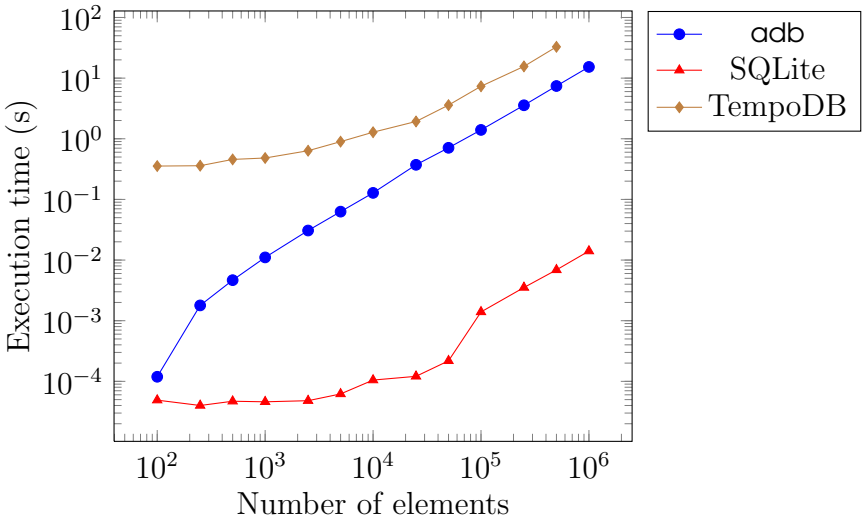

Figure 4–4: Query performance comparison for scalar values

For this test, we measured the execution of an aggregate function (sum) over n entries. This approach insures that the values are consumed without having to print them individually, where printing time would have introduced a significant bias in the results.

The tests were performed on the same setup: Intel Core i7 860 quad-core processor clocked at 2.80GHz, 4GB of DDR3 memory clocked at 1333MHz and a 500GB hard drive. Again, we picked the number of queries to be approximately evenly spaced on a logarithmic scale. The results of this benchmark are presented in figure 4–4.

We omitted the RRD tool from these tests because of the nature of this time-series database: the aggregate functions are to be defined at the start, and computed as the data is added. The results for this test would not be a measure of

a query execution, but only of the access to a precomputed value. Moreover, the time taken for such an access is significantly below our measurement resolution.

In this test, SQLite performs significantly better than adb. SQLite is indeed a database optimized for read access, and it is possible some optimizations allow it to compute aggregate function such as the sum faster. This is therefore an area where adb has ample room for improvement. We also see that TempoDB is still largely slowed down by the network latency and throughput, but its distance from adb's performance is reduced.

We conclude from these two tests that each of the databases has its strengths and weaknesses. Whereas SQLite and the RRD tool fare better in queries, adb retains an edge for large write volume scenarios.

CHAPTER 5
Conclusions and Discussions

5.1 Performance and Usability

We have seen that in terms of raw speed, adb performs admirably in comparison to other time-series databases, and even very widely used general purpose databases such as SQLite. In absolute terms, adb is well suited for the storage of very high-frequency data. Its high throughput and the separability of variables makes it a very good historical persistence solution for a CCN system.

The very few dependencies of the core part of adb, and its very small executable and memory footprint make adb a good choice for embedded systems that need a time-series recording component. In more feature-complete systems, the user has a choice of programming languages to interface with adb: C/C++, Python, Matlab and Haskell.

As a tool designed specifically to assist in numerical computing tasks, adb offers flexibility in data type, storage method and compacting strategies. To the best of the author's knowledge, there is no freely available tool with similar characteristics. While our results show that SQLite can show only slightly poorer performance, all time manipulation and knowledge must be opaquely implemented on top of the database system. Additionally, compacting of the database can be costly and halt all other operations while it is carried out.

5.2 Future Work

Whereas the core of adb is ready for use, further work is necessary on the Python and Matlab wrappers, as well as the higher-level interface and its data manipulation language. As more extensive use is made of adb and the need for even better performance arises, we can also look into refining the storage and caching model. While the mechanisms native to the operating systems and file systems used seem adequate, and it is unclear whether it can be optimized further, the particular nature of time-series data may offer a chance for better data management. Finally, users of adb may find specific features that they wish implemented, and depending on their generality, we may decide to introduce them.

5.3 Closing Remarks

We have created a simple yet powerful tool for storing arbitrary time-series data for numerical computing. It can be used as a natural historical persistence layer within the CCN, but can also be used as a standalone component in monitoring systems, embedded robotics applications or simply as a quick and easy persistence solution for researchers. Its features are unique amongst open-source time-series databases, and its performance rivals that of competitors. We hope that it will find adoption outside of the laboratory and project that it was originally designed for, and that it can make its mark in the field of robotics.

