TECHNOLOGIES FOR MAIN MEMORY DATA ANALYSIS

DISSERTATION FOR THE AWARD OF THE DOCTORAL DIPLOMA
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

2020

Marios D. Fragkoulis
Department of Management Science and Technology
Athens University of Economics and Business

Department of Management Science and Technology
Athens University of Economics and Business
Email: mfg@aueb.gr

Supervised by Professor **Diomidis Spinellis**

To my parents

Contents

List of Figures

List of Tables

Acknowledgements

Throughout my PhD, I have been blessed to be surrounded by wonderful, kind, and, most of all helpful people that contributed significantly in my progress.

First and foremost, I can't thank Diomidis Spinellis enough. He put his faith on me to embark on the PhD and invested great amounts of time and energy in my supervision. He has been there for me, in the good times and the difficult times. A true mentor always willing to devote his precious time and share his talent and profound experience. His support has been invaluable in many aspects, research, coding, funding, teaching. He is a rolemodel, as a scientist, as a hacker, as a person. If we could just have a few copies of him, I think Greece would be in better shape. So if by any chance a prospective PhD student is reading this, here is my advice to you: if you are hard-working, eager to learn, and like challenges, I strongly encourage you to go for it.

Panos Louridas was unfortunate enough to teach me the first clues about academic writing. But this encounter was very fortunate for me. I frequently turn to him for advice because he owns a knowledge base right there in his head. His breadth and depth of knowledge and creative thinking are astounding. Although very busy, Panos has always been available to help me improve my work with his meticulus reviews. I admire that he is so well grounded and active both on industry and research. We are very lucky to have him around; he is a precious asset to the lab.

If we came to observe the characters of the SENSE crew (besides my crappy self) I think we would observe some trademark characteristics: hard-working, caring, and fine quality of persons. I bet Tushar couldn't find a single smell among them :-) . The SENSE group of PhD students, which consists of Maria Kechagia, Tushar Sharma, Stefanos Georgiou, and Antonis Gkortzis, has a new tradition: watching StarWars movies with junk food!

Maria is one of the purest persons I have ever met. Armed with kindness and a smile, even when things don't go as planned, she brings grace to our group. She is friendly and easy to talk to and also great to collaborate and exchange thoughts on any subject. Maria is also a fantastic hostess; we all remember the fantastic whole-day trip to Salamina for swimming, lunch, and coffee.

I remember when I went to the airport to pick up Tushar. A thoughtful, gentle, and, genuine guy that came to Greece from India and managed to adapt and pick up speed in no time. Tushar has helped me become more succinct in my research. I have enjoyed our paper collaborations very much, not to mention the amazing Indian food and the good company. He is very communicative and a great story teller.

Stefanos, the benign gym guy, brought energy in our lab and a steady lunch schedule :-) . The conversations and the happy short breaks for a joke he provides make my day. That said, what I like most about Stefanos is his frequent proposals about doing staff together as a group. Even though I haven't followed up as much as I wanted, his enthusiastic suggestions keep us close.

Antonis owes us a boat ride! And I'm not afraid to ask for it (please be careful :-) . Besides that SENSE has its own skipper now, Antonis has a good word for everybody and the intention to go the extra mile to help. Not to mention that he always finds an interesting underground story to share about his days in Thessaloniki or Groningen.

And there is also the old school SENSE: Dimitris Mitropoulos, Vassilis Karakoidas, Giorgos Gousios,

Stefanos Androutselis, and Kostas Stroggylos.

Dimitris helped me improve my perspective on things through our conversations. Although a person with a skill set in diverse aspects he remains humble and that's a good example to follow. He is much fun to hang out with and a good company to discuss serious matters. I have enjoyed our bike rides back in the days.

Vassilis has his own fun way of expression. I enjoy how he finds analogies and examples to back up his statements. He is a deep thinker and always has an answer as to why it is so. Vassilis has helped me look for the right questions and have the answers to those questions.

Georgios is the first I met from the SENSE group. I will never forget how patiently he endured my questions when I was beginning my PhD. When I was dubious on a writing approach for my first research papers, he pointed me at the right direction. He has also kindly reviewed my papers many times. Now our paths meet again at the examination of my PhD.

Stefanos is known for his writing prowess and creative logo designs among other things. The few times that we collaborated, I noticed how his work lends to art.

With Kostas we didn't overlap significantly, but I do remember his sharp technical advice and expert biking takes.

Furthermore, I want to express my gratitude to the rest of the Professors of my PhD committee for their help in improving my work: Damianos Chatziantoniou, Angelos Bilas, Yannis Smaragdakis, Nektarios Koziris, and Georgios Gousios. Their participation is an honor for me.

Damianos Chatziantoniou is a cool business-oriented Professor in databases to whom I turn to for the quirks of database research. I am fortunate to have him as advisor to my PhD. He is always keen to answer my questions and explore new ideas.

Angelos Bilas gave me invaluable advice at the dawn of my PhD and helped me improve significantly my first research paper. Through our communication and my one-day visit to his group in Heraklion I learnt to think and write more pragmatically.

Yannis Smaragdakis has been kind to review my papers and answer my questions (even during lunch break at the PL seminar :-). His advice on my demo presentation for a paper saved me from a lot of mistakes.

I didn't have the pleasure to collaborate with Nektarios Koziris so far, but I respect his work and I am grateful to have him in my PhD committee.

I am thankful to Prof. Cristian Cadar who provided me with insightful hints during the poster session of a paper where I was fortunate to meet him. I am also thankful for his quick responses to my questions.

I am grateful to Prof. Adam Vrehopoulos for helping me prepare the user study that I carried out for the validation of my research work.

I would also like to thank two Professors of the DMST department, Christos Tarantilis for his sharp advice regarding the PhD grind and Nancy Pouloudi for her support when I assisted her with one of the undergraduate courses she taught.

I continue with the staff of the DMST department's computer labs: Christos Lazaris, Stavros Grigorakakis, Vaso Tangalaki, and Giorgos Zouganelis (former employee). Because they are based next to our offices, we stalk them with requests from time to time. Christos has helped me with network diagnostics and data rescues numerous times. Stavros is always quick to help me overcome a system's malfunction and provide expert advice. Vaso takes care to provide us with access to various systems of the department. Giorgos has provided technical support in the past.

Switching to a more personal tone, Panos, Vasilis, Dimitris, and Aleksandros have been my closest friends. Thank you guys for your huge support in the course of my life and for all the wonderful time we have together.

Christina is my virtual sister since we grew up together. Although we haven't spent loads of time recently, we share a bond, and she is there when I need her.

My brother, Lefteris, is a super cool person whose experience in life can fill numerous pages. He is famous for this sort of life-drawn advice, which he couples with amazing stories.

My cousin and godmather Anna is the sponsor of my humble home where I have spent countless hours of undistracted research work. Thank you Anna! My aunt, Tasoula, and my cousin, Sofia, have also contributed in this context.

My parents, Dimitris and Vaso, are the kind that do everything for their children even at their own expense. They have supported me financially, morally, with prayers, by cooking, and the list goes on. They are the unseen heroes whose only desired reward is seeing their effort bloom. Thus, this whole effort is devoted to them as a minor acknowledgment of their service to me for the past thirty plus years.

Special references come last. So this is for you Maria, my love and my smile. Thank you for your encouragement and support that made the dark days sunnier. Thank you for bringing joy and laughter in my life. Thank you for being by my side.

Summary

The digital data has become a key resource for solving scientific and business problems and achieving competitive advantage. With this purpose the scientific and business communities worldwide are trying to extract knowledge from data available to them. The timely use of data significantly affects scientific progress, quality of life, and economic activity.

In the digital age the efficient processing and effective data analysis are important challenges. The processing of data in main memory can boost processing efficiency especially if it is combined with new software system architectures. At the same time useful and usable tools are required for analysing main memory data to satisfy important use cases not met by database and programming language technologies.

The unified management of the memory hierarchy can improve the processing of data in main memory. In this architecture the communication between the different parts of the memory hierarchy is transparent to the applications and optimization techniques are applied holistically. The data flow in the memory hierarchy so that the ones that will be processed shortly are closest to the CPUs and programming languages treat temporary and permanent data of any type uniformly. As a result, new data analysis systems can be developed that take advantage of faster main memory data structures over disk-based ones for processing the data leaving the memory hierarchy to care for the availability of data.

The absence of suitable analytical tools hinders knowledge extraction in cases of software applications that do not need the support of a database system. Some examples are applications whose data have a complex structure and are often stored in files, eg scientific applications in areas such as biology, and applications that do not maintain permanent data, such as data visualization applications and diagnostic tools. Databases offer widely used and recognized query interfaces, but applications that do not need the services of a database should not resort to this solution only to satisfy the need to analyze their data.

Programming languages on the other hand rarely provide expressive and usable query interfaces. These can be used internally in an application, but usually they do not offer interactive ad-hoc queries at runtime. Therefore the data analysis scenarios they can support are standard and any additions or modifications to the queries entail recompiling and rerunning the application.

In addition to solving problems modeled by software applications, data analysis techniques are useful for solving problems that occur in the applications themselves. This is possible by analyzing the metadata that applications keep in main memory during their operation. This practice can be applied to any kind of system software, such as an operating system.

This thesis studies the methods and technologies for supporting queries on main memory data and how the widespread architecture of software systems currently affects technologies. Based on the findings from the literature we develop a method and a technology to perform interactive queries on data that reside in main memory. Our approach is based on the criteria of usefulness and usability. After an overview of the programming languages that fit the data analysis we choose SQL, the standard data manipulation language for decades.

The method we develop represents programming data structures in relational terms as requires

SQL. Our method replaces the associations between structures with relationships between relational representations. The result is a virtual relational schema of the programming data model, which we call relational representation.

The method's implementation took place on the C and C++ programming languages because of their wide use for the development of systems and applications. An additional reason why C++ was chosen is the availability of a large number of algorithms and data structures that it offers. The implementation includes a domain specific language for describing relational representations, a compiler that generates the source code of the relational interface to the programming data structures given a relational specification, and the implementation of SQLite's virtual table API. SQLite is a relational database system that offers the query engine and the ability to run queries to non-relational data through its virtual table API.

The implementation expands to the development of two diagnostic tools for identifying problems in software systems through queries to main memory metadata related to their state. C as the implementation language of many software systems is ideal for the application of this idea. For this purpose we incorporate our implementation in the Linux kernel. Important implementation aspects that we address is synchronized access to data and the integrity of query results. We also apply our approach to expand the diagnostic capabilities of Valgrind, a system that controls the way that software applications use memory.

The overall evaluation of our approach involves its integration in three C++ software applications, in the Linux kernel, and in Valgrind, where we also perform a user study with students. For the study we combine qualitative analysis through questionnaire and quantitative analysis using code measurements. In the context of the C++ applications the performance measurements between PiCO QL queries and the corresponding queries expressed in C++ show that SQL combined with our relational representation provides greater expressiveness. The same happens when we compare our approach with SQL after importing the data into a MySQL relational database system. The efficiency of our approach is worse than C++ and better than MySQL. The queries with our approaches need twice as long time to run compared with C++ regardless of the problem's size. The SQL queries in MySQL require double, triple, or more time to execute compared to our approach.

In the context of the Linux kernel where our relational interface functions as a diagnostic tool we find real problems by executing queries against the kernel's data structures. Access to files without the required privileges, unauthorized execution of processes, the identification of binaries that are used in loading processes but are not used by any, and the direct execution of system calls by processors belonging to a virtual machine are the security problems we identify. In addition we show queries that combine metrics from different subsystems, such as pages in memory, disk files, processor activity, and network data transfers, which can help identify performance problems. The measurement of query processing time and the added overhead to the system encourage the use of our tool.

The diagnostic tool we developed for Valgrind detects problems, additional to those found by Valgrind, through the use of questions in the collected metadata of the application being tested. The `bzip2` tool for instance wastes nine hundred KB where all the memory cells are consecutive in a single pool. This size is equivalent to twelve percent of the total memory that the application needs to operate. Through queries on the dynamic function call graph formed during an application's execution we find a code path that is performance critical. It is located in the `glibc` library and is widely used by the `sort` and `uniq` Unix tools. This optimization was implemented by glibc's development team and was included in the next version without our contribution.

Finally, in the user study the one group expresses analysis tasks with SQL queries and the other with Python code. The results show that the time required for the expression of an analysis job is smaller when SQL is used. On the contrary no statistically significant differences are observed between the two approaches in terms of usefulness, efficiency, and expressiveness, although our approach has a higher rating. For the dimension of usability the evaluations demonstrated no clear winner, but

both approaches achieved very good evaluation. The evaluation of the SQL group code's performance shows that the SQL group had more correct replies achieved with less time of programming. We consider this metric indicative of our approach's usefulness vis a vis Python, which is also widely used for data analysis. We also consider the time required for the expression of an analysis task as a usability factor.

The challenges to the processing of data continue to emerge at an unabated pace. In this environment software applications require solutions for the analysis of user data, but also to solve problems relating to their operation. The processing of data in main memory can bring important benefits in combination with other innovations. In this direction new architectures that benefit the efficient processing of data can play an important role. We hope that this thesis will aid the efficient processing and effective data analysis expected by users.

Chapter 1

Introduction

1.1 Context

Today's systems employ multiple levels of control for various data management aspects, such as querying, naming, access control, storage reclamation, and organisation. An indicative example concerns the programming language and the file system, both of which manage storage and perform buffering. This separation of concerns in the conventional architecture accounts for double system workload, independent optimisation of memory levels, and increased system software development costs. The current trend of establishing standard interfaces for clean communication between modules, isolated in their own corners with local authority, is shortsighted.

Data management has become a crucial aspect of system operation, yet we continue to walk along lines drawn decades ago. We trust programming languages to deliver data resident in main memory, utilise explicit operating system calls to access files on disk, and rely on translation code to access database data. We manage data differently with respect to their longevity, we underutilize our memory hierarchy as well as impose increased software application development costs.

An alternative approach embraces a data model where transient and persistent data share a uniform representation while data of any type may persist for as long as it is required and data persistence is independent of the storage medium. This approach provides direct access to data without the intermediation of expensive operating system calls. Data move automatically across the memory hierarchy smartly and binaries can execute for both transient and persistent data. The hierarchy of memory is abstracted into a *single-level store* [KELS62] where no functionality repeats itself. This design allows each system function to be implemented and applied at a single level, such as query processing and garbage collection, which concern the virtual memory system as a unity. This approach follows the principles of orthogonal persistence [ABC+83], which entails a unified data model across the hierarchy of memory (no translation layer), for the whole data lifecycle (no special manipulation of persistent data), for all data types (no ineligible data types), for all storage media. This thesis does not present an orthogonally persistent system nor is orthogonal persistence a crucial aspect of our work. However, we draw part of our motivation for carrying out this work from the concept of orthogonal persistence. Therefore, we consider useful to examine it in the related work chapter.

Query systems can benefit from advancements in main memory processing in that they can operate with main memory data structures, which are faster than disk-based ones and provide more opportunities for optimisation [Spi10b]. However, the support for querying main memory data provided by current programming languages and database systems leaves something to be desired. This thesis examines the characteristics of main memory query systems designed and implemented for the conventional architecture that is adopted by most systems nowadays and studies the architectural properties of orthogonal persistence. It presents a main memory query system that is useful in the context of the conventional architecture and could prove to be also useful in an orthogonally

persistent system.

1.2 Problem statement

The problem this dissertation attempts to solve is how to query main memory data in a useful and usable manner. The answer to this question can provide the specifications of the query service for an architecture that adheres to the principles of orthogonal persistence. Such an architecture is described in the previous section. Besides its potential of use, a query service for main memory data is important today to data analysts for obtaining user defined views from an application's data and to system administrators for diagnosing problems with a system's operation using its metadata. Although queries to main memory data are possible, the query service comes with deficiencies.

Queries to application data can be provided by database management systems (DBMS). DBMSs complement applications with an API and a standard query language to manage data [Ree00, San98]. They offer an important set of properties, namely atomicity, consistency, isolation, and durability (known as ACID) and expressive, powerful views on database data. Outsourcing an application's data management to a DBMS is the typical design option, but one size does not fit all [SC05].

Many applications do not require a fully-fledged DBMS. Such applications either do all their processing completely online, or store their data using bespoke file formats. This can be a sound design choice, for program data may be stored in data structures provided by the programming language; in fact, most modern languages offer a collection of such data structures accompanied by algorithm implementations to traverse and manipulate them. Examples include the C++ STL library and the *Java.util* containers. While containers are optimized for storing objects and running algorithms, they do not offer an easy means to perform ad-hoc queries on program objects stored in them in the powerful way databases do. This leads to situations where lengthy custom code has to be written for querying container objects or, worse, where the programmer has to introduce unnecessary dependencies to a DBMS just for improving in-memory data querying capabilities of a program.

Systems, such as operating systems and application systems, require a wide variety of diagnostics checks to maintain integrity, ensure security, and optimize performance. The available tools of the trade are typically event-based [CSL04, PCE+05]. This kind of instrumentation is useful for resolving many issues, but there are situations where the state of the system provides an advantageous view to the problem at hand that event instrumentation cannot tell. A trivial example regards querying a system's open files. Instrumenting system calls to `open()` for a specific time window in order to observe open files will probably not output the complete list of open files in the system, only those that were opened during the instrumentation.

1.3 Proposed solution and contributions

Our proposed solution is a method and an implementation for representing an arbitrary imperative programming data model as a queryable relational one. The Pico COllections Query Library (PiCO QL) uses a domain specific language to define a relational representation of application data structures, a parser to analyse the definitions, and an SQL interface implementation.

The implementation of PiCO QL imposes a minimal overhead to the program and the programmer, as most of the tedious effort of wrapping a data structure in an SQL view is done through metaprogramming, while a domain specific language (DSL), which follows the style of the SQL data definition statements, allows the specification of data structures to be queried. PiCO QL has been developed for C/C++ and can provide a live relational view of arbitrary main-memory data structures.

Our thesis provides the following contributions.

- A method for mapping an imperative programming data model into a relational one to provide querying of in-memory object graphs using SQL and an SQL relational query processor
- A demonstration of the method's validity through the implementation and evaluation of PiCO QL, a library that supports interactive SQL queries against data structures of C/C++ applications
- A diagnostic tool to extract relational views of the Linux kernel's state at runtime; each view is a custom image of the kernel's state defined in the form of an SQL query.
- A diagnostic tool that extends the Memcheck, Cachegrind, and Callgrind utilities of the Valgrind instrumentation framework for analysing an application's memory profile through SQL queries
- A user study that compares the PiCO QL interactive SQL interface to a Python scripting interface

1.4 Research methodology

The research method followed throughout this thesis is shown in the UML diagram of Figure 1.1.

The first step in the method regards choosing the area of interest for this thesis by means of exploring ideas and studying the literature in a number of areas of systems' research. The area of uniform data management emerged from this process. Then focused study of this area revealed opportunities for research, which in turn gave birth to the problem statement of this thesis.

With the problem at hand, the method for the relational representation was conceived followed by the design of PiCO QL. Then an examination of the various available implementation technologies for the implementation of PiCO QL pointed to C and C++. The reasons for this decision are grounded on the wide use of C in system settings and the popularity of C++ for developing systems and sophisticated applications. In addition, C++ offers an impresssive array of libraries with data structure and algorithm implementations.

PiCO QL's evaluation followed its implementation. The evaluation began with three C++ applications. Next PiCO QL was fit into the Linux kernel as a loadable module and was evaluated as a kernel diagnostic tool. Finally, experience with users was evaluated compared to Python scripting on the basis of analysing memory profiles produced by the Valgrind instrumentation framework.