Appendix A

A.1 Internal Time Representation

The best method for representing instant time values, or timestamps, in digital form has been the subject of some studies, and different solutions are used in different circumstances. As it is usually the case with design decisions, the best solution depends on the area of applications, as different trade-offs need to be made. In the case of timestamp representations, issues such as range, precision and which operations are most frequent will dictate the optimal format.

Dyreson and Snodgrass [15] proposed a model to represent the entirety of time, with different representations and different precisions. Their work was intended for use in a general-purpose temporal database, and is consequently more complex than what is needed for a time-series database, where we can make the assumption that time values will be close to "now", and not extend too far into the future or the past. Nevertheless, their analysis of the different existing representations is very relevant to our pursuit.

They identified four types of operations the database system may need to perform on the timestamps. In order of frequency, and therefore importance, they are comparisons, arithmetic operations and conversion, and separated as "input" and "output" in their work. In our time-series database, their order stays unchanged. Comparisons are used frequently in almost all operations: finding elements, checking query boundaries, ensuring monotonicity on inserts to name a few. Arithmetic operations are particularly relevant in the time-slotted storage

scheme, since timestamps are implicit and their values need to be computed from offsets. Finally, conversions happen at only higher levels of abstractions, and only the most basic conversion utility is provided in the C API.

True to Dyreson and Snodgrass' analysis, but bearing in mind that we do not need to represent all of time, we opted for a format that extends the UNIX time representation from 32 to 64 bits, and resizes the chronon from the second to the microsecond. We could have picked any epoch, but for simplicity's sake and a slight ease of conversion, we kept it to January 1st, 1970.

A.2 C code listings

```
typedef int64_t     adb_time;
typedef int64_t     adb_store;

typedef enum {
    ADB_WRMODE, ADB_RDMODE, ADB_CLMODE
} adb_mode;

typedef struct {
    adb_store   storage;
    size_t      size;
    char        info[25];
} adb_type;

typedef struct {
    const char  *path;
    adb_mode    opmode;
    adb_time    start;
    adb_type    type;
} adb_var;

adb_time adb_now(void);

int adb_open(adb_var *var);
int adb_close(adb_var *var);

adb_time adb_key(adb_var *var);
const void * adb_value(adb_var *var);

int adb_insert(adb_var *var, adb_time key,
               const void *value);
int adb_remove(adb_var *var);

int adb_prev(adb_var *var);
int adb_next(adb_var *var);
```

Listing A–1: Header file for adb

```
#include <adb.h>
#include "ros/ros.h"
#include "std_msgs/Int8MultiArray.h"

static adb_var image_var;

void chatterCallback(
  const std_msgs::Int8MultiArray::ConstPtr& arr)
{
  adb_open(&image_var);
  adb_insert(&image_var, adb_now(), &arr.data);
  adb_close(&image_var);
}

int main(int argc, char **argv)
{
  // adb variable setup
  image_var.path = "./db/image";
  image_var.type.storage = ADB_STAMPED;
  image_var.type.size = 640 * 480 * sizeof(int8);
  image_var.start = ADB_NEWBLOCK;

  // ROS setup
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("adb_example",
      10, chatterCallback);
  ros::spin();
  return 0;
}
```

Listing A–2: Simple ROS recording node

Domain Width	8 bits	16 bits	32 bits	64 bits
signed integer	int8	int16	int32	int64
unsigned integer	uint8	uint16	uint32	uint64
floating point number	N/A	N/A	float32	float64

Table A–1: Primitive types and their attributes

A.3 Data Manipulation Language

At the time of writing of this document, the data manipulation language is still in its infancy. Its main points and format are somewhat established, but the specific features may change with different patterns of use.

Table A–1 gives current type literals that can be used when casting values from one type to another. Table A–2 shows a categorically sorted list of operators featured in the language.