The evaluation of PiCO QL triggered circles of redesign and implementation until fitting the criteria of *usefulness* and *usability*.

1.5 Thesis outline

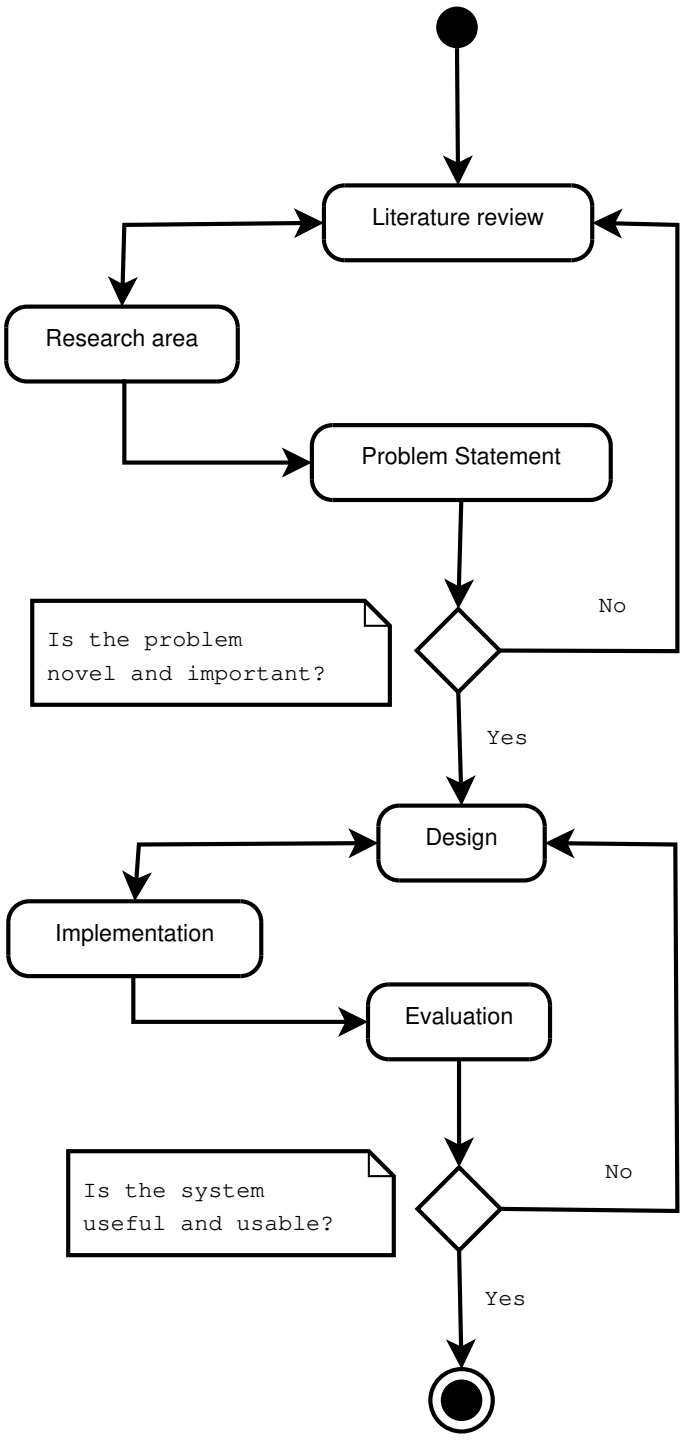
This dissertation consists of six chapters. We outline the remaining five below.

In *Chapter 2* we present the related work on uniform data management, programming language support for queries on the language's data structures, and diagnostics tools for operating and application systems.

In *Chapter 3* we present our method for the relational representation of arbitrary data structures in relational form and the design of the PiCO QL SQL query library.

In *Chapter 4* we detail the implementation of PiCO QL, which provides SQL queries on data structures of the C and C++ programming languages.

Figure 1.1: The research methodology followed



In *Chapter 5* we describe the evaluation of PiCO QL on the Linux kernel, three C++ applications, and the Valgrind instrumentation system. For the Linux kernel we implemented PiCO QL as a loadable kernel module in order to query kernel data structures, such as the list of active processes in the system. The evaluation includes a user experiment where students use the PiCO QL interface and a Python scripting interface to analyse memory profiles produced by Valgrind tools.

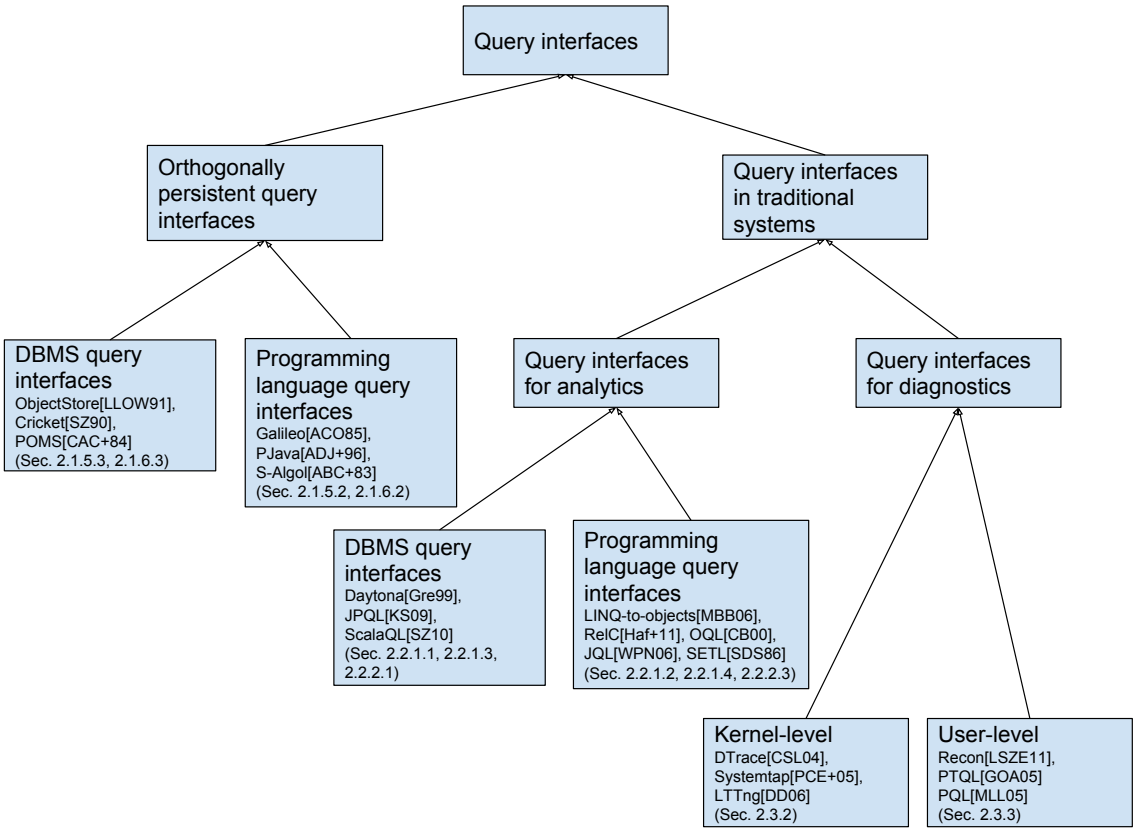
Finally, in *Chapter 6* we present the conclusions of this thesis and discuss possible avenues for future work.

Chapter 2

Related Work

Our work on query interfaces for program data covers the areas of orthogonal persistence (Section 2.1), programming language support for queries on data structures (Section 2.2), and diagnostic tools (Section 2.3). Figure 2.1 presents a map of the state of the art related to our thesis and lists significant pieces of work in each one.

Figure 2.1: Map of research work related to this thesis



2.1 Orthogonal persistence

Orthogonal persistence entails a unified data model across the hierarchy of memory (no translation layer), for the whole data lifecycle (no special manipulation of persistent data), for all data types (no ineligible data types), for all storage media. Systems that adhere to these principles are called orthogonally persistent. We investigate orthogonal persistence within three contexts, the operating system, programming systems, and the storage system. For brevity we refer to these systems as *persistent operating systems, persistent programming systems*, and *persistent stores* throughout this section. Respectively, the term persistent system stands for an orthogonally persistent one.

Our study on persistent systems concerns those that provide *transactions* and/or *distributed processing*. This decision is based on two reasons. First, the wide demand for transactions and distributed processing from applications makes these two aspects important to study. Second, providing transactions and distributed processing in the paradigm of orthogonal persistence offers exciting challenges. We compare and contrast persistent systems on four dimensions namely *persistence implementation, data definition and manipulation language, stability and resilience, and protection*.

In the sections that follow we provide background information on orthogonal persistence (section 2.1.1), describe in detail the scope of our study (section 2.1.2), present existing surveys on the subject (section 2.1.3), and list the dimensions that we use to examine persistent systems (section 2.1.4). Furthermore, in sections 2.1.5 and 2.1.6 we describe the existing work on orthogonal persistence. Finally, in section 2.1.7 we present performance optimisations for persistent systems.

2.1.1 Background

Orthogonal persistence is no new concept. The decades of 80s and 90s, for the most part, saw the rise of a new research paradigm that promoted uniform manipulation of data, that is, systems where the use of the data is orthogonal to their persistence [AM95]. Orthogonal persistence is described in terms of three principles [ABC+83]:

- **persistence independence**

  dictates that data manipulation is uniform for both transient and persistent data. Behind the scenes data move automatically across the hierarchy of storage devices in a uniform format. Consequently, applications are not concerned with programming data transfers in and out of main memory and translating between different data representations.

- **data type orthogonality**

  regards a complete data model that is independent of the data's persistence. Applications need not distinguish between data types that are eligible or ineligible to persist.

- **persistence identification**

  denotes how the identification of persistent data is performed. This has to be transparent to the programmer and independent of the type system.

The first aspect related to orthogonal persistence is the single level store devised in 1962 where data transfers across the hierarchy of memory levels are undertaken by the system in a manner that is transparent to the programmer. Two decades later, PS-Algol is born, a persistent programming language, and the principles of orthogonal persistence are stated. A few years later automatic garbage collection is linked with orthogonal persistence where orphan data, either transient or persistent, are garbage collected by the system. At the same time a crash recovery mechanism for a persistent memory system is presented.

The first trails of orthogonal persistence can be traced in the distant past. In 1962, Kilburn et al [KELS62] devised the *single-level storage* to describe the practice of treating memory as a uniform resource. They describe an automatic system that appears to employ a single level of storage as opposed to its actual two-level central store hierarchy. Information flows automatically between the two levels without explicit I/O.

The central store is a combination of core store and drum store. When an address is not found in core store, the drum store, and tape store are searched in turn to retrieve the requested memory block. Hence, the drum store and tape store communicate with the machine via the main core store. Although the authors defend single level storage in terms of the central store, the approach seems to operate with success in successive levels such as the tape store. Emphasis is paid to the automated data transfer between the levels of the memory hierarchy.

Twenty years later, Atkinson et al [ABC+83] extend the S-Algol language with features to support persistence and introduce the principles of orthogonal persistence. First, data objects can persist the execution of a program irrespective of their type (data type orthogonality). Second, the same coding constructs can be applied to objects either transient or persistent (persistence independence) and, third, the identification of persistence is independent of the selection of data objects at the language level. These principles constitute a unified programming model.

The result is PS-Algol, an orthogonally persistent language, where the determination of persistence is left to the system and is based on object reachability. PS-Algol is designed to affect minimally the underlying language. As such, PS-Algol contributes standard functions implemented by means of functional extensions to S-Algol. The outcome of the endeavour is a language with very low adoption cost from the S-Algol community that addresses persistence using already established methods of data preservation, that is by referencing data with their name.

Type checking in PS-Algol is carried out by the system and is based on the type information that each structure holds. Following programming experience with the language, the authors argue that the abstraction of persistence contributes a significant reduction in development time, source code size, and software maintenance due to the embracement of a unified programming model.

In 1986, Thatte [Tha86] describes a persistent memory system with a crash recovery mechanism based on a uniform memory abstraction implemented on a large, virtual address space. The system provides uniform treatment of data irrespective of their longevity, where persistence is implemented in terms of reachability from a persistent root. Objects not reachable from the transient root or the persistent root are garbage collected. As such, object persistence is orthogonal to object types and storage media. Recovery from system crashes is achieved with a checkpointing scheme, which saves system state, and a rollback scheme, which restores the state of the last checkpoint following a crash. To support resilience, an undo and a redo log are utilised to achieve finer data restoration after the last checkpoint.

2.1.2 Scope

Our study targets orthogonally persistent systems. The notion creates a set of requirements in the context of an operating system on the one hand and another set in the context of programming and data management systems.

- Along the lines of [ABC+83] *programming systems* and *data management systems* are considered orthogonally persistent if:

  - **they employ a uniform programming/storage model for data irrespective of their longevity**
    In our data-centric era, data may need to persist the end of the application program or multiple versions of the application program or even the support system that accommodates data. It is irrational to impose special manipulation for persistent data. In persistent

systems data access should be independent of the storage medium, i.e. there should not exist a storage medium that requires special manipulation to access. In practice, access to data in cache, RAM, hard disk etc. should be manipulated alike.

In addition, the loss of uniformity creates a mismatch between the programming model and storage model. One instance of this situation is the notorious object-relational impedance mismatch where specific interfaces are used to access relational databases from programming languages and translate result sets into objects. Another case of mismatch concerns the transformation of file contents into main memory data structures. The solution, a translation layer, amounts to 30% of the application code written [ABC+83].

– **data share the same rights to persistence irrespective of their type**

This principle states that persistent systems should not have data types ineligible for persistence or data types that require special treatment to persist.

- According to Vaughan *et al.* [VD92] **persistent operating systems** entail the following aspects.

  – persistent object management along with the relationships between them
  – transparent and reliable management of the hierarchy of memory
  – persistent processes
  – protection of objects through access control

  The conventional operating system architecture has separate functions for file management and memory management, which reflect the management of persistent and transient data respectively. Persistent operating systems unify the two. In the conventional architecture, the two modules behave autonomously and implement one or more data management tasks within their boundaries. Policies optimised to fit a certain, sub-systemic goal is the general practice. As a result, a task may happen multiple times in the course of data movement within the hierarchy of memory thereby increasing system workload. Storage management is an indicative example performed by both the programming language and the file system.

  Persistent systems implement the above aspects at a single point and apply policies system-wide. Consequently, they reduce system workload, optimise performance systemically, have less complex modules, and lower software production costs.

2.1.3 Surveys on orthogonal persistence

Four surveys [DRH+92, DH00, AM95, DKM10] have documented and summarised problems, progress, and achievements in the area of *orthogonal persistence*. The first two focus at the support that persistent systems require from the *operating system*. Specifically, the first notes the reasons why traditional operating systems can not support orthogonal persistence, and lays out the fundamental dimensions for *persistent operating systems*. The second studies the appropriateness of both traditional operating systems and persistent operating systems to support persistent systems, but proposes a new class of operating systems in view of the deficiencies of both the existing classes. The third survey targets the *application level* and, specifically, the support required from programming systems and data management systems for developing persistent application systems. The last survey examines the support for orthogonal persistence provided by modern programming language systems.

In their article [DRH+92], the authors research the requirements that persistent operating systems raise and reason why operating systems to date are inconvenient vehicles for the cause. They ground their examination on four dimensions namely *addressing, stability and resilience, process*

management and *security*, explain the persistence implications in each of these areas, and sketch the structure of a persistent operating system.

Regarding *addressing*, the authors argue that software address translation and memory mapped files can help implement a uniform addressing scheme as required by orthogonal persistence. However, both have shortcomings. The former is simple but slow, while the latter does not support resilience.

Resilience and stability of a persistent store can be achieved primarily by means of shadow paging. Experiments with two stabilised versions of the address mapping tables certify that atomic updates can happen in a secure manner. Moreover, checkpoint operations that move the persistent store between stable states are adequate but impose a heavy load on performance. Implementations are described that reduce this effect mainly through identifying and updating only modified pages.

Process management in a system that employs orthogonal persistence poses significant challenges. A single logical shared address space, which directly utilises the address translation hardware, is the desired solution according to the authors. The lightweight thread model employed by modern operating systems can be used, but it does not offer *protection* at the process level. Possible solutions for this concern are type secure programming languages, protection at the storage level, multiple persistent address spaces, and page-level protection to isolate access by threads. The shortcomings carried by each of these solutions are decsribed.

The survey of [DH00] examines the inabilities of traditional operating systems to become vehicles for persistent systems. Specifically, operating system support:

- is insufficient to manage persistent objects and their interrelationships
- provides limited abstractions for resilience since the control over the swapping location is limited
- obstructs the construction of persistent processes through their inappropriate protection and naming and inaccessible process metadata

However, although persistent operating systems to date are explicitly designed and constructed to provide the necessary abstractions for orthogonal persistence, these abstractions mismatch with the abstractions required by persistent application systems.

The authors argue that a new class of operating systems is required, which moves much of the responsibility of the operating system to the application system based on the following principles.

- Flexibility in construction of application systems
- Low kernel-user level coupling
- Ease of writing synchronisation mechanisms
- Independence of operating system authority

The *Charm* [HD98] operating system follows the above principles. It provides no thread or process model but autonomous addressing environments, the basis for the construction of arbitrary concurrency models. Sharing and protection are controlled by each persistent application system.

The third survey [AM95] presents the motivation for persistent application systems and describes the importance and benefits of integrating the programming language perspective with the database and file system perspective through the persistence abstraction.

The authors explore four architectures regarding the support of persistent object systems:

- combination of existing systems

- extension of existing systems
- explicit integrated system design
- persistent operating systems

The authors review the design principles for such systems and discuss how each architecture can support the principles of orthogonal persistence stated in section 2.1.1. Thorough description of the technology and the achievements towards the adoption of persistent object systems is given.

The last survey [DKM10] provides a review of programming language support for the uniform treatment of objects independently of their longevity and data type. The article emphasises the object-relational impedance mismatch and the benefits of orthogonal persistence (section 2.1.1). The authors refer to approaches, such as Microsoft's LINQ, towards orthogonal persistence and specify their orthogonality by measuring the level of adoption of the principles detailed in section 2.1.1.

2.1.4 Dimensions of persistent systems

We use four dimensions to compare and contrast systems that exhibit characteristics of orthogonal persistence in response to four questions presented below. Since we study orthogonally persistent systems, the first question, which regards the implementation of persistence, emerges naturally. The same holds for the second question that concerns the data management interface supported by a persistent programming system or a persistent store. The last two questions come from [DRH+92]:

1. **How does the system implement persistence?**

   One of the main characteristics of orthogonal persistence is the uniform manipulation of transient and persistent data and the availability of persistence to all data types. How the system implements these requirements is an important aspect. This dimension regards the persistent storage module and its organisation for preserving the data representation.

2. **What language does it use for data definitions and manipulation (DDL, DML)?**

   The DDL and the DML show how data management is provided by the system. The previous dimension regards the implementation of persistence. This dimension is concerned with the manipulation of persistence and other aspects, which happen through the DDL and the DML. These two dimensions should be independent in a persistent system. The aspect of data queries is of particular interest in this dimension.

3. **How does the system support stability and resilience?**

   Resilience is a property of fundamental importance for a persistent system. The persisted objects are not independent as is the case with files in a conventional file system, but relate to other objects. Thus, the loss of one object can have profound impact on the integrity of the overall system.

4. **How does the system ensure data protection?**

   Due to the uniform addressing of transient and persistent objects that orthogonal persistence entails, data protection is a major challenge. Protection mechanisms at the programming level (compilers), process level (address space organisation, page access), and storage level (capabilities) are proposed.