A.4 Getting and Installing adb

The source code for adbis available in the author's public repository on Github. The following tools are needed to get, build and install the software:

- GCC or another compatible C compiler
- CMake
- Git

To install, open a terminal window in a temporary location and run:

```
$ git clone git://github.com/zykos/adb.git
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

Table A–2: List of available operators

Category	Operation	Symbol
Arithmetic	addition	`+`
	subtraction	`-`
	multiplication	`*`
	division	`/`
	remainder	`%`
Bit shift	left shift	`<<`
	right shift	`>>`
Comparison	equal	`==`
	not equal	`!=`
	greater than	`>`
	greater or equal	`>=`
	less than	`<`
	less or equal	`<=`
Bitwise	and	`&`
	nand	`~&`
	or	`|`
	nor	`~|`
	xor	`^`
	xnor	`~^`
Logic	and	`&&`
	nand	`!&`
	or	`||`
	nor	`!|`
	xor	`^^`
	xnor	`!^`
Other	concatenation	`++`
	type cast	`::`
	filter	`|>`
	map	`*>`
	reduce	`/>`

References

[1] T. Abraham and J.F. Roddick. Survey of spatio-temporal databases. *GeoInformatica*, 3(1):61–99, 1999.

[2] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. Deeds towards a distributed and active real-time database system. *SIGMOD Rec.*, 25(1):38–51, March 1996.

[4] C.W. Bachman. Data structure diagrams. *ACM Sigmis Database*, 1(2):4–10, 1969.

[5] D. Barry and T. Stanienda. Solving the java object storage problem. *Computer*, 31(11):33–40, 1998.

[6] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[7] L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang. Optimizing data analysis with a semi-structured time series database. In *SLAML10: Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques*, pages 7–7, 2010.

[8] Michael H. Böhlen. Temporal database system implementations. *SIGMOD Rec.*, 24(4):53–60, December 1995.

[9] M. Brosey and B. Shneiderman. Two experimental comparisons of relational and hierarchical database models. *International Journal of Man-Machine Studies*, 10(6):625–637, 1978.

[10] Norman Carver and Victor Lesser. Evolution of blackboard control architectures. *Expert Systems with Applications*, 7(1):1–30, 1994.

[11] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[13] C. Coronel, S. Morris, and P. Rob. *Database systems: design, implementation, and management*. Course Technology Ptr, 2012.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[15] Curtis E. Dyreson and Richard T. Snodgrass. Timestamp semantics and representation. *Information Systems*, 18(3):143–166, 1993.

[16] Shashi K. Gadia and Jay H. Vaishnav. A query language for a homogeneous temporal database. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '85, pages 51–56, New York, NY, USA, 1985. ACM.

[17] R.H. Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

[18] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *Knowledge and Data Engineering, IEEE Transactions on*, 6(4):572–586, 1994.

[19] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[20] B.C. Hardgrave and N.P. Dalal. Comparing object-oriented and extended-entity-relationship data models. *Journal of Database Management (JDM)*, 6(3):15–22, 1995.

[21] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Data access scheduling in firm real-time database systems. *Real-Time Systems*, 4:203–241, 1992. 10.1007/BF00365312.

[22] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.

[23] Christian Jensen, Curtis Dyreson, Michael Böhlen, James Clifford, Ramez Elmasri, Shashi Gadia, Fabio Grandi, Pat Hayes, Sushil Jajodia, Wolfgang KÄfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John Roddick, Nandlal Sarda, Maria Scalas, Arie Segev, Richard Snodgrass, Mike Soo, Abdullah Tansel, Paolo Tiberio, and Gio Wiederhold. The consensus glossary of temporal database concepts february 1998 version. In Opher Etzion, Sushil Jajodia, and Suryanarayana Sripada, editors, *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 367–405. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0053710.

[24] Ben Kao and Hector Garcia-molina. An overview of real-time database systems. In *Advances in Real-Time Systems*, pages 463–486. Springer-Verlag, 1995.

[25] W. Kim. *Introduction to object-oriented databases*. 1990.

[26] A. LITH and J. MATTSSON. Investigating storage solutions for large data. *Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden*, 2010.

[27] F. H. Lochovsky and D. C. Tsichritzis. User performance considerations in dbms selection. In *Proceedings of the 1977 ACM SIGMOD international conference on Management of data*, SIGMOD '77, pages 128–134, New York, NY, USA, 1977. ACM.

[28] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[29] E. Oomoto and K. Tanaka. Ovid: Design and implementation of a video-object database system. *Knowledge and Data Engineering, IEEE Transactions on*, 5(4):629–643, 1993.

[30] Richard Thomas Snodgrass. Temporal databases. *IEEE Computer*, 19:35–42, 1986.

[31] M. Stonebraker, D. Moore, and P. Brown. *Object-relational DBMSs: Tracking the next great wave*. Morgan Kaufmann Publishers Inc., 1998.

[32] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, April 2010.

[33] Teruo Sunaga. Theory of an interval algebra and its application to numerical analysis. *Japan Journal of Industrial and Applied Mathematics*, 26:125–143, 2009.

[34] Burghard von Karger. Temporal algebra. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 310–386. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-47797-7_9.

[35] W.W. Wadge and E.A. Ashcroft. Lucid, the dataflow programming langugage1. 1985.