System type	Level	Questions
Persistent operating system	System	1, 3, 4
Persistent programming system	Application	1, 2, 4
Persistent store	Application	1, 2, 3, 4

Table 2.1: Types of persistent systems examined and questions applicable to them

2.1.4.1 Persistence implementation

Persistent systems implement persistence mainly in two ways, using software address translation or memory-mapped files. Software address translation makes it possible to translate a main memory address to a secondary storage address and vice versa with the aid of the virtual memory system. Some systems require that the activation of a persistent structure happens in the same main memory area as the initial one. This is an important restriction but not so much for the current generation of 64-bit systems due to their huge address space. Nevertheless, there are techniques to overcome this constraint, e.g. by making references relative.

Memory mapped files allow a mapping to happen between their on-disc representation and a process's address space. As such, a process gains access to a file on disc without implementing the expensive seek and read system calls. Instead, the paging mechanism retrieves pages not present in the process space on demand. Dirty pages are written back to the file's original location on the disk in a lazy manner, i.e when the page in main memory is replaced. Reading and writing to persistent files happens transparently without user intervention. Memory-mapped files obviate the need to convert between data representations.

2.1.4.2 Data definition and manipulation language

The DDL and DML can be offered by a general-purpose programming language extended to promote persistence. However, the type system should not reflect this extension as this would disagree with the second principle of orthogonal persistence, i.e all data types have a right to persistence. Persistent programming languages designed from scratch for persistent systems also exist.

2.1.4.3 Stability and resilience

Two ways to achieve stability and resilience are the implementation of transactions borrowed from the database community and shadow paging. The first way is appropriate for persistent systems built on top of conventional architectures that use files as their unit of abstraction. A transaction log is used to record transactions and roll back to a stable state after a failure.

Shadow paging ensures a persistent store's integrity through the utilisation of memory mapped files and virtual memory. Its core concept is the mapping table, which maps the virtual addresses of the persistent store to physical disk addresses. Shadow paging maintains a consistent state of the data and the mapping table that is updated after each checkpoint operation. The consistent state can be used to recover from a system failure.

2.1.4.4 Protection

Protection in a system regards the integrity of data, their durability, and access control. Durability and integrity can be achieved by introducing persistent stores and stability mechanisms such as a log.

Access control can be provided through compilers, multiple address spaces, page access control, and capabilities [DVH66]. Compilers in type secure programming languages can enforce protection through the language's type system if the system is to support a single programming language. At the

process level, multiple persistent address spaces can safely separate a process's working area from another's. This approach requires communication mechanisms to provide processes with a sharing channel. Alternatively, restricting access on a per page basis can allow processes to coexist without trust issues in a single address space. At the storage level, capabilities can be used, which encapsulate the rights for accessing a persistent object. They are provided to eligible users upon the creation of an object and secure the users' permission to access it.

Table 2.2: Systems that adhere to orthogonal persistence

System	Type (OS/-PL/PS)	Transactions/ Distributed	Persistence implementation	DDL/DML	Stability & resilience	Protection
Grasshopper [DdBF+94]	OS	T	single-level store, commit pages to stable store	X	consistent data copies in subsets, persistent kernel	capabilities
Keykos [BFH+92]	OS	T	single-level store, checkpoint modified pages coarsely, commit pages synchronously	X	system wide checkpoints, page-level journaling, persistent kernel	capabilities
Galileo [ACO85]	PL	T	assumes a persistent environment is present	Galileo's DDL, for comprehensions on class extents	atomic transactions, undo when failure strikes	static type checking, abstract types match types with operations on values
PJava [ADJ+96]	PL	T	transitive persistence: objects reachable from a persistent root	persistent store class, Java's DDL and DML for objects	checkpoint mechanism	Java type system, class information in stable store
Java cards [ST06]	PS	T	persistent memory (flash) for objects, atomic and serialised data access	CreateNewObject, GetField, PutField	transactions, undo/redo log	-
Cricket [SZ90]	PS	TD	single-level store through memory mapping	-	transactions without recovery	capabilities
POMS [CAC+84]	PS	T	software translation via pointer swizzling, persistent objects of any type reachable from a persistent root	PS-Algol	shadow paging	passwords
Clouds [DLAR91]	OS	DT	single-level store using mapped files	X	transactions, replication across nodes	data encapsulation within an address space
Napier88 [Mun93]	PL	DT	software address translation between local and stable heap	create, update, and delete objects	transactions, shadow paging	none, read, write, or execute access control to pages
SHORE [CDF+94]	PL	DT	software address translation: pointer swizzling	language generic: SHORE definition language similar to ODL, C++ binding	transactions, redo a client's work at server	object access control component
Objectstore [LLOW91]	PS	DT	memory-mapped architecture, single-level store using the virtual memory system, persistence as a storage class, object storage	C++ statements, object relationships, queries on collections, associative queries, indexes, optimiser	two-phase commit transactions, object versioning, write-ahead log	set no/read/write access for individual pages through the OS's virtual memory system, static type checking applied to persistent objects too
SPOMS [MDRK93]	PS	D	memory mapped store, compiled-class: language agnostic storage unit	-	-	capabilities distributed when an object is created and used to control access to objects

2.1.5 Persistent systems performing transactions

From resilient operating systems with lower management costs to the simplicity of programming systems that secure the data's integrity automatically, persistent systems that perform transactions lower software production and system management costs and provide an advanced quality of service. For each system we present in Table 2.2 the research undertaken in terms of persistence implementation, data management, stability and resilience, and protection.

2.1.5.1 Persistent operating systems

Persistent operating systems present an appropriate architecture and the necessary primitives for the support of persistent systems. In addition, some persistent operating systems offer resilience to failures by recording their state in fixed intervals and recovering it after a failure. Representative examples are *Grasshopper* [DdBF+94] and the *Keykos* nanokernel [BFH+92]. They both rely on capabilities for enforcing access control and employ multiple address spaces. In terms of persistence and resilience, Grasshopper uses a two-phase commit protocol to persist first the user data and then the kernel itself. Keykos, on the other hand, combines a coarse system-wide checkpoint mechanism and a journaling mechanism for stabilising the system's state in finer intervals.

Grasshopper is an operating system explicitly designed to support orthogonal persistence. Grasshopper employs a fully partitioned address space model where processes execute in a partition and have access only to the data available in that partition. Partitions are called containers. *Containers* are the only storage abstraction, replacing the notions of files and address spaces. *Loci*, on the other hand, replace processes. At any given time, a locus can only access data inside the container it is executing. A container with program code, mutable data, and a locus composes a basic running program. Containers and loci are orthogonal. The interaction between containers and loci in Grasshopper follows the procedure oriented model where a locus invokes a container.

Managers, which are similar to external page handlers of the Mach operating system, provide access to the kernel pages of data stored in the container, handle page-faults, receive data extorted from main memory, and maintain non-RAM resident data. Managers decide the format to store data and abstract over storage media. They only can understand the difference between temporary and persistent data. Managers are ordinary programs and reside in their own containers.

Sharing data is possible through mappings between the containers where a container's region is mapped to another container. Container mappings are composable. The resulting hierarchy can form a directed acyclic graph that is managed by the kernel.

Access control over containers and loci is provided by capabilities. A capability binds an entity with a name, holds the access rights for that entity, and defines whether the capability itself can be copied. The advantages of using capabilities are that they define unique names for entities and that they cannot be forged. Capabilities in Grasshopper are stored in a protected memory region.

Finally, the store's stability is maintained in consistent subsets with no dependencies between them. Managers stabilise the data under their responsibility in turn. The consistent data copy created can be used for recovering the data in case of a system failure. The kernel is itself persistent. It stabilises its state after all managers complete the stabilise operation and commits the new system state.

Keykos is an object-oriented nanokernel implemented in C. It is resilient to failures, of advanced security, and of high availability. Implementations of Keykos have been developed for a number of operating systems, such as System/370 and Unix. In Keykos each process runs in each own segment, the equivalent of an address space. A segment consists of pages or other segments. Segments also replace the function of files in traditional operating systems. In Keykos interprocess communication relies on message passing and access control to objects is based on capabilities.

Persistence in Keykos is the rule rather than the exception. Only the nanokernel distinguishes

between main memory and disk. Registers, virtual memory, and applications persist and are resilient to failures. From an application viewpoint, objects reside in a persistent virtual memory and only the nanokernel knows the distinction between main memory and secondary storage. The single-level store model of Keykos reduces complexity.

Stability and resilience are achieved through a checkpoint scheme and a journaling mechanism. The checkpoint scheme records system-wide snapshots every few minutes. It cooperates closely with the paging system and ensures that updated files and processes in the system are written to the checkpoint area on disk. For finer stability and resilience, Keykos also includes a journal mechanism that provides advanced transaction processing. The mechanism commits pages synchronously and links them with the last checkpoint. This way, if failure occurs, the pages will be consistent after the system restarts.

2.1.5.2 Persistent programming systems

Persistent programming systems take care of the data's persistence automatically. This style of programming where a programmer does not deal with the persistence of the data is termed persistent programming. Some languages, such as Galileo [ACO85] and PJava [ADJ+96], offer an environment for persistent programming with transactional support. The latter is implemented on top of Java. The two approaches implement persistence differently. Galileo assumes the presence of an environment that manages the data's persistence transparently, while PJava uses transitive persistence where all persistent objects can be reached from a persistent root. Galileo defines a conceptual schema for persistent programming with rich support of object-oriented features. Data queries take the form of for comprehensions on the collections of objects of a specific type. In addition to the common object interface of Java, the data definition language of PJava includes creating a persistent store and getting a persistent root. The data manipulation commands are pure Java. For type protection, both approaches rely on the base language's type system; Galileo has a strong type system, which borrows features from ML. PJava stores an object's class information with it and checks that it is cast to an appropriate type when it is retrieved from the persistent store. For data protection against failures, Galileo and PJava use transactions. When failure strikes, Galileo rolls back the effects of incomplete transactions, whereas PJava uses its checkpoint mechanism to recover a consistent state.

Galileo is a strongly-typed, interactive functional programming language that borrows features from ML. It is an expression language where a construct takes values as input and provides a value. Galileo combines programming language and database features to provide a semantic data model for supporting persistent applications.

Galileo's data definition language supports object-oriented features, that is the classification of entities with common characteristics in a class, the generalisation of classes that share a parent-child relationship, the modularisation of data and operations, class extents, associations by means of aggregation, and abstract types.

The language employs uniform treatment of transient and persistent data, without resorting to special manipulation for persistent data. Persistence in Galileo relies on a global environment, which is managed by the system that supports the language, where all values automatically reside.

For querying data, Galileo relies on class extents, sequences, and for comprehensions. A class extent includes the instances of a class, which can be accessed successively through an iterable data structure, such as a sequence. A for comprehension is a programming language construct that takes an iterable data structure and operations, such as filters or transformations, and returns the collection of elements after performing the operations on them.

Protection in Galileo relies on static type checking and abstract types. Each expression possesses a type that can be decided statically. This design allows identifying type violations. Static type checking facilitates testing and provides execution safety. In addition, abstract types allow programs to be

independent of changes in the data representation and define unique operations for each type, which can not be invoked by instances of other types.

Galileo supports single and compound atomic transactions. Single transactions include a single expression; compound ones include multiple. Each expression at top level is a transaction. The system deals with failures by rolling back the effects of interrupted transactions.

PJava extends Java with database facilities according to the principles of orthogonal persistence:

- all data types can persist (data type orthogonality)
- transient and persistent data are treated alike; there are no special code routines for persistent data (persistence independence), and
- the identification of persistent objects, along with their complete structure, is based on reachability from a persistent root (persistence identification).

The extended language, PJava works with a typical Java compiler that outputs standard class files. Only the JVM has been tapped to provide automatic object fetching, persistence, transactions, and recovery. To achieve prefetching, an object cache has been added.

PJava allows the decoupling of programming from considerations related to persistence. Programs use two API calls for creating a persistent store and getting a reference to its persistent root respectively. But the manipulation of objects involves standard Java code. PJava utilises common Java classes produced by the standard compiler. In an example application used in [ADJ+96] only 18 lines of code reference PJava's persistence API explicitly out of a total of 7690 lines of code.

PJava supports transactions utilising Java's exception mechanism to handle errors. The code that commits a transaction is placed in a `try catch` block. A transaction will abort if an exception is not caught and the system will roll back to the state at the beginning of the main method. Otherwise the transaction will implicitly commit at the end of the method. All structures reachable from a persistent root will be promoted together with any referenced objects that they include. Transactions that do not amend any persistent structures behave as read only. A checkpoint mechanism maintains global recovery points. The system will transparently fetch from disk any objects needed to process a transaction without any modification to user defined classes.

Java's strong type checking is preserved by storing objects' class information in the stable store. When a persistent object is retrieved, a cast is made in order to assign the stored object to a new instance. Then the type information reflected by the cast is checked againt the class information stored with the object.

2.1.5.3 Persistent stores

The motivation behind many persistent stores is to accompany persistent programming systems. Cricket [SZ90], and POMS [CAC+84] are two such instances. A recent trend in persistent stores is to store data persistently by default. Java cards [ST06] belongs in this category. Cricket, POMS, and Java cards are object-based stores. Java cards is implemented for flash memory [BCMV03]. In all three cases objects can have any longevity required by definition and their treatment is uniform irrespective of their type. To achieve orthogonal persistence, Cricket employs memory mapping while POMS employs software address translation. Objects in Java cards are by default persistent because they are stored and updated in persistent memory. For access control, Cricket uses capabilities while POMS uses passwords. Access control in Java cards is not reported. For protection against data type violations, POMS stores type information with the objects.

Java cards is a persistent, transactional, garbage-collected memory management system for implementations based on flash technology. In this environment objects are persistent by default. Memory consists of a small RAM module of 8KB size and a flash module of 1MB. The Flash memory used is of type NOR that is memory mapped.

Objects are created in the persistent memory, so they are persistent by default. The DDL includes the `createNewObject` API function. The data manipulation language (DML) consists of the `GetField` and `PutField` API functions. Java cards supports persistent arrays with similar DDL and DML statements. Access to objects in flash memory can happen either via physical pointers and the memory mapping interface or via references using the API methods. References are unique identifiers of objects generated at creation time. A search tree maps references to flash addresses. Unreachable objects are garbage-collected and their space is reclaimed.

Transactions are supported by maintaining a commit buffer that acts as an undo/redo log for rolling back transactions. Java cards supports three atomicity models. The first model offers atomic access to objects, which should be serialised. The second provides explicitly stated transactions that can wrap a group of statements. Finally, copying a region of an array can be performed atomically. Java cards stores modified values in new locations, hence both old and new values coexist in flash memory.

The system offers block level granularity of operations in comparison to commercial EEPROM implementations, which read, write, and erase at byte level. As a result, flash outperforms the EEPROM alternative. However, flash performance degrades as memory is filled up with data because space reclamation only obtains part of the deallocated space.

The Cricket persistent store is built on top of the Mach memory management unit and utilises memory mapping, external pagers specifically, to provide a shared, transactional, directly accessible single-level store. To support transactions, Cricket utilises the transaction mechanism of the EXODUS storage manager [CDRS86].

Cricket provides uniform representation and treatment for persistent and transient data of any type. This approach results in application code simplicity and development time reduction, which is the motivation behind Cricket. Performance tests verify the benefits incurred for specific application systems, such as design environments and multi-media office systems, where response time is the challenge rather than system throughput.

Cricket embraces a client-server architecture, supports distributed applications, and provides multithreading for true parallelism with multiprocessors. Applications in Cricket execute in separate protection domains and communicate via an RPC interface. Access control is implemented by means of capabilities.

The Persistent Object Management System (POMS) supports the PS-Algol persistent programming language. In POMS object persistence is orthogonal to the data's longevity and the type of store, such as RAM and disk.

POMS supports incremental loading and storing of objects. It utilises two separate heaps, one for main memory and one for disk, and two types of addresses. Concerning data transfers, objects reachable from a persistent root are loaded into main memory on demand and their persistent address is swizzled to the local address where they reside.

POMS supports type checking and access control. Type checking in POMS leverages class definitions that are stored within object code and databases. Objects on the heap contain adequate information for identifying their type description. For access control to data, programs are obliged to provide the correct password. Passwords in POMS can contain up to 16 characters.

POMS uses shadow paging to support transactions and recovery. Two versions of directories of databases are maintained in contiguous disk blocks. In this way the data resident on disk before a transaction will not be overwritten by the updated data committed within the transaction. In case of a fault the consistent version of data prior to the transaction is used to restore state.

2.1.6 Persistent systems performing distributed processing

Distributed persistent systems combine the unified management of the interconnected computing resources with lower programming effort for data management to boost user collaboration and per-

formance. Table 2.2 presents how each system implements persistence, supports data management, provides stability and resilience, and offers protection.

2.1.6.1 Persistent operating systems

A distributed operating system offers integrated management of the connected computers as a unity. A persistent one, such as Clouds [DLAR91], offers also transparent management of the hierarchy of memory.

The Clouds distributed, general purpose operating system consists of three levels, the minimal kernel, system objects, and higher level objects. The Clouds architecture consists of compute servers and data servers. The former carry out processing whereas the latter store objects and supply them to the compute servers. Clouds supports the programming languages DC++ and DEiffel, which are distributed versions of C++ and Eiffel respectively.

Clouds is composed of persistent virtual address spaces, called objects, at the operating system level. Objects enclose code and data. They are shared and available to be invoked by clients. Memory in Clouds is shared and distributed and is represented by a two dimensional address space. A variety of memory constructs offer powerful assistance in programming. Per-object memory, per-invocation memory, and per-thread memory provide memory with different scope and sharing attributes. The first is globally sharable, the second is visible within an object, and the third is visible within an object for a particular thread.

Each object contains user code, permanent data, a volatile heap for transient data, and a persistent heap where permanent data reside. As encapsulation dictates, data can only be accessed inside an object. Objects survive system crashes since objects are by default persistent until explicitly deleted. Objects abstract storage and threads and are used to implement computations. Clouds employs a single-level store using mapped files.

Processing in Clouds is driven by threads. Threads traverse objects and execute the code within them. Thus, threads enter different address spaces. When more than one thread execute code in an object address space, semaphores or locks guarantee concurrency control. Distributed programming is efficient in Clouds, since the computations of centralised algorithms can run in a distributed fashion. Threads execute concurrently on multiple compute servers where they access data in parallel.

Clouds supports transactions and guarantees atomicity by use of locking and recovery. Resilience is achieved by replication of objects and threads in different nodes. If the initial thread aborts, one of the replicated ones that managed to commit is selected.

2.1.6.2 Persistent programming systems

Many persistent programming systems for distributed environments facilitate sharing and collaboration by offering a general language or multi-lingual support for managing the stored data across programming languages. Napier88 [Mun93] and SHORE [CDF+94] have this ability. In terms of persistence implementation, both Napier88 and SHORE use a software address translation technique between a local cache and a persistent store. Napier88 defines a simple API for creating, updating, and deleting objects, while SHORE features a generic language similar to ODL. Napier88 achieves resilience through shadow paging, while SHORE records at server-side the work of a client using its log records. For protection, local servers in SHORE have an access control component that check object namespaces. Napier88 supports protection of a page range with none, read, write, or execute access control privileges.

A number of different models of concurrency and distribution [Mun93] have been incorporated into the *Napier88* persistent transactional programming system, which is backed by a persistent object store. It is strongly-typed featuring a sophisticated type system and supports first-class procedures and environments. Napier88 has a layered architecture with no predefined view of the concepts

of distribution, transactions, and concurrency. In this way it allows cost-effective development of arbitrary models of these concepts.

The object store is hidden from the programming environment. An abstraction layer with a heap-based architecture stands in the middle. It features a local heap for creating new objects and caching persistent ones and a stable heap where presistent objects are stored. The stable heap is a view to the object store. It defines an object format that is agnostic to programming languages. Transitions from the one heap to the other happen via swizzling, a software address translation technique.

The API to the stable heap includes functions for accessing the persistent object store, that is, functions to create and delete objects, to update the store in a new consistent state, and to call the garbage collector.

Napier88 provides stability through transactions, resilience with shadow paging, and protection through access control of page ranges. Four access modes are supported: none, read, write, and execute.

The SHORE persistent object system supports heterogeneous distributed processing. It features the SHORE Definition Language (SDL) to establish a language-generic notation so that applications written in different programming languages can interact and share resources. The SDL is similar to the object definition language (ODL) proposed by the object database management group (ODMG).

SDL communicates with object-oriented languages, such as C++, through bindings. Application class definitions are written in SDL and generate type objects accordingly. Then, language-specific tools generate class declarations and special function definitions from the type objects. The generated code is placed in files read by the application programming language along with the OID of the type object.

SHORE offers a number of services. Locking is used to provide concurrency control, transactions safeguard the system's stability, and recovery is achieved with the use of logs. Queries over objects and object clustering add to this list. Regarding transactions, applications transmit a commit request to the local server to make changes permanent. A two-phase commit protocol secures transactions that enable more than one servers. The system extends the ARIES algorithm [MHL+92] to support recovery and rollback. It involves redoing a client's progress at the server side using the client's log records.

SHORE adheres to a symmetric peer-to-peer server architecture. The system executes as a group of communicating processes via an RPC protocol. Each process, a SHORE server, implements page cache management, an object service, concurrency, and recovery. Applications may access data in a uniform fashion whether they reside at the machine or at another server. SHORE retains compatibility with Unix systems by providing object-based access over the file system.

Application calls to unswizzled objects invoke the object cache manager, which in turn requests the object from the local server via RPC, if it is not found in the cache. The local server searches the object at its disk and redirects the request to a remote server if it can not find the object either. The local server is responsible for managing lock and commit requests regarding the data resident at its disk. The access control component of each local server maintains object namespaces to enforce data protection.

2.1.6.3 Persistent stores

One motivation for persistent stores that support distributed environments is to offer a programming language agnostic interface to the data. The simplified data management that programmers in a distributed team are required to apply with a persistent system results in efficient collaboration and increased productivity. The most popular persistent stores are object-based. Objectstore [LLOW91] and SPOMS [MDRK93] are two typical examples of this class. The persistence model of Objectstore and SPOMS is based on memory mapping. Both stores provide transparent persistence, but they differ significantly with regards to the level of coupling with application programming languages; Objectstore

is strongly coupled with C++ while SPOMS is language independent. Objectstore's DDL includes a storage class keyword called *persistent* and a pointer to a database where a specific object should be stored. These two elements are part of a persistent object's declaration. SPOMS reveals no specific details regarding its API to the persistent store. Protection in Objectstore relies on C++'s type checking, which is applied to database objects in addition to transient ones, and on the operating system's virtual memory system that controls access to individual pages. Finally, SPOMS dispatches capabilities for newly created objects to the applications that requested them and applications use the provided capabilities to request objects.

The Objectstore object-oriented DBMS supports orthogonal persistence through a memory-mapped architecture and a storage class at the programming level. Objectstore is used mainly by C++ applications due to the language's popularity. Key to the tight integration of Objectstore with C++ is orthogonal persistence, i.e. that persistence is independent of an object's type and longevity. Objectstore's architecture results in high performance and code reusability since data handling happens by the same machine code sequences irrespective of the data's longevity. Objectstore extends C++ with an identifier called *persistent*, which specifies a storage class, to denote that a declared object should persist in the database. A pointer to the database is also provided as an argument in the object's declaration statement. A persistent object's manipulation in the program is the same as a transient one's.

Protection in Objectstore is based on access control to virtual memory pages and C++'s type system. Objectstore relies on the CPU's virtual memory hardware and the operating system's virtual memory system to control access to pages, which can be set to none, read, or write. Objectstore extends C++'s type checking to database objects. Thus, applications use a single type system that also takes into account persistent objects.

An Objectstore server is responsible for persistent storage, transaction processing, concurrency control, recovery and back-up. It provides two-phase locking at page granularity and utilises a write ahead log to support recovery. When a transaction commits, the updated pages are transferred to the server and stored to disk. The server signals an acknowledgement and the pages exit the address space but not the client cache in order to avoid requesting a page in case it is reused in a transaction. A two-phase commit protocol takes care of transactions that involve more than one servers.

The client cache is a core feature of distributed processing. It contains recently accessed database pages and resides in the client's virtual memory. When an application files a request for a page, the client cache is searched first. If the page is there, it is mapped into the client's virtual address space. Otherwise, the server places the page into the client's cache. Cache coherence is achieved by keeping track of pages in clients and assuring that they all share the same mode, either shared or exclusive. Objectstore utilises the machine's hardware, virtual memory addresses, and the operating system page-in mechanism to provide transparent, fast data transfers from disk to main memory.

Queries in Objectstore are strongly coupled with C++. Queries operate on object collections and result in other collections or an object reference. In addition to iterating and checking, Objectstore utilises indexes and a query optimiser. Indexes are more complex than relational indexes because they might traverse objects or even collections. Objectstore supports associative queries and object relationships through an explicit relationship facility.

SPOMS is a distributed, language independent, run-time system for persistent storage. The system utilises memory mapping together with Mach's External Memory Management Interface to provide persistence, sharing, and automated fetching of objects on demand.

The system provides transparent persistence and sharing of objects. The persistence model supports the uniform manipulation of objects at the programming language level, irrespective of an object's presence in memory. Objects are stored in native format; no conversion is required between storage formats. Objects stored in SPOMS are concurrently sharable between distributed applications. Sharing happens at real time, thus changes are immediately visible to all clients. This type of sharing

is finer opposed to the load-store programming routine.

The core concept in the SPOMS's storage system is a compiled class, which is a storage unit and language independent template. Creating a compiled class requires processing of object definitions and implementations. Any object can be persisted with SPOMS as long as a compiled class for the object is registered with the system. The run-time system retrieves object information and maps the object to processes' virtual address space.

Capabilities are distributed to users when an object or a compiled class is created. A capability is passed as a token to retrieve an object from SPOMS. The system uses the passed capability to find the location of the object and control access to it. This way objects are traced at run-time and replicated objects are transparently supported.

2.1.7 Performance Optimisations

Architectural support for orthogonal persistence emphasises a number of aspects including the management of the memory hierarchy, the efficient data communication to disk using a single representation, and efficient access to disk. These key aspects invite optimisations that can have a significant impact on the system's performance. A hierarchical memory model of computation [ACFS94] that models a processor's activity with respect to the levels of the memory hierarchy is in line with the rationale of an orthogonally persistent system. How the transparent movement of data between the main memory and secondary storage is implemented is another key performance issue. Two techniques are quantitavely investigated: pointer swizzling [Mos92] implementations and memory mapping where a performance study with traditional IO routines is presented [RPV09]. Finally, an approach that minimises disk accesses for retrieving objects is described [IJC01].

The Uniform Memory Hierarchy (UMH) model [ACFS94] is a hierarchical memory model of computation that studies performance issues of the memory hierarchy with respect to the activity of the processor during program execution. The model is parameterised by three aspects:

- the rate of increase in block size as we move farther from the processor,
- the ratio between the number of blocks and block size, and
- the cost of data movement between the memory modules of the hierarchy.

The first two parameters of the UMH model of computation are constants. The third parameter is a cost function. The cornerstone of UMH is the concept of *communication efficiency*, which distinguishes the computational tasks from the communicational tasks, which are related to data movements between the memory modules. A program's communication efficiency is measured by the ratio of processor activity during the program's execution, i.e. user and system time, to the program's total running time, that is real or elapsed time. In UMH all buses can be active at the same time to allow pipelining of operations. The paper [ACFS94] presents a generalised version of the model that incorporates parallelism.

The way programs perform read and write operations in a database or persistent object store can make a great difference in performance. Persistent objects can contain references to other objects that are resident in the same database or store. The references are called unique object identifiers or *oid*s. The possible profits incurred by converting oid references between objects resident in persistent memory into direct pointers for faster manipulation is examined in [Mos92]. This conversion is known as pointer swizzling. Pointer swizzling attempts to amortise up-front conversion cost by saving a little each time a reference is traversed. Pointer swizzling is a software address translation technique that is useful for the uniform treatment of data.

The same paper investigates five swizzling approaches namely, eager in-place swizzling, lazy in-place swizzling, eager copy swizzling, lazy copy swizzling, and non-swizzling. Eager and lazy swizzling

Table 2.3: Query languages and interfaces for querying OO data: table with columns signifying data location and rows signifying the software data model (objects or relations) against which queries are executed.

Data model	Programming language in-memory data structures	Orthogonal persistence across main and secondary storage	Secondary storage
Objects	PiCO QL, PQL [RIS+10], LINQ-to-Objects [MBB06], DEAL [RSI12], JQL [WPN06], SETL [SDSD86], OQL [CB00]	OPAL [MSOP86], ObjectStore [LLOW91], O++ [AG89]	XSQL [KKS92], MemSQL [Mem13], Daytona [Gre99], JPQL [KS09]
Relations	RelC [HAF+11]		ScalaQL [SZ10], LINQ-to-SQL [MBB06]

means prefetching and on demand respectively. In place swizzling happens in place in the object manager's buffers while copy swizzling creates a separate in-memory copy. The performance study's results show that pointer swizzling is 30% more expensive than non-swizzling if pointers are visited only once. The cost of swizzling is amortised between a few and a few tens of visits. In addition, the cost increases with the number of objects and object size. Consequently, the appropriate method to use depends on the size and shape of an application's objects, the frequency of traversals, and the way the application works with its objects. Given adequate memory copy swizzling provides flexibility, which is often an important aspect.

Algorithms for the efficient management of persistent storage, which minimise disk accesses and provide full recovery from failures can provide performance benefits for persistent systems [IJC01]. Traditional methods of storing data in file systems and databases generate significant overhead and are particularly inefficient in case of frequent operations on a large number of objects. The algorithms can be implemented to allocate storage on a raw disk or to allocate blocks that are stored in a single random access file. The former approach provides optimal performance, but is less portable than the latter. The algorithms can be used with three different memory management methods. The first makes use of in-memory free lists for fast disk block allocation and deallocation, the second maintains lists on disk, which contain both free and allocated blocks, e.g. for segregating blocks by size, and the third maintains only free lists on disk. An algorithm that minimises searches, splits, and coalesces is described. It can be combined with any of the above memory management methods.

A performance study between memory mapped files and traditional I/O techniques on applications that perform intensive data manipulation suggests memory mapping is more efficient [RPV09]. For the I/O techniques the *fread* function is used.

Memory mapped files allow a whole file to be mapped to a process's virtual address space. On demand of a byte from the file, the whole block of data containing the byte is copied directly to the respective process address space. On the other hand, traditional I/O incurs a system call in order to open the file, remove a disk block from the cache, copy the block on demand to the buffer cache, and finally copy it from the buffer to the process's address space.

The experiments involve the execution of data mining algorithms on a dual boot machine with Linux and Windows installed. The study reports better results for the memory mapping alternative under both operating systems independently of the number of dimensions, samples, and clusters of the algorithms.

2.2 Programming language query support

Our work involves the design and implementation of an external SQL relational interface to query program data structures. To do that, we combine a relational representation of objects and a synthesis of

object and relational query evaluation techniques. Our work is related to query languages and interactive interfaces for OO data (Section 2.2.1) relational representations for objects (Section 2.2.2), and queries against the non-first normal form (N1NF) relational data model (Section 2.2.3). The presented art attacks the object-relational impedance mismatch problem [Mai90], from diverse perspectives. Table 2.3 presents a categorization of query languages according to the software data model they support and the data location where query execution takes place.

2.2.1 Query languages and interactive interfaces for object-oriented data

Object-oriented DBMS and manipulation languages provide persistence, transactions, and querying native to the object model for applications written in OO programming languages. We focus on the expressiveness of database query languages for objects and the techniques utilized in query evaluation for graph-based data structures. Another line of work related to querying objects regards offering a declarative interface for performing programming tasks over collections of items, also called query comprehensions.

Providing an external interface for querying program data is tough for two reasons. First, general purpose programming languages typically do not provide an interpreter to evaluate a query at runtime in a safe manner. Second, such queries require synchronized access to data. For these reasons and because the provision of safe interactive queries is one of PiCO QL's contributions we also present here the state of the art for ad-hoc queries to program data. Besides the programming language, ad-hoc queries to main memory data are also offered by main memory DBMS with the expense of extra dependencies and overhead for storing the data. Our review includes this line of work too.

2.2.1.1 Database query languages for object-oriented data

In the past decades research in database query languages for object-oriented data has produced significant results. OPAL [MSOP86], ObjectStore [LLOW91, OHMS92], O++ [AG89], and XSQL [KKS92] are representative examples of this line of work. We note two differences between our work and database query languages presented in this section. First, PiCO QL employs a relational representation of program objects in 1NF and an interactive interface for executing SQL queries on them. This model is unique among the related work. Second, our work utilizes query optimizations offered by the relational query processor and leverages programming language algorithm implementations for specific container classes to boost query processing performance. On the other hand, the presented database query languages implement arbitrary indices. An appropriate implementation to support such indices in PiCO QL is a future work plan.

OPAL is an object-oriented database language. It is computationally complete and features auxiliary storage structures, indices, for associative access to object collections in the system. The language evaluates, in a sequential manner or using an index, a conditional expression specified through path expressions against an object collection and returns the objects that satisfy it.

ObjectStore is another system providing persistence orthogonal to object types and to queries against objects. It provides referential integrity for *has-one*, *has-many*, and *many-to-many* relationships between objects by placing pairs of inverse pointers within them. ObjectStore queries are expressions evaluated on one or more object collections; they return a collection, an object reference, or a boolean. Expressions include C++ expressions and selection predicates. Nested application of expressions is allowed to compose complex queries operating on embedded collections. Joins between collections on arbitrary attributes are supported without optimizations. ObjectStore provides associative queries, that is indices to optimize evaluation of expressions against indexed attributes. Indexed attributes can be deeply nested within objects. Indices are the main vehicle for applying optimizations in query processing. Both strategy selection and query evaluation depend on their presence.

O++ extends C++ with facilities for iterating through sets of objects in a declarative manner and orthogonal persistence with respect to object types. An iterator can be followed by a sequence of expressions, which are executed on each object of the set. Expressions include set selection, ordering, and recursion. O++ tracks type extents in collections and provides queries to either the extents or the hierarchies that derive from them. Iterators of different sets can be combined to form arbitrary joins.

XSQL is an SQL-like language for querying object-oriented databases. The language is based on F-logic [KL89], a logic with higher-order syntax but first-order semantics. XSQL features extended path expressions with variables ranging over classes, attributes, and methods and with selectors that retrieve data or schema. It provides explicit and implicit joins, nested subqueries, schema queries over inheritance hierarchies, set operations, and an equivalent of a GROUP BY clause. Queries return tuples or object ids after creating the objects.

2.2.1.2 Query comprehensions to programming language collections

Numerous approaches provide a declarative interface for transforming collections. LINQ to Objects [MBB06] is a branch of the polymorphic LINQ paradigm for object collections. LINQ is based on monads, a generalization of list comprehensions and, thus, implements the `map`, `filter`, `bind`, and `fold` higher order functions. PQL [RIS+10] is a purely-declarative logic sublanguage for querying collections, that is set, array, and map, in Java. It offers transparent parallelism. DEAL [RSI12] is also a logic-based declarative language but for expressing heap assertions at garbage collection time in Java. JQL [WPN06] and SETL [SDSD86] build on list comprehensions. JQL supports query optimizations, namely incrementalized caching for mutable collections.

PiCO QL is different from the aforementioned pieces of work in that it creates a relational representation for program data structures and provides an SQL interface for querying them rather than offer declarative queries in the existing programming language object model. In addition, PiCO QL's interface is interactive.

2.2.1.3 Ad-hoc queries to main memory D(B)MSs

Main memory databases [GMS92] leverage the memory's random access model to provide a performant service especially for hot data that is frequently accessed and has tight response time requirements. Main memory query processing leverages compact data structures, such as T-trees, in order to speed up queries. It also employs pointers for the cross-referencing of data, including object associations and foreign keys to relational table tuples [LC86]. In these cases pointers drive the use of pre-computed joins where, for example, a foreign key is substituted with the tuple(s) it points to. MemSQL [Mem13] and Daytona [Gre99] are main memory object oriented database systems. Contrary to D(B)MSs, which store program data in own containers, PiCO QL queries program data structures in place. A DBMS introduces unnecessary dependencies and overheads to an application that only requires in-memory data querying capabilities.

MemSQL supports SQL queries through an external query interface. MemSQL uses lock-free data structures in memory, translates SQL queries to C++ code for efficiency, swaps data to disk after a transaction succeeds, and behaves exactly like MySQL, with which it is wire-compatible.

AT&T's Daytona data management system with its high level fourth generation query language Cymbal also fits in this category. Cymbal is a powerful multi-paradigm language that includes ANSI 89 SQL as a subset. Daytona is based on a code-generation architecture, like PiCO QL. Ad-hoc queries in Cymbal translate into C programs complete with a makefile, compile, and execute against data stored in standard UNIX filesystems. Cymbal provides containers optimized for in-memory computation, which can also support applications required to operate in main memory at all times.

2.2.1.4 Ad-hoc queries to object collections

Some query languages for in-memory object collections support ad-hoc queries. JQL [WPN06] and OQL/C++ [CB00] provide limited support for interactive queries. LINQpad[1] provides an interactive interface to LINQ, including LINQ to Objects, for .NET languages. A debugger [LHS03] that supports dynamic queries on an object's state and relationships is available in Java. Contrary to these query interfaces, which adopt the programming language object model, PICO QL provides a relational interface to the program's data structures. In addition, while JQL and OQL/C++ have been developed for use within a general purpose, typically static, programming language, PICO QL has been specifically developed to support interactive queries. In contrast to LINQpad, which supports .NET languages, PICO QL supports C/C++ applications.

JQL supports dynamic queries against objects in Java containers. The query evaluator can accept an abstract syntax tree at run time and execute the query with the cost of runtime type checking. Although flexible queries can be constructed from user input, there is always the cost of programming the conversion of an untyped query expression to a typed one.

OQL/C++ supports queries against C++ STL containers. Because queries take the form of untyped strings, they can be input from an external interface. Query input parameter types, however, are checked at runtime and type violations trigger an error exception. Each query's result is specified as a parameter to the function that executes it. An error exception is generated if the actual result type differs from the specified one.

LINQpad uses the .NET `Reflection.Emit` functionality to build data contexts on the fly and .NET's code generator and code compiler to compile and execute LINQ queries at runtime.

Query-based debugging with an appropriate tool is available in Java. The debugger demonstrates capabilities for dynamic and on-the-fly queries in Java using load-time code instrumentation. A custom class loader generates and compiles custom debugger code. Queries expose object state and object relationships. The debugger has the ability to gather plain statistical data but does not support sophisticated operations such as aggregations, nested queries, and views offered by general purpose query facilities.

2.2.2 Relational representations for objects

Fundamental issues in creating a relational representation for objects concern the mapping of programming language classes, associations (*has-a, many-to-many*), inheritance (*is-a*), and polymorphism to relational constructs. Persistent data management of object-oriented applications using a relational DBMS is achieved through a set of object-relational mapping rules and specialized software systems that implement those rules bidirectionally. Two popular such systems are Hibernate for Java [BK06] and Microsoft's ADO .NET Entity Framework for .NET [MAB07]. Considerable effort has been put forward to support associations as first class programming language constructs [AGO91, BW05, Rum87]. Hawkins *et al.* [HAF+11] propose relations as classes that implement a high-level relational specification. Below we elaborate on these relational representations.

2.2.2.1 Object-relational mapping

Object-relational mapping [KJA93] studies the principles for transforming object-oriented data structures to relational ones stored in a relational DBMS in order to solve the object-relational impedance mismatch problem. Under most approaches in this area classes become relational tables and non-scalar values within classes, i.e. associations, give their place to relationship instances, i.e. primary key–foreign key chains between relational tables. *Many-to-many* associations require an intermediate table for tracking the association instances between the two tables.

[1] http://www.linqpad.net/

Three approaches have been proposed to implement the following relational mapping of inheritance and polymorphism.

1. Include the whole class hierarchy in a single table. Thus each column maps to each attribute of the class hierarchy and an extra column identifies the object type.

2. Each table maps to a class, abstract or concrete, so that the full information on an object is obtained by joins on the tables up the hierarchy.

3. Map each concrete class to a table containing all the attributes (both own and inherited).

Database operations happen through queries embedded in the programming language. JPQL [KS09], ScalaQL [SZ10], and LINQ to SQL [MBB06] query database data and store the result set in programming language containers. JPQL is a platform independent SQL-like language that executes queries against Java persistent entities stored in a relational database. LINQ to SQL extends .NET programming languages to generate at runtime an equivalent SQL query out of a LINQ query, which is executed against database tables. ScalaQL, on the other hand, implements an API within Scala in the form of a domain-specific language. Queries in ScalaQL are executed against the database tables. Both LINQ and ScalaQL provide statically type checked queries.

PiCO QL does not convert objects to relational tables; instead, it provides a relational representation on top of the programming language object model.

2.2.2.2 Relationships and associations as first class programming language constructs

Object associations as first-class programming language constructs [AGO91, BW05, Rum87] provide a relational representation for objects, that is a model native to the programming language for supporting object collaborations. The emerging relational model, groups object collaboration instances in a relation construct together with associated constraints. A relation is a set of object tuples and each tuple holds references to the collaborating objects. In effect, embedded object references that often account for object collaborations are not required.

Our work does not extend the programming language to support relational constructs. It provides a relational view of the underlying object relationships, that is associations, inheritance, and polymorphism.

2.2.2.3 Relational interface to programming language data structures

Two different lines of work provide a relational interface to programming language data structures: DEAL [RSI12] and RelC [HAF+11].

Object-oriented programming query languages, like DEAL [RSI12] mentioned earlier, automatically create a relational interface from an object-oriented model. The technique is simple; for each property B of an object A that is a reference to an object, a relation between A and B is generated in the form of a function $f : A \rightarrow B$. Listing 2.1 depicts an example.

PiCO QL also generates a relational interface from an object-oriented model but PiCO QL's interface is customizable through a user-provided relational specification. In this way, it provides flexibility as to the relational interface. In addition, PiCO QL's interface is queryable through SQL.

RelC is a language of decompositions that maps a relational specification of data structures and a set of associated functional dependencies to a concrete data representation for relations. Users can change data structure choices by changing the high-level relational specification, not the code that uses the relations.

RelC lies close to our work. Both in RelC and PiCO QL users provide a relational specification as input for the automatic implementation of a relational interface towards programming language data

Listing 2.1: Automatic relational interface generation from an object-oriented model

```
class List {
 ListNode first ;
 ListNode last ;
 int count;
}

class ListNode {
 Data d;
 ListNode next;
 ListNode prev;
}

class Data {
 int x;
 int y;
}

 List_first ( List ,ListNode)
 List_last ( List ,ListNode)
ListNode_next(ListNode,ListNode)
ListNode_prev(ListNode,ListNode)
ListNode_data(ListNode,Data)
```

structures. The main difference is that PiCO QL's relational specification represents the existing programming language model, while RelC uses the specification to produce a relational programming language interface to the specified data structures. PiCO QL's relational interface supplements the imperative programming language interface to provide interactive queries against mapped data structures.

2.2.3 N1NF relational data model

The first normal form principle of the relational model [Cod70] is deficient for modelling complex objects. This was the motivation for introducing nested relations [RKB85, SS86, TF86]. These provide an extension to the relational model where a relation may have relation-valued attributes. The extended data model requires relational query language extensions for executing recursive queries on relations.

Researchers have taken a number of approaches to provide queries on nested relations. A popular one regards restructuring a relation [TF86]. An *unnest* operator is used to flatten the relation until the target attributes reach the outmost level. Nesting, provided by the *nest* operator, may be necessary to transform back the result of an operation into the original structure of the relation. Roth *et al.* [RKB87] introduce *nest* and *unnest* operators to SQL. Their work leverages nested application of queries to avoid relation restructuring. Deshpande *et al.* [DL87] allow the use of a subrelation constructor within queries. Users can reference relation-valued attributes by creating and naming new relations while traversing the nested ones. Schek *et al.* [SS86] provide access to subrelations by using the projection operator as a navigator. An unnest step in the end exposes the target subrelation. In this approach, renaming may be required to avoid conflicts. Colby [Col89] introduces a recursive algebra to provide access to relation-valued attributes nested at arbitrary depth within a relation. The algebra includes recursive definitions of the relational operators; no restructuring of relations or special navigation operators are required. Google's Dremel [MGL+10] is a popular system whose query language is inspired by the aforementioned work. The notable aspect of the query language, which builds on SQL and is implemented efficiently for Dremel's nested columnar model, is the use of path expressions as input to the relational operators for navigating within nested records.

The common point between these pieces of work and PiCO QL is that they provide relational

queries that execute on a graph-based data model of arbitrary depth, that is a nested relational model on the one hand and a procedural programming or object-oriented model on the other. The difference is that the former extend the relational model and a relational query language, while PiCO QL maps a procedural code or object-oriented data model into an 1NF relational representation and provides standard SQL queries through the relational view against represented programming language data structures.

2.3 Diagnostic tools

Our work is related to relational interfaces within operating systems (Section 2.3.1), kernel diagnostics tools that use performance counters (Section 2.3.2) and program instrumentation techniques (Section 2.3.3), and application diagnostics tools (Section 2.3.4).

2.3.1 Operating system relational interfaces

Relations are part of the structure of some operating systems. The Barrelfish operating system features a general purpose system knowledge base (SKB) used for many core purposes, such as declarative device configuration [SBRP11] and cache-aware placement of relational database operators [GSS+13] among others. The knowledge base runs as a user-level process, collects information from the kernel about the various system services, and provides an interface to a constraint logic programming engine (CLP) written in Prolog. The collected information forms the input for the CLP algorithm and the produced solution is used to derive kernel data structures for the appropriate system service. Barrelfish's logic engine is of equivalent expressivity to relational calculus.

The Pick operating system [Bou86] is built around a multivalued database representation of hash files, which form nested representations. A file in Pick is a hash table consisting of records, called items in Pick's language. Each item consists of fields called attributes and each attribute consists of values. All files and items are organized uniformly. Elements in this hierarchy are variable-lengthed allowing Pick to form nested representations. The data manipulation language includes access queries with two commands LIST, which can express selection using a syntax similar to a WHERE clause, and SORT. Pick features no data typing and data integrity is left to the application programmer.

In contrast to the Barrelfish SKB and Pick, which use a declarative specification to derive kernel data structures, PiCO QL represents existing kernel data structures in relational terms and provides SQL queries on them through their relational representation.

Osquery [The14a] is a new relational instrumentation and monitoring framework developed by Facebook. Osquery, like PiCO QL, provides an SQL interface to operating system data leveraging SQLite's virtual table API. However, osquery is not part of the kernel and conducts analysis based on data exported by the operating system to user-level programs as opposed to PiCO QL, which accesses kernel data structures. In common with PiCO QL, osquery features an interactive command-line query console. In contrast to PiCO QL, which targets a single host, osquery features a host monitoring daemon that schedules queries for execution across an infrastructure. RocksDB, an embeddable key value store, caches query results to disk.

Relational-like interfaces exist at the operating system level for carrying out validation tasks. The work by Gunawi *et al.* [GRADAD08] contributes a declarative file system checker that improves over `e2fsck` [MJLF86], mainly because of its high-level abstraction. It imports the file system's metadata in a MySQL database, carries out file system integrity checks by means of SQL queries, and performs repairs accordingly. PiCO QL would alleviate the need for storing the file system's metadata in a database, but would require loading the metadata in memory. For a disk-based file system this would create a major overhead, but not for a main-memory one. In a potential near future PiCO QL could be a useful tool for verifying main memory file systems that use non-volatile RAM [Oik13].

Along the same lines, DeTreville [DeT05] proposes a checkable declarative system configuration approach, which removes the need for imperative system configuration updates in place. They define a declarative system model as a function, which configures a type checked system instance when applied in system parameters. Instances are automatically checked by policies, which express ad-hoc rules in a declarative manner. If the qualities of a system instance were part of its in-memory model, PiCO QL would be able to check its appropriateness by expressing the policies as SQL queries.

2.3.2 Kernel diagnostic tools that use built-in performance counters

Built-in performance counters are available at the hardware level and at the operating system level. Modern CPUs export hardware performance counters for use by diagnostic tools like Chopstix [BKFP08] and DCPI [ABD+97]. Counters are accessible from special-purpose registers that record hardware-related events, such as cache misses. Operating systems also include performance counters to record various aspects of system activity, such as process, I/O, virtual memory and network statistics. Tools like `top`(1), `iostat`(8), `vmstat`(8) and `netstat`(8) use the afore-mentioned performance counters respectively. While performance counters typically track system events, PiCO QL focuses on data structures state. PiCO QL can obtain read access to operating system performance counters like any other kernel data structure. The process for representing and accessing the counters is available online. [2]

2.3.3 Kernel diagnostic tools that use program instrumentation techniques

Program instrumentation techniques augment a system's code with special-purpose code, typically in an automated manner, to create system execution metadata. They consist of dynamic or binary code instrumentation, which replaces or rewrites binary code, and static or source code instrumentation, which adds instrumenting code to source code.

At the data collection phase, tools either consume the input sources or sample them. Sampling-based tools like GProf [GKM82] employ a sampling function to decide which events to record. Good sampling functions achieve near-accurate statistical approximation with negligible execution overhead [BKFP08].

Dynamic or binary code instrumentation is used more widely than static or source code instrumentation. Tools like DTrace [CSL04] and SystemTap [PCE+05] perform dynamic or binary code instrumentation. In contrast, LTTng [DD06] performs automated source code instrumentation. LTTng is a tracer; it records statically determined events in trace buffers written to files and offers post-mortem analysis. Source code instrumentation requires a fresh kernel build for each tracing attempt.

Binary code instrumentation tools are most related to this work. DTrace employs a virtual machine and a set of kernel modules that manage a probe infrastructure in the Solaris kernel. Specifically, the DTrace virtual machine accepts input in the form of the DTrace procedural programming language, which is reminiscent of C, and injects this code to the kernel's binary code at runtime. The injected code informs the referenced kernel modules to activate the appropriate probes for gathering the data required to compute the analysis task. Regarding safety, DTrace includes checks for illegal instructions, division by zero, invalid pointer references and other runtime errors. Its language is also designed in a form that prevents the expression of non-terminating code.

SystemTap is a meta-instrumentation tool, which extends and uses the `kprobes` [Moo01] kernel debugging infrastructure. It compiles scripts written in the SystemTap procedural language, which resembles C, into instrumenting loadable kernel modules and leverages the kernel's dynamic loading facility for injecting it. SystemTap utilizes memory reference checks like Dtrace, and checks for infinite loops.

[2] https://github.com/mfragkoulis/PiCO_QL/blob/master/src/Linux-kernel-mod/server/pico_ql_dsl.sql#L415

MS-Windows based operating systems provide the Windows Management Instrumentation (WMI) [TC03] infrastructure, which provides access to management data and operations. Management data are divided in namespaces, which contain classes. Object associations are modelled in association classes instead of embedded object references. WMI supports a query language, WQL, which implements a subset of SQL. A WQL query targets a specific WMI class hierarchy; joins are not allowed, nor cross-namespace queries or associations. WQL is in fact syntactic sugar over the object-oriented WMI data model.

System execution metadata typically capture events, but KLASY [YKC06] provides pointcuts, that is, predicates, which select matching function calls and/or data structure fields. KLASY is a dynamic aspect-oriented system, which uses `Kerninst` [TM99] as a backend. Kerninst is a dynamic instrumentation tool, which exports a low-level interface.

PiCO QL consumes directly lower level operating system data, that is, kernel data structures. The PiCO QL analysis process combines source code instrumentation for generating C functions to implement the virtual tables and dynamic query evaluation for running an SQL query through an SQL interpreter and query evaluator on the kernel's data structures. In contrast to WQL, PiCO QL exposes a relational model view of the underlying hierarchical data model. This is what makes it queryable through standard SQL. Instrumenting data structure accesses through declarative SQL queries is a novel approach.

PiCO QL's data access technique is probe-based but very different than typical probe-based instrumentation techniques, like DTrace's, in three ways. First, as mentioned, PiCO QL obtains read access to data structures; it does not instrument events as is the typical case with most tools such as SystemTap and DTrace. Second, PiCO QL requires a single access point for each data structure in order to track it. Event-based tools generate even thousands of probes to achieve instrumentation coverage [CSL04]. Third, PiCO QL's data structure tracking is performed manually at the module's source code. Its data access footprint is negligible compared to automatic binary instrumentation of event-based tools. Consequently, PiCO QL, like the other tools mentioned here, incurs zero performance overhead in idle state, because PiCO QL's "probes" are actually part of the loadable module and not part of the kernel. When PiCO QL is active, it forces idle threads upon context switches to all of the system's CPUs but one that executes the query. The performance overhead is the processing time of CPUs spent in idle state during query evaluation. PiCO QL employs checks for `null` and empty data structures when executing queries, which are type-safe. Joins through pointers are only allowed between compatible types.

2.3.4 Application diagnostic tools

Related to our work are sophisticated software diagnostics tools and scripting languages utilised for post-processing of tools' output. High-level interfaces attached to diagnostics tools typically take the form of browsers [Jos] and profilers [HJ91].

Recon [LSZE11] supports debugging of distributed systems through interactive SQL-like queries that target a relational model of system artefacts. Queries compile into instrumentation code that collects data from system artefacts at application execution replay for answering the queries. PTQL [GOA05] is a relational SQL-like query language for program traces with a relational data model. It compiles queries to bytecode instrumentation, which is injected to a running Java application to achieve online evaluation. In common with Recon and PTQL, PiCO QL features a relational data model, an SQL query interface, and online query evaluation. Differently to the other two, PiCO QL queries C, C++ application metadata stored in Valgrind data structures as opposed to compiling queries to byte code instrumentation of Java applications. Azadmanesh *et al.* [AH15] define a relational data model for program traces and import a Java program trace collected by BLAST, a dynamic analysis tool, into a relational database system in order to analyse the collected trace offline with SQL. PiCO QL queries data collected by Valgrind interactively at runtime and in place without storing them in a database system. PQL [MLL05] is another query language for program traces, but is oriented to static analysis

of Java applications whereas PiCO QL offers interactive analysis of an application's memory operations metadata.

The DTrace [CSL04] and SystemTap [PCE+05] diagnostic tools are also available for user applications. GDB [SS96], the GNU debugger, can attach to arbitrary processes and explore their in memory data structures with data break points. Compared to the above pieces of work, we identify two basic differences. First, PiCO QL provides a query interface on a software application's memory space metadata, not its data structures. Second, PiCO QL combines data structures through a relational interface rather than merely browse their values. In fact, The combined use of Valgrind's GDB server with PiCO QL can offer complementary advantages because it is possible to query data structures at specific time instances using SQL. On the other hand, the PiCO QL analysis process combines instrumentation of Valgrind's source code for hooking to Valgrind's data structures, PiCO QL source code generation for implementing the virtual table interface by compiling a PiCO QL DSL specification, and dynamic query evaluation for running an SQL query through an SQL interpreter and query evaluator on Valgrind tool's data structures.

Popular high-level interfaces for performance and diagnostics tools include browsers, such as KCachegrind [Jos] and interactive facilities to control profiling, such as Purify [HJ91]. Purify is a memory checker and performance profiler, which features a graphical representation to ease the detection of performance bottlenecks. For Cachegrind and Callgrind KCachegrind is available, a browser to navigate call graphs. In addition, Callgrind includes an interface for observing the status of a running program interactively to retrieve statistics or request dumping of profile data without stopping the program.

When the output, either final or intermediate, from a Valgrind tool does not provide a solution at its current shape then usually scripting languages, such as `awk` [AKW87], `Perl` [Wal00], and `Python` [Lut03] come into play. Alternatively, the output can be inserted into a relational DBMS to capitalise on its SQL interface. In fact, Memcheck can output errors in XML format, which makes it easy to combine with a relational DBMS. At the intersection of scripting languages and database systems, Perl's DBD::CSV driver [Thea] allows executing SQL queries on CSV files. Cachegrind's output matches this format. PiCO QL offers live interactive SQL queries to evolving metadata. In this way it obviates post-processing and shortens the write query-analyse results cycle significantly. Finally, with Memcheck we are particularly interested in leveraging internal metadata, such as shadow memory and memory allocation blocks, which are not output.

Chapter 3

Query interface design

Our contribution to the state of the art is the design and implementation of an interactive relational interface to a program's main-memory data structures that eliminates the overhead of storing the data in a DBMS, yet provides a typical relational database query engine's facilities, namely standard SQL views, queries, and optimizations. Our method maps imperative programming data structures into a relational interface. Object-oriented data models can use numerous sophisticated features; providing a relational representation of them can be difficult. This chapter describes how we map procedural programming and restricted OO data models to a relational interface. Models that don't use OO features are easier to represent in relational terms. For brevity, when we refer to objects or object-oriented programming or object-oriented data model we also include procedural programming and its constructs, such as C structs.

Our method for exposing an object-oriented data model through a relational interface addresses two challenges: first how to provide a relational representation of objects; second how to evaluate SQL queries against objects through their relational representation. The key points of the design that address these challenges include

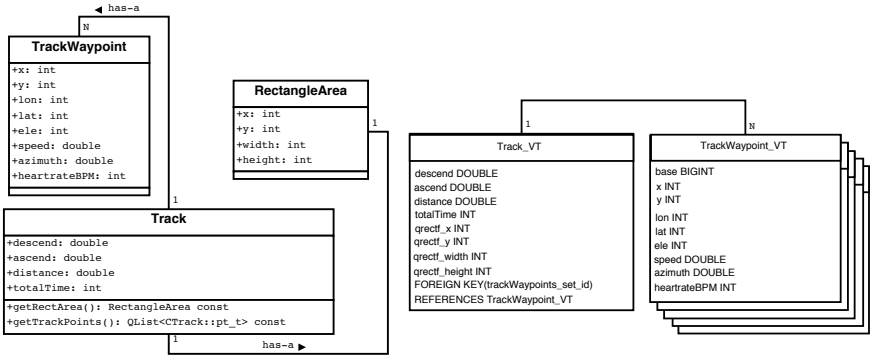
- rules for creating a relational representation out of objects (Section 3.1),
- a DSL for specifying relational representations and access information of objects (Section 3.2),
- an evaluation method for relational queries in an object-oriented environment (Section 3.3), and
- the formal description of an extension to the join relational operator introduced to achieve the mapping of object associations into a relational representation (Section 3.4).

Finally, Section 3.5 describes the software architecture of our relational interface.

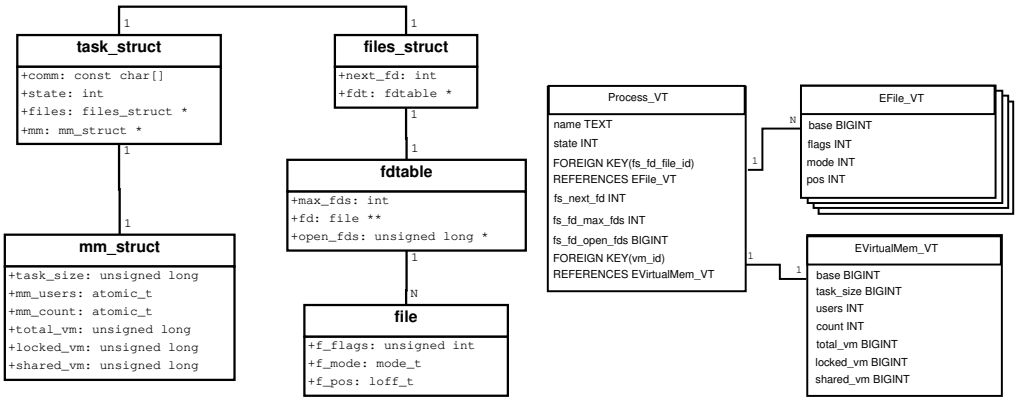
3.1 Relational representation of objects

The problem of transforming object-oriented data to their relational counterpart, and vice versa, has been studied thoroughly in the literature [BK06, MAB07]. We address a different but related, problem: how to define a relational representation of an underlying object-oriented data model. Providing a relational interface to object-oriented data without storing them in a relational database management system is not straightforward; the issue at hand is not the transformation of data from object structures into relational structures, but the representation of data in different models. In other words, we do not transform the object data; instead we want to provide a relational view on top of it. Issues that emerge in bidirectional transformations [CFH+09] between data models, such as the O-R data mapping, are not examined.

Figure 3.1: *has-a* association



(a) Class diagram (b) Virtual table schema



(c) Linux model (d) Virtual table schema

Figure 3.2: *Has-one* and *has-many* associations between data structures are normalized or denormalized in the virtual relational schema.

In our work the underlying data model can be object-oriented and therefore requires us to address features, such as inheritance and subtype polymorphism in queries. To do that we must solve the representation mismatch between relations and objects. Relations consist of a set of columns that host scalar values, while objects and their relationships form graphs of an arbitrary structure.

Our method defines three basic entities: objects, object associations and virtual relational tables. Specifically, it provides a relational representation of objects and object associations in the form of virtual relational tables. Objects can be unary class instances or containers grouping multiple objects. Object associations include *has-a* associations between objects, *many-to-many* associations between objects, and *is-a* associations and subtype polymorphism.

We exemplify our method in the context of a) a GIS application, which utilizes the above associations to model GPS data, such as tracks and waypoints, and b) the Linux kernel data model regarding processes, files, and virtual memory. The outcome is a queryable relational representation of the GIS application's and the Linux kernel's data model respectively.

3.1.1 Mapping *has-a* associations

Has-a associations are of two types: *has-one* and *has-many*. To describe them we define the *containing* object to be an object with some contents and the *contained* to be those contents. *Has-one*

associations include objects and references to objects. These are represented as columns in a virtual table that stands for the containing object. *Has-many* associations include collections of contained objects and references to collections of contained objects. These are represented by an associated table that stands for the collection of contained objects. Although the associated table's schema is static, *the contents of the associated table are specific to the containing object instance*: each instance of the containing instance has distinct contents. It is possible to define a separate table also for *has-one* associations.

Figure 3.1(a) shows the class diagram of a GIS application's object-oriented data model. It models a track's rectangle area that includes the track and track waypoints on maps using recorded coordinates from GPS devices. Figure 3.1(b) shows the respective virtual table schema. On the schema, each record in the map track table (Track_VT) represents a track. A track's associated bounding rectangle has been included in the track's representation: each attribute of the RectangleArea class occupies a column in Track_VT. In the same table, foreign key column *trackWaypoints_set_id* identifies the set of track waypoints that the track contains. A track's waypoint information may be retrieved by specifying in a query a join with the track waypoint table (TrackWaypoint_VT). This specification gives rise to an instantiation. The instantiation of the waypoint table is track specific; it contains the waypoints of the current track only. For another track another instantiation would be created. Thus, multiple instances of *TrackWaypoint_VT* implicitly exist in the background as Figure 3.1(b) shows.

In our method we provide each virtual table representing a nested data structure with a column named *base*, which takes part in *has-a* associations. The one side of the association is rendered by a foreign key column, which identifies the contents of an associated table as shown in the previous example, while the other side is rendered by the associated table's *base* column, which fulfills an appropriate instantiation. The *base* column is instrumental for mapping associations into a relational representation. We expand on this subject at Section 3.3.

Let's consider another example drawn from a procedural programming data model, Linux kernel's more specifically. Figure 3.2(a) shows a simplified kernel data structure model of the Linux kernel's files, processes, and virtual memory. Figure 3.2(b) shows the respective virtual table schema. There, a process's associated virtual memory structure (*mm_struct*) has been represented in an associated table (EVirtualMem_VT). The same applies for a process's open files (*file*), a *has-many* association, which is represented by EFile_VT.

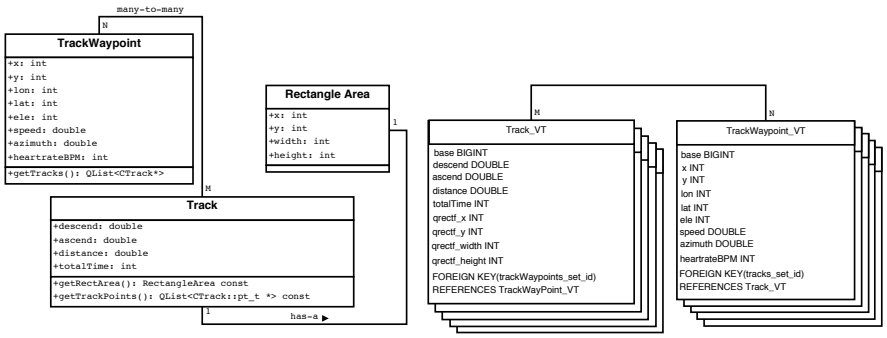
As we already mentioned an associated table is optional for *has-one* associations. In the same schema, for instance, the structure *fdtable* has been included in its associated *files_struct* and *files_struct* has been included within the associated process's representation. Consequently, each member of *fdtable* and *files_struct* occupies a column in Process_VT, that is, column *fs_next_fd* maps to *file_struct*'s *next_fd* member and columns whose name starts with *fs_fd_* map to *fdtable*'s members. By allowing to represent a *has-one* association as a separate table or inside the containing instance's table the relational representation flexibly expands or folds to meet representation objectives.

Combining virtual tables in queries is achieved through join operations. In table Process_VT the foreign key column *fs_fd_file_id* identifies the set of files that a process retains open. A process's open file information can be retrieved by specifying in a query a join to the file table (EFile_VT). By specifying a join to the file table, an instantiation happens. The instantiation of the file table is process specific; it contains the open files of a specific process only. For another process another instantiation would be created. Thus, as in the previous example, multiple potential instances of *EFile_VT* exist implicitly, as Figure 3.2(b) shows.

3.1.2 Mapping *many-to-many* associations

Many-to-many associations require no special treatment; they are treated similarly to *has-many* associations. Continuing with the GIS application example, suppose that, in addition to the above specification, a waypoint can be part of many tracks (Figure 3.3). The relationship can be described as

Figure 3.3: Many-to-many association



(a) Class diagram (b) Virtual table schema

a *has-many* association from both sides, that is, waypoint to track and vice versa. The effect in the virtual table schema is multiple instantiations for Track_VT as well, since it is now possible to identify the tracks that contain a specific waypoint.

In the relational model, a *many-to-many* relationship requires an intermediate table for the mapping. In our method virtual tables provide a relational representation of an application's data structures, but are only views of the data. For each instance of a Track (say Athens_Marathon) a distinct TrackWaypoint_VT (say Athens_Marathon_TrackWaypoint_VT) virtual table is instantiated. Similarly, for each instance of a TrackWayPoint (say Olympic_Stadium) a distinct Track_VT (say Olympic_Stadium_Track_VT) is instantiated.

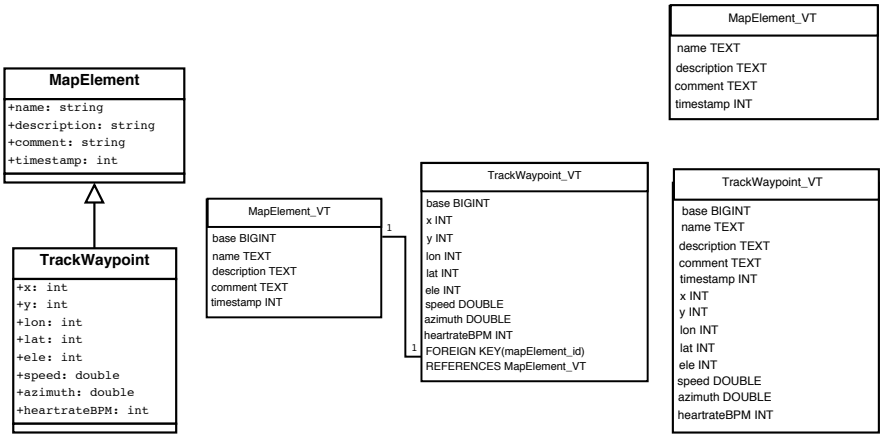
3.1.3 Mapping *is-a* associations

The support of *is-a* associations in the object-oriented paradigm provides powerful features, namely inheritance and subtype polymorphism. Our method offers two ways to incorporate OO inheritance and subtype polymorphism addressable at a relational representation of data structures; Figure 3.4(a) presents an example inheritance hierarchy. These ways correspond to two of the object-relational mapping strategies presented in Section 2.2.2.1. Currently our relational representation does not support multiple inheritance.

First, it is possible to represent each class in the inheritance hierarchy as a separate virtual table and use a relationship to link them (Figure 3.4(b)), following the *table to class* mapping approach. Second, it is possible to include inherited members as columns in each of the subclasses represented as virtual tables (Figure 3.4(c)), following the *table to concrete class* mapping approach.

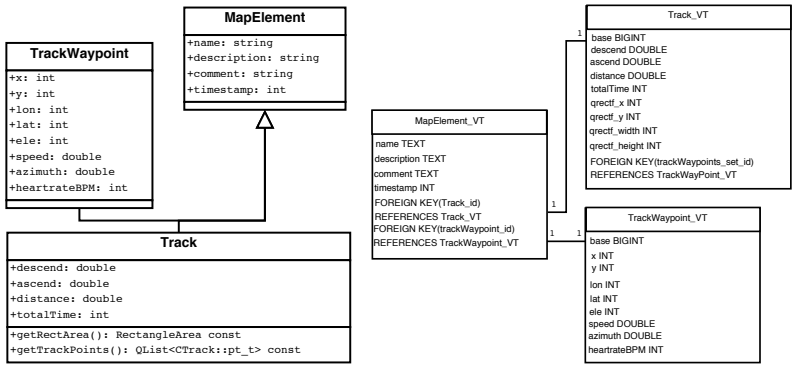
For polymorphic containers care must be taken in representing their contents that involve subtypes of the container element type. Suppose we represented a polymorphic container of map elements, that is each element could be a reference to a Track or TrackWaypoint, as in Figure 3.5(a). Virtual table MapElement_VT (Figure 3.5(b)), which represents the container of map elements, includes columns that map to members of MapElement type. In this way basic map element information can be retrieved directly from MapElement_VT. Virtual table Track_VT includes columns that map to members of type Track. Similarly virtual table TrackWaypoint_VT includes columns that map to members of type TrackWaypoint. A relationship instance links the virtual table representing the base class with the virtual table representing a derived class. Consequently track and waypoint information can be retrieved from the virtual tables representing the derived classes through the relationship instances.

Figure 3.4: Inheritance and subtype polymorphism support



(a) Class diagram (b) Table to class mapping (c) Table to concrete class mapping

Figure 3.5: Full support of polymorphic containers



(a) Class diagram (b) Virtual table schema

3.2 A domain specific language for defining relational representations of data structures

Our method provides a DSL for describing the mapping of the OO data into a relational model. The mapping is performed in two steps:

1. *struct view* definitions describe a virtual table's columns and
2. *virtual table* definitions link a *struct view* definition to a program's data structure type.

Together they compose a relational representation definition. The DSL's syntax is formally described in Figure 3.6 using Backus-Naur Form (BNF) notation.

3.2.1 Struct view definition

Struct view definitions (Listings 3.1 – 3.3) describe the columns of a virtual table. They resemble relational table definitions. Struct view definitions include the struct view's name and its attributes. Each attribute description contains the essential information for defining a virtual table column.

Figure 3.6: DSL syntax in BNF notation

; Virtual table definition

<virtual_table_def> **::=** 'CREATE VIRTUAL TABLE' <virtual table name>
'USING STRUCT VIEW' <struct view name>
['WITH REGISTERED NAME' <base variable>]
'WITH REGISTERED TYPE' <struct_type>
['USING LOOP' <loop_variant>] '$'

<struct_type> **::=** <container> | <object> | <struct> |
<primitive_data_type> **['*']**

<container> **::=** **(**<container_class> '<' <struct_type>
; if associative container
[',' <struct_type>**]** '>' **['*']) |** <C_container>

<container_class> **::=** 'list' | 'vector' | 'deque' | 'set' | 'multiset' | 'map' |
'multimap' | <any traversable container>

<C_container> **::=** <d_type> | <struct> ':' <d_type>

<d_type> **::=** <struct> | <primitive_data_type> **['*']**

<struct> **::=** **['**struct'**]** <struct name> **['*']**

<primitive_data_type> **::=** 'int' | 'string' | 'double' | 'char' | 'float' | 'real' |
'bool' | 'bigint'

<object> **::=** <class name> **['*']**

<loop_variant> **::=** <user defined loop variant>

; Struct view definition

<struct_view_def> **::=** 'CREATE STRUCT VIEW' <struct view name> '('
<column_def> {',\n' <column_def>} ')$'

<column_def> **::=** <primitive_column_def> | <struct_column_def> |
<struct_view_inclusion>

<primitive_column_def> **::=** <column name> <primitive_data_type> 'FROM'
<access_statement>

<struct_column_def> **::=** 'FOREIGN KEY(' <column name> ') FROM'
<access_statement> 'REFERENCES'
<virtual table name> **['**POINTER'**]**

<access_statement> **::=** <valid C/C++ expression> | 'tuple_iter'

<struct_view_inclusion> **::=** 'INCLUDES STRUCT VIEW' <struct view name>
['FROM' <access_statement> **['**POINTER'**]]**

; Standard relational view definition

<rel_view_def> **::=** <ANSI 92 SQL standard> '$'

Listing 3.1: Struct view definition - member methods as access paths

```
CREATE STRUCT VIEW MapElement_SV (
  name STRING FROM getName().toStdString(),
  description  STRING FROM getDescription(). toStdString  (),
  comment STRING FROM getComment().toStdString(),
  timestamp INT FROM getTimestamp())
```

Listing 3.2: Struct view definition - *is-a* table per class mapping (Figure 3.4(b)) and *has-a* relationship normalization (Figure 3.3(b))

```
CREATE STRUCT VIEW TrackWaypoint_SV (
  FOREIGN KEY(mapElement_id) FROM tuple_iter                    // is-a table per class
                             REFERENCES MapElement_VT,
  x INT FROM x,
  y INT FROM y,
  lon FLOAT FROM lon,
  lat FLOAT FROM lat,
  ele FLOAT FROM ele,
  speed FLOAT FROM speed,
  azimuth DOUBLE FROM azimuth,
  heartrateBPM INT FROM heartRateBpm,
  FOREIGN KEY(tracks_set_id) FROM get_tracks()          // has-a relationship
                             REFERENCES Track_VT)             // normalization
```

Listing 3.3: Struct view definition - *is-a* table per concrete class mapping (Figure 3.4(c)) and *has-one* flattening (Figure 3.3(b))

```
CREATE STRUCT VIEW Track_SV (
  INCLUDES STRUCT VIEW MapElement_SV FROM tuple_iter.value() POINTER,
                                                    // is-a table per concrete class
  descend DOUBLE FROM tuple_iter.value()->getDescend(),
  ascend DOUBLE FROM tuple_iter.value()->getAscend(),
  distance  DOUBLE FROM tuple_iter.value()->getTotalDistance (),
  totalTime  INT FROM tuple_iter.value()->getTotalTime (),
  FOREIGN KEY(trackWaypoints_set_id) FROM tuple_iter.value()->getTrackWaypoints()
                             REFERENCES TrackWaypoint_VT POINTER,
  INCLUDES STRUCT VIEW RectangleArea_SV FROM tuple_iter.value()->getBoundingRectF())
                                                    // has-one flattening
```

Column definitions are of two types, data column definitions and special column definitions for representing *has-a*, *many-many*, and *is-a* object associations. Data column definitions include the column's name, data type, and access path, that is, a C/C++ expression that retrieves the column value from the object. Except for a path expression to an object member, an access path can consist of a method call that preserves state (Listing 3.1). Special column definitions are of two kinds, *foreign key* definitions (Listing 3.2) and *struct view inclusion* definitions (Listing 3.3). Foreign key definitions include the column's name, access path, and associated virtual table, while struct view inclusion definitions include the column's name, the relational view of the included data structure, and the access path to this data structure.

The foreign key column definition (Listing 3.2) supports relationships between virtual tables that represent a *has-a* or an *is-a* association between the underlying data structures. A foreign key specification resembles its relational counterpart except that referential constraints are not checked in this context and no matching column of the referenced table is specified. This is because the foreign key column matches against an auto-generated column of the referenced virtual table, the *base* column. The *base* column does not appear in relational representation definitions because the DSL parser-compiler can understand when the column is required and generates the appropriate code for it.

Listing 3.4: Struct view definition includes the mapping of *has-a* inclusion (Figure 3.2(b)) in the virtual table schema.

```
CREATE STRUCT VIEW FilesStruct_SV (
 next_fd INT FROM next_fd,
 INCLUDES STRUCT VIEW Fdtable_SV FROM tuple_iter)
```

Listing 3.5: Struct view definition can contain calls to custom functions in column access paths.

```
 ...
long check_kvm(struct file *f) {
   if ((! strcmp(f->f_path.dentry->d_name.name,
                 "kvm-vm")) &&
       (f->f_owner.uid == 0) &&
       (f->f_owner.euid == 0))
     return (long)f->private_data;
   return 0;
}
$
 ...
CREATE STRUCT VIEW File_SV (
    ...
   FOREIGN KEY(kvm_id) FROM check_kvm(tuple_iter)
     REFERENCEStKVM_VT POINTER) )
```

Listing 3.6: Example of customized loop variant for traversing track waypoint list

```
#define TW_VT_begin(X,Y) X = Y->begin()
#define TW_VT_end(X,Y) X != Y->end()
#define TW_VT_advance(X) X++

CREATE VIRTUAL TABLE TrackWaypoint_VT
USING STRUCT VIEW TrackWaypoint_SV
WITH REGISTERED C NAME trackwaypoints
WITH REGISTERED C TYPE QList<CTrack :: pt_t >: QList<CTrack :: pt_t >:: iterator
USING LOOP for (TW_VT_begin(tuple_iter,base), TW_VT_end(tuple_iter,base),
                TW_VT_advance(tuple_iter))
USING LOCK QMutex(base->twp_mutex)
```

Listings 3.2 and 3.3 illustrate the supported inheritance mapping options in terms of the DSL. Listing 3.2 shows how to represent each class in the inheritance hierarchy as a separate virtual table (*table per class* mapping) and using a relationship to link them. *tuple_iter* is a language keyword that denotes a virtual table's tuple iterator. It selects an access path to reach a member nested to a represented object. If the access path's selector can be derived from the specification, the keyword can be omitted as in Listing 3.1. The tuple iterator can be also passed as a method argument. In Listing 3.2 we specify as access path the tuple iterator itself to reference each tuple's represented object. It is used to support an inheritance mapping through a relationship instance. The object's identity is adequate information for the mapping in these cases. On the other hand, Listing 3.3 shows the support for including inherited members as columns in each of the subclasses represented as virtual tables (*table per concrete class* mapping).

Including relational representations into others is useful for representing not only *is-a* but also *has-one* associations inline (Listing 3.3). Such is the case with a bounding rectangle included in a track's relational representation in Figure 3.3. Another example is the association between files and processes in the Linux kernel. Instead of mapping structures' *fdtable* and *files_struct* fields manually to Process_SV columns, one can use the INCLUDES STRUCT VIEW directive to reuse struct view definitions. Listing 3.4 shows how the relational representation of *fdtable* (Fdtable_SV) defined elsewhere

can be included to structure's *files_struct* relational representation (FilesStruct_SV). Mapping an associated *fdtable* to another data structure's relational representation besides *files_struct*'s requires only a similar INCLUDES STRUCT VIEW directive to the one presented in Listing 3.4.

Listing 3.5 demonstrates some more characteristics of the DSL. First, the special identifier POINTER is used to denote that the column maps to a pointer type. It is required to generate code that meets mapped data structures' types. Second, the extensive use of path expressions [FLU94] in access paths is a core aspect of the mapping method and of the query evaluation technique. Path expressions identify a data structure by specifying how to navigate to it in a graph of data structures. Third, accessing kernel information often requires additional constructs, such as calling kernel functions and macros. For instance, paging information in the kernel should be accessed through the kernel function `find_get_pages()` in order to access memory page utilization per process. The DSL accepts calls to functions and macros and defines a reserved keyword, *tuple_iter* to allow references to a virtual table's tuple iterator.

From our experience in querying data structures, it is also very useful to be able to write a block of code relevant to an access path in order to manipulate data or define conditions. For this purpose, a DSL file can start with code of the application's programming language, e.g., C code. This comprises include directives, macro definitions, and function definitions. Functions and macros can then be called from an access path context as shown in Listing 3.5. The defined function `check_kvm()` accepts an open file and tests whether the file belongs to a Kernel-based Virtual Machine (KVM) [KKL+07] virtual machine (VM) instance. KVM is manageable through file descriptors and a set of *ioctl* calls that operate on open file handles. Through this interface authorized users can manipulate VM instances and allocated virtual CPUs. Behind the scenes, a data structure modelling an open file – the usual *struct file* – maps back to a data structure modelling a KVM VM or virtual CPU instance depending on the *ioctl* call. We leverage this mapping to access KVM related data structures. With `check_kvm()` we ensure that the file is indeed an interface to a KVM VM or virtual CPU by checking the file's name and file ownership against the root user's account id.

3.2.2 Virtual table definition

Virtual table definitions (Listing 3.7) link a data structure to its relational representation. They carry the virtual table's name and information about the data structure it represents. Data structure information includes an identifier (NAME) and a type (TYPE); the identifier maps the application code data structure to its virtual table representation; the type must agree with the data structure's programming language type. The identifier is omitted if the data structure is nested in another data structure. A virtual table definition always links to a struct view definition through the USING STRUCT VIEW syntax.

An interface is required for traversing data structures to execute queries. The USING LOOP directive serves this purpose. In our approach, a container/iterator-based uniform abstraction is utilized that wraps diverse types of data structures. For example, Listing 3.6 makes use of a standard C++ STL loop variant for traversing STL-like collections. If this is the case the directive can be omitted as in Listing 3.7, but in absence of such a mechanism user defined macros in the first part of a DSL description can customize the loop variant by means of iterator methods (`declare, begin, end, advance`). This is particularly useful for traversing bespoke collections, which are found in the C programming language, and in domains such as systems programming, embedded applications, and scientific computing. The DSL parser will substitute references to `base`, which abstracts the data structure instantiation, with an appropriate variable instance. Listing 3.8 presents a more sophisticated example for iterating an array of open file objects using a bit array that records the indexes of open file objects in the array.

To synchronize access with other execution paths, the USING LOCK directive selects a locking method from those defined in the DSL. *twp_mutex* is an object of type mutex used to serialize access to the list of track points, that is *trackwaypoints*.

Listing 3.7: Virtual table definition

```
CREATE VIRTUAL TABLE TrackWaypoint_VT
USING STRUCT VIEW TrackWaypoint_SV
WITH REGISTERED C NAME trackwaypoints
WITH REGISTERED C TYPE QList<CTrack :: pt_t>
USING LOCK QMutex(base->twp_mutex)
```

Listing 3.8: Virtual table definition includes a customized loop variant for traversing file array.

```
#define EFile_VT_decl(X) struct file *X; \
        int bit = 0
#define EFile_VT_begin(X, Y, Z) \
        (X) = (Y)[(Z)]
#define EFile_VT_advance(X, Y, Z) \
        EFile_VT_begin(X,Y,Z)

CREATE VIRTUAL TABLE EFile_VT
USING STRUCT VIEW File_SV
WITH REGISTERED C TYPE struct fdtable : struct file *
USING LOOP for (
        EFile_VT_begin( tuple_iter , base->fd,
                (bit = find_first_bit (
                (unsigned long *)base->open_fds,
                base->max_fds)));
        bit < base->max_fds;
        EFile_VT_advance( tuple_iter , base->fd,
                (bit = find_next_bit (
                (unsigned long *)base->open_fds,
                base->max_fds, bit + 1))))
```

Listing 3.9: Lock/unlock directive definition

```
CREATE LOCK QMutex(m)
HOLD WITH m.lock()
RELEASE WITH m.unlock()
```

Ad-hoc in-place querying requires locking data structures to ensure synchronization for concurrent accesses. Since our method needs only read access to program data structures, serialization with concurrent write accesses is required. For the GIS application, the DSL's lock/unlock directives can be seen in Listing 3.9.

To support recurring queries and minimize time-to-query in these cases relational non-materialized views can be defined in the DSL using the standard CREATE VIEW notation, as exemplified in Listing 3.10. The particular view defines an alias for queries that access KVM virtual machine instances in the system. A similar view definition wraps the representation of virtual CPU instances.

3.3 Mapping a relational query evaluation to the underlying object-oriented environment

The relational representation of a data structure comprises one or more virtual tables. Each virtual table in the representation enables access to some part of a data structure using path expressions (see Listing 3.11 for an example of the underlying auto-generated routines). For example, a container of *Track* objects could be represented by rendering the *is-a* association between classes *MapElement* and *Track* via a table per class mapping — recall Figure 3.4(b); then the design would include two virtual tables, one for each class. The virtual table representing the *MapElement* type provides access to MapElement members, while the virtual table representing the *Track* type provides access to *Track*

Listing 3.10: Relational view definition identifies KVM VM instances.

```
CREATE VIEW KVM_View AS
  SELECT P.name AS kvm_process_name, users AS kvm_users,
    P.inode_name AS kvm_inode_name, online_vcpus AS kvm_online_vcpus,
     stats_id  AS kvm_stats_id, online_vcpus_id  AS kvm_online_vcpus_id,
     tlbs_dirty  AS kvm_tlbs_dirty, pit_state_id  AS kvm_pit_state_id
  FROM Process_VT as P
  JOIN EFile_VT  as  F
  ON F.base = P. fs_fd_file_id
  JOIN EKVM_VT AS KVM
  ON KVM.base = F.kvm_id;
```

Listing 3.11: The DSL compiler generates code for the virtual table interface.

```
int Process_VT_search (...)  {
   ...
  switch(col) {
  case 0:
     list_for_each_entry_rcu  ( tuple_iter , \
    & init_task ->tasks, tasks) {
        if (compare(tuple_iter->comm,operator,rhs)
                    add_to_result_set ();
    }
  case 1:
     list_for_each_entry_rcu  ( tuple_iter , \
    & init_task ->tasks, tasks) {
        if (compare(tuple_iter->state,operator, rhs)
                    add_to_result_set ();
    }
  case  ...
  }
}
```

members.

Another example drawn from the Linux kernel data model has Process_VT represent some fields of *task_struct*. Since *task_struct*'s *has-a* association with *mm_struct* has been modeled as a separate virtual table (EVirtualMem_VT), the latter provides access to the associated *mm_struct* fields. Member access is provided by path expressions according to the DSL specification.

Virtual tables may be combined in SQL queries by means of join operations (Listings 3.12, 3.13). Data structures may span arbitrary levels of nesting. Although the nested data structure may be represented as one or more virtual table(s) in the relational interface, access to it is available through the parent data structure only. The virtual table representing the nested data structure (VT_n) can only be used in SQL queries combined with the virtual table representing the parent data structure (VT_p). For instance, one cannot select a track's associated bounding rectangle without first selecting the track. If such a query is input, it results in an empty instantiation, that is an empty result set.

A join is required to allow querying of VT_ns. The join uses the column of the VT_p that refers to the nested structure (similar to a foreign key) and the VT_n's base column, which acts as an internal identifier. When a join operation references the VT_n's base column it instantiates the VT_n by setting the foreign key column's value to the base column. This drives the new instantiation thereby performing the equivalent of a join operation: *for each value of the join attribute, that is the foreign key column, the operation finds the collection of tuples in each table participating in the join that contain that value*. In our case the join is essentially a pre-computed one and, therefore, it has the cost of a pointer traversal. The base column acts as the activation interface of a VT_n, and guarantees type-safety by checking that the VT_n's specification is appropriate for representing the nested data structure.

Listing 3.12: Join query — querying *has-a* and *is-a* associations

```
SELECT *
FROM Track_VT
JOIN TrackWaypoint_VT
ON TrackWaypoint_VT.base = Track_VT.trackWaypoints_set_id
JOIN MapElement_VT
ON MapElement_VT.base = TrackWaypoint_VT.mapElement_id;
```

Listing 3.13: Join query — querying polymorphic containers

```
SELECT *
FROM MapElement_VT
JOIN Track_VT
ON Track_VT.base=MapElement_VT.track_id
JOIN TrackWaypoint_VT
ON TrackWaypoint_VT.base=MapElement_VT.trackWaypoint_id;
```

Listing 3.14: Relational join query

```
SELECT *
FROM TrackWaypoint_VT AS TW, WaypointInterest_VT AS WI
WHERE sqrt(power(TW.x − WI.x, 2) + power(TW.y − WI.y, 2)) < 1;
```

Listing 3.15: Relational join query shows which processes have same files open.

```
SELECT P1.name, F1.inode_name, P2.name, F2.inode_name
FROM Process_VT AS P1
JOIN EFile_VT AS F1
ON F1.base = P1. fs_fd_file_id ,
Process_VT AS P2
JOIN EFile_VT AS F2
ON F2.base = P2. fs_fd_file_id
WHERE P1.pid <> P2.pid
AND F1.path_mount = F2.path_mount
AND F1.path_dentry = F2.path_dentry
AND F1.inode_name NOT IN ('null' ,' ' );
```

Queries to polymorphic containers (Listing 3.13) require additional checks to ensure type-safety. Columns that map to derived type container members may be accessed by issuing joins to link the relational representation of the base class to the relational representations of the derived classes. Joins between virtual tables take the form of left outer joins; hence join operations trigger checks to match a container element's derived type against the type represented by a derived class' relational representation. In the case of a MapElement_VT container element that is a reference to a Track object instance, the type check performed when joining with Track_VT will succeed, since Track_VT represents type Track, while the type check resulting from joining with TrackWaypoint_VT will fail in a controlled manner. In the result set the columns of the relational representations of derived classes other than the one the element belongs to are populated with null values. Type checks ensure type consistency for each container element.

In addition to combining relational representations of associated data structures in an SQL query, joins may also be used to combine relational representations of unassociated data structures; this is implemented through a nested loop join [SKS06, p. 542].

Consider as an example the case where we would like to query the proximity between a collection of waypoints of interest and track waypoints. After incorporating the list of waypoints of interest in the system and creating its relational representation (WaypointInterest_VT) we could issue the query shown in Listing 3.14. Another example is presented in Listing 3.15. It returns which processes have the same files open.

3.4 Relational algebra extension for virtual table instantiations

Although relational algebra concerns sets of items, its operators can be applied to other types of item collections. According to Meijer *et al.* [Mei11], LINQ also builds on this approach.

Relational algebra [Cod70] defines the join operator that combines relations based on a common domain. Many variants of the join operator appear in the database literatute [SKS11], such as the *θ-join and equijoin*, the *semijoin*, the *antijoin*, the *division*, and *outer joins*.

The **natural join** ($R \bowtie S$) produces all combinations of tuples of relations R and S, for which the common attribute of the relations has the same value.

The **θ-join** ($R \bowtie_\theta S$) produces all the combinations of tuples of relations R and S where the relation between the joined attributes' value satisfies the θ binary relational operator, which is one of $<, \leq, =, \geq, >$. The **equijoin** is a specific case of a θ-join where θ is the equality operator ($=$).

The **semijoin** ($R \ltimes S$) produces all tuples found in one relation, for which a matching tuple exists in the other relation. Specifically, the left semijoin will produce all tuples of relation R that match a tuple of relation S. The definition of the right semijoin ($R \rtimes S$) is similar.

The **antijoin** ($R \triangleright S$) produces all tuples found in relation R that match with no tuples of relation S. It produces the complement of the respective semijoin.

The **division** ($R \div S$) produces all tuples of relation R containing attributes unique in R, for which all combinations of tuples with tuples in relation S exist in relation R.

The **outer joins** between two relations will produce all tuples of one or both relations regardless of a match for a tuple. If a match is not made, the attributes corresponding to the other relation will be filled with default values, usually NULL. There are three types of outer joins, **left** ($R \mathbin{⟕} S$) where no match is required for tuples of relation R, **right** ($R \mathbin{⟖} S$) where no match is required for tuples of relation S, and **full** ($R \mathbin{⟗} S$) where no match is required for either R or S.

Our method introduces an extension to the join relational operator, β,[1] to instantiate virtual tables that represent object collections accessible through others. Let Ξ be a virtual table representing an object collection (the *containing*) and Λ a virtual table representing an object collection (the *contained*) nested within the first collection. The instantiation of Λ relies on its *base* column, which is matched to a corresponding *foreign key* column of Ξ (fk_{base}). For each row of Ξ (say ξ_*) the operator instantiates a virtual table (Λ_*) representing the contained object collection (presented in Sections 3.1.1, 3.1.2). This operation casts the foreign key column value into the contained object collection's type and the object collection starting at this address is instantiated through the virtual table representing it. A formal description is presented in Listing 3.16.

Taking aside the instantiation part, the β-join resembles a relational equijoin, because the join attribute is explicitly mentioned, the join always uses the equality operator, and it produces all combinations of tuples of the two virtual tables.

3.5 Software architecture

Figure 3.8 depicts PICO QL's software architecture. It consists of the following components:

- the PICO QL API,

[1] The initial letter of the Greek word $\beta\acute{\alpha}\sigma\eta$ which means base

Listing 3.16: Formal description of virtual table instantiations

Let β be an extension to the join relational operator casting an untyped memory address into the address of an object, or a collection of objects,
π stand for the projection operation,
Ξ be a virtual table instantiating the containing object collection,
$\xi_1, \xi_2, ..., \xi_v$ be rows of virtual table Ξ, that is $\xi_1, \xi_2, ... \xi_v \, \varepsilon \, \Xi$, and
$\Lambda_1, \Lambda_2, ..., \Lambda_v, \Lambda$ be virtual table instantiations of the contained object collection
then

$$\beta(\pi_{fk_{base}}(\xi_1)) \rightarrow \Lambda_1,$$

$$\beta(\pi_{fk_{base}}(\xi_2)) \rightarrow \Lambda_2,$$

...

$$\beta(\pi_{fk_{base}}(\xi_v)) \rightarrow \Lambda_v,$$

$$\beta(\pi_{fk_{base}}(\Xi)) \rightarrow \Lambda, \quad \text{where} \quad \Lambda = \Lambda_1 \cup \Lambda_2 \cup ... \cup \Lambda_v$$

- the SQLite [Owe06] database query engine,
- an implementation of SQLite's virtual table API,
- a source-code compiler or generator, and
- a web interface module.

Most components communicate through the *delegation connector*, which signifies that a component's functionality is realised by the linked component where the connector's arrow leads. The *assembly connector*, which denotes the use of a component's interface by another component, appears at the top level where an application requires the services of PiCO QL and for producing the implementation of the relational interface for querying the application's data structures. In the latter case, the application uses the services of the PiCO QL compiler by providing a relational representation of the application's data structures in the form of the PiCO QL DSL. A description of each component follows.

The PiCO QL API provides the public interface to applications. Two methods of the API are required to get started with PiCO QL, one for registering a program data structure with PiCO QL and another for starting the query interface. The full API is given in Figure 3.7.

SQLite is an embeddable open source relational database system that supports SQL ANSI92. It features a virtual table module, which provides users with the building blocks for attaching arbitrary data sources to SQLite's query engine. Data sources can benefit from the relational facet by implementing SQLite's virtual table API. Because SQLite is embeddable, it is appropriate for use as a query interface plugin to applications.

PiCO QL implements SQLite's virtual table API in order to present a relational query interface to object-oriented program data. The virtual table API implementation splits in two parts. The one part concerns API methods fixed for all applications, such as the ones that open or close a virtual table. The other part implements the API methods specific to each application. These carry out query processing operations, such as searching a virtual table or return a column of it. Because virtual tables mirror program data structures, the query processing methods have to be generated according to the data structures at hand.

The source code generator or meta-programming module, which is implemented in Ruby, takes a specification of virtual tables linked to an application's data structures and generates the implementation of the query processing methods. The virtual table specification is written in the PiCO QL DSL.

Figure 3.7: The C++ PiCO QL API under the library's namespace

```
namespace picoQL {
  int  init (const char** pragmas, int  npragmas, int  port_number, pthread_t *t);
  void  register_data (const void * collection , const char * col_name);
  int  exec_query(const char *query, stringstream &s,
                  int (* callback )( sqlite3 *, sqlite3_stmt *, stringstream &));
  int  step_mute( sqlite3 *db, sqlite3_stmt *stmt, stringstream &s);
  int  step_text ( sqlite3 *db, sqlite3_stmt *stmt, stringstream &s);
  int  step_swill_html ( sqlite3 *db, sqlite3_stmt *stmt, stringstream &s);
  int  step_swill_json ( sqlite3 *db, sqlite3_stmt *stmt, stringstream &s);
  int  progress ( int n, int (* callback )( void *p), void *p);
  int  interrupt ();
  int  shutdown();
  int  create_function (const char *name, int argc , int  text_rep , void *p,
                        void (*xFunc)( sqlite3_context *, int , sqlite3_value **),
                        void (*xStep )( sqlite3_context *, int , sqlite3_value **),
                        void (* xFinal )( sqlite3_context *));
}
```

The web interface module is responsible for passing query input from the web interface, which is presented at a localhost port, to the SQLite engine. The interface hosts the schema of the relational representation to facilitate query input. SQLite observes that the query regards a virtual table and invokes the appropriate methods of the virtual table implementation to perform the query. Once a result set is ready, SQLite makes it available to the web interface for presentation to the user. The web interface is based on the SWILL library [LB02]. SWILL sets up a lightweight pseudo web server that presents HTML pages registered to it. In fact, the pages are C functions that blend HTML and C code for parameterising a page with dynamic data, for instance a query's result set. SWILL supports HTTP methods, such as GET and POST.

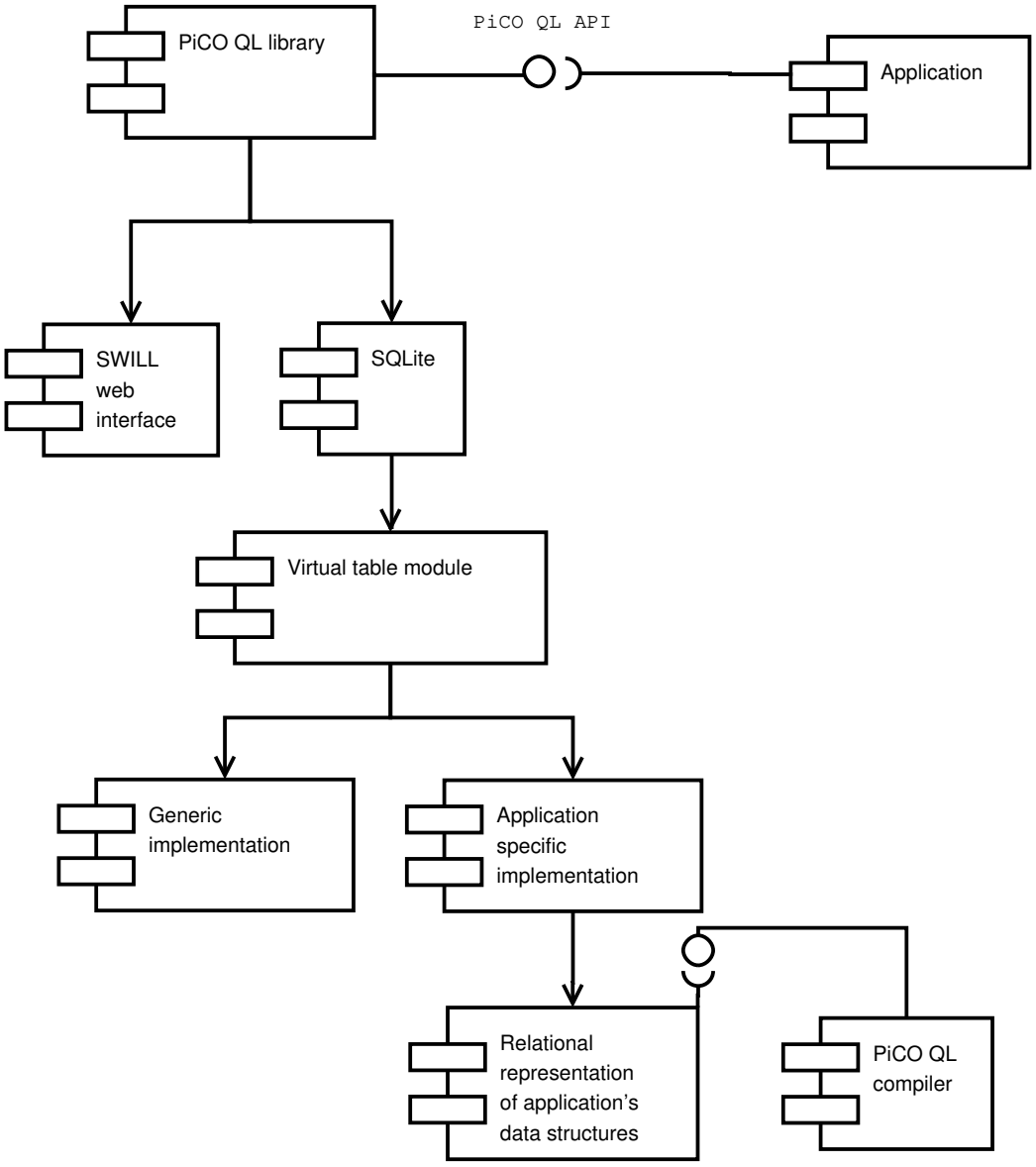


Figure 3.8: Software architecture

Chapter 4

Query interface implementation

PiCO QL is an SQL query library currently available for C/C++ applications, the Linux kernel, and the Valgrind instrumentation framework. The cornerstone of the PiCO QL implementation is a meta-programming technique devised

- to create a relational representation of arbitrary program data structures using a relational specification of these data structures, and
- to generate code for querying the data structures through their relational representation.

The meta-programming component of PiCO QL is presented in Section 4.1. PiCO QL leverages the virtual table module [The13] of SQLite [Owe06], an embeddable database engine, to support SQL queries to data structures. We describe our virtual table implementation in Section 4.2. Then we provide details on PiCO QL's SQL support using SQLite as point of reference (Section 4.3) and present the available query optimisations (Section 4.4). PiCO QL's user interface, which is described in Section 4.5, is web-based. It is implemented using the SWILL [LB02] library. We conclude the generic implementation description with the steps required to embed PiCO QL in applications (Section 4.6).

PiCO QL has also been implemented as a loadable kernel module (LKM) for Linux. We present our work there in a separate section, Section 4.7. Specifically, we describe the implementation challenges we encountered, the tailoring of the user interfaces for the kernel environment, and the security of our approach. We define consistency in the context of our work and state how we treat synchronization. In addition, we present proof of PiCO QL's reliability through the use of a Linux kernel test suite. Finally, we present PiCO QL's maintainability in the course of the kernel's evolution and discuss how PiCO QL can be ported to other operating system kernels.

We conclude this chapter with implementation aspects for including PiCO QL in the Valgrind instrumentation framework (Section 4.8).

4.1 Generative programming

Our solution to the problem of representing any C/C++ data structure as a virtual table is based on a user-driven relational specification and a meta-programming technique, generative programming. Specifically, a parser-compiler analyzes relational specifications written in the PiCO QL DSL and C/C++ program data structure information. Then it generates virtual table definitions and C/C++ callback functions for SQL queries. These functions implement constraint evaluation (Listing 3.11) and value retrieval for each of a virtual table's columns. The generative programming component of PiCO QL is implemented in Ruby [FM08].

4.2 Virtual table implementation

PiCO QL implements an SQLite virtual table module, that is a set of callback functions that specify a PiCO QL virtual table's behavior. These are: create, destroy, connect, disconnect, open, close, filter, column, plan, advance_cursor, and eof. The SQLite query engine calls these functions when performing a query on a PiCO QL virtual table. Hence queries are resolved by executing the implemented callback functions, which operate on the C/C++ application data structures.

In query processing, PiCO QL and SQLite share responsibilities. PiCO QL controls query planning through an implemented callback function (*plan*) and carries out constraint and data management for each virtual table. The hook in the query planner ensures that the constraint referencing the base column has the highest priority in the constraint set for the virtual table representing a nested data structure (VT_n) and, therefore, the instantiation will happen prior to evaluating any real constraints. This is important for ensuring integrity in query processing. SQLite performs high level query evaluation and optimization [Owe06, p. 360]. For the most part, query efficiency mirrors SQLite's query processing algorithms enhanced by simply following pointers in memory in the case of some join operations.

Having virtual table instantiations come into existence is not hard nor computationally intensive. A virtual table can be thought of as a concept whose rules are defined in the DSL. Data structures that adhere to the concept's rules use its representation and instantiate the virtual table. In effect, a virtual table is a structure that the query engine uses when the virtual table is referenced in a query. Multiple instantiations use the same structure and the structure stores a reference to its current instantiation; a new virtual table instantiation has the cost of a pointer dereference.

4.3 SQL support

PiCO QL supports all relational algebra operators as implemented by SQLite in [Owe06], that is the SELECT part of SQL92 excluding right outer joins and full outer joins. Queries expressed using the latter, however, may be rewritten with supported operators [Owe06, p. 76]. For a right outer join, rearranging the order of tables in the join produces a left outer join. A full outer join may be transformed using compound queries. The query engine is a standard relational query engine.

Providing relational views of data structures imposes one requirement to SQL queries. Virtual tables representing parent data structures (VT_p) have to be specified before VT_ns in the FROM clause. This stems from the implementation of SQLite's syntactic join evaluation and does not impose a limitation to query expressiveness. Let us clarify this through an example. The query in Listing 4.1 contains a join operation between FunctionNode and File. The former represents an array holding the instrumented program's function information with corresponding metadata. The latter represents a file, where a function's source code is contained. The join uses the foreign key column *file_id* of FunctionNode that refers to File and File's *base* column, which acts as an internal identifier. When an appropriate join operation references File's base column it instantiates File by setting the foreign key column's value to the base column. This drives the new instantiation thereby performing the equivalent of a join operation: *for each value of the join attribute, that is the foreign key column, the operation finds the collection of tuples in each table participating in the join that contain that value*. In our case the join is essentially a precomputed one and, therefore, it has the cost of a pointer traversal. Essentially, the foreign key column represents a reference to a data structure of type *file_node*. The base column receives this reference and acts as the activation interface of File. Finally, the *base* column guarantees type-safety by checking that File's specification is appropriate for representing a data structure of type *file_node*.

The virtual table API includes a function prototype for controlling query planning. Through the callback function that controls query planning PiCO QL can ensure that the constraint referencing the

Listing 4.1: Join query to combine function nodes and their associated file

```
SELECT *
FROM FunctionNode
JOIN File
ON File.base = FunctionNode.file_id ;
```

Listing 4.2: Query optimization – associative access

```
SELECT *
FROM Track_VT
WHERE track_title_key LIKE 'Athens Marathon';
```

base column has the highest priority in the constraint set for a virtual table representing a nested data structure, e.g., File, and, therefore, the instantiation will happen prior to evaluating any real constraints. In this way PiCO QL ensures integrity in query processing.

4.4 Query optimizations

PiCO QL benefits from SQLite's query rewrite optimizations, e.g., subquery flattening. The SQLite query optimizer [The14c] offers OR, BETWEEN, and LIKE optimizations in the WHERE clause, join optimizations, and order by optimizations associated with indices. However, these are not currently supported by PiCO QL; this is a future work plan. SQLite provides a virtual table API for indices and a command that scans all indices of a database where there might be a choice between two or more indices and gathers statistics on the appropriateness of those indices.

PiCO QL leverages algorithmic implementations of specific container classes offered by programming language libraries to provide query evaluation speedup when possible. For instance, queries on virtual tables representing containers that offer associative access, such as *map* or *multimap*, are optimized if a selection predicate on the container's key is specified. Let Track_VT represent a container of type *map* that associates a track's title to the respective track object. Then the query in Listing 4.2 takes advantage of the map's index to retrieve information about the Athens Marathon track. In addition, PiCO QL queries that execute against collections of items allow referencing items via their position by selecting a special auto-generated column of the virtual table representing the container, the *rownum* column that records the index in the container where a specific element is stored. For container classes providing random access such as *vector* PiCO QL utilizes this trait to boost query performance. Let TrackWaypoint_VT represent a container of type vector instead of a list, Listing 4.3 depicts a query leveraging the underlying container's random access feature. The query selects the ninth waypoint in order from the waypoint container.

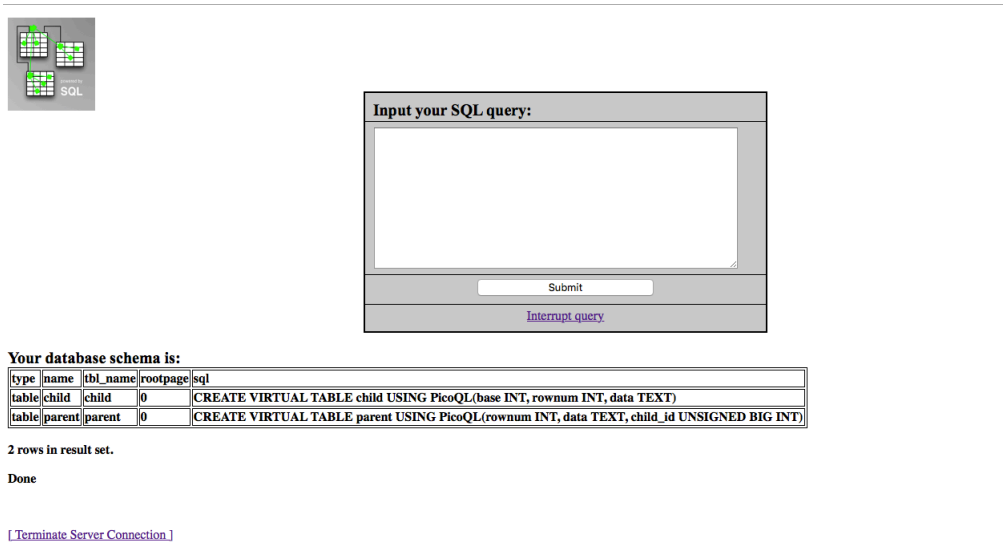
4.5 Query interface

A user interface is required in order to issue queries and view the results. For this we adopt SWILL, a library that adds a web interface to C/C++ programs. Figure 4.1 shows PiCO QL's web-based query interface. Each web page served by SWILL is implemented by a C function that blends HTML and C/C++ application code to present useful information about an application. For the query interface three such functions are used, one to input queries, one to output query results, and one to display errors. Using SWILL as a bridge the user interface can interact with SQLite easily through SQLite's C API. Queries are interpreted by the SQLite engine, which in turn calls the virtual table implementation's callback functions (section 4.2). Non-blocking I/O and polling for queries within a regularly executed function provide the basis for interactive queries. Our modest asynchronous query approach through polling

Listing 4.3: Query optimization – random access

```
SELECT *
FROM TrackWaypoint_VT
WHERE rownum = 9;
```

Figure 4.1: Web-based query interface



provides safe interactivity with a small penalty on timeliness. We have also added a REST interface that presents the query results in JSON format [Cro06], thus allowing the query interface to be easily used within AJAX [CPJ06] applications.

4.6 Embedding PiCO QL in applications

Prior to use with an application, PiCO QL requires a one-off setup process that includes three tasks presented in Figure 4.2. These are:

1. register each data structure with PiCO QL using a simple method call within the application's code and start the query library using another method call; these are shown in Listing 4.4.

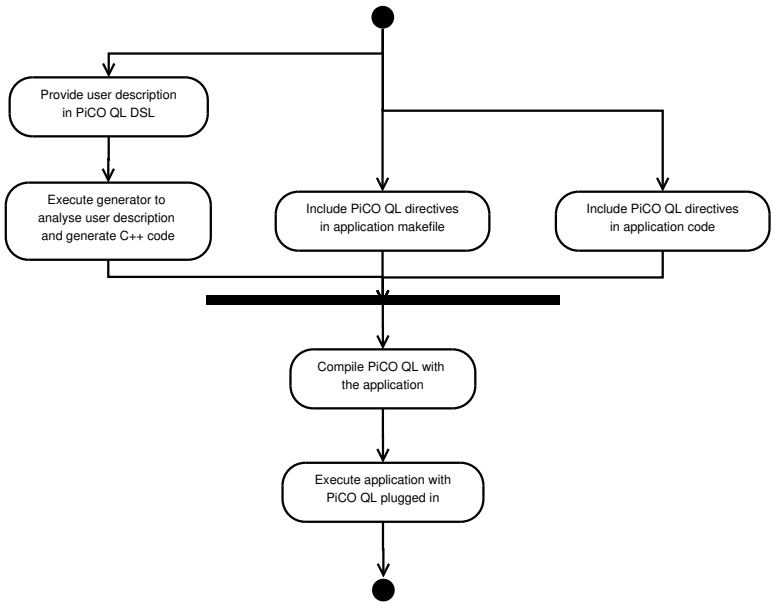
2. write a relational representation of the selected data structures in PiCO QL's domain specific language (DSL), which resembles relational table definitions. A number of applied examples are present in PiCO QL's codebase[1] and the project's wiki pages highlight the installation and application plugin process.[2]

3. compile the application together with PiCO QL by including PiCO QL's directives.

[1] https://github.com/mfragkoulis/PiCO_QL/tree/ a) PiCO_QL-application-release/examples b)PiCO_QL-kernel-release/src/Linux-kernel-mod c) PiCO_QL-Valgrind-release/src/Valgrind-mod

[2] https://github.com/mfragkoulis/PiCO_QL/wiki/Quickstart

Figure 4.2: Steps for plugging PiCO QL in an application.



Listing 4.4: PiCO QL directives

```
#include "pico_ql_search.h"
using picoQL;
...
register_data("trackwaypoints", &trackWaypoints);
init(NULL, 0, 8080, NULL);
```

4.7 Loadable module implementation for the Linux kernel

The process for implementing PiCO QL in the Linux kernel requires further action in addition to the steps listed in Section 4.6. Specifically, we implement the kernel module's initialization, permission, ioctl, open, input, output, close, and exit routines in order to provide an interface to PiCO QL. The query library's involvement in this step is to register virtual tables representing kernel data structures and start the query library at the module's initialization routine. It then receives a query from the module's input buffer, places a result set into the module's output buffer and terminates the query library through the module's exit routine. Finally, we compile PiCO QL LKM with the Linux kernel.

The main challenge in producing a loadable module is to compile SQLite with the Linux kernel. This includes omitting floating point data types and operations, omitting support for threads, eliminating SQLite's file I/O activity, and substituting user-space library function calls with the corresponding ones available to kernel code.

Floating point support and thread support for SQLite can be controlled through compile flags. File I/O is unnecessary since PiCO QL stores nothing. SQLite's in-memory mode alleviates this complexity except for the journaling mechanism, which can be turned off with a pragma query following the establishment of a database connection. SQLite's flexible configuration removed significant roadblocks with minimal effort.

The C library facilities available within the Linux kernel are spartan. Some user space library function calls are matched through a macro definition to implemented kernel functions. This is the case for the memory allocation routines. A few other library function calls are substituted with dummy

macro definitions and a couple are implemented in a straightforward manner by combining kernel functions.

A Unix kernel running PiCO QL supports dynamic analysis of accessible data structures with the following features.

SQL SELECT **queries** Queries conform to the SQL92 standard [DD97].

Type safety Queries perform checks to ensure data structure types are used in a safe manner.

Transparent consistency Queries can provide a consistent view of the kernel's state transparently, that is, without needing to protect data structure accesses with synchronization primitives.

No instrumentation PiCO QL does not affect the execution of kernel code, because it does not require its modification.

No fixed overhead PiCO QL's presence in the Unix kernel does not affect system performance. Section 5.2.3 presents a quantitative evaluation of PiCO QL's performance impact on kernel operation during query execution.

Relational views To reuse queries efficiently and store important queries, standard relational non-materialized views can be defined in the PiCO QL Domain Specific Language (DSL).

4.7.1 Query interfaces

Our kernel module uses the `/proc` file system for receiving queries and communicating query result sets. A high level interface that runs in user space is also available. Security is handled by implementing `/proc` callback functions that provide access control and synchronization is achieved by executing a query alone across the system's CPUs.

4.7.2 Security

Overall, there are three dimensions to PiCO QL's security, Confidentiality, Integrity, and Availability (CIA) [Bis04]. PiCO QL does not affect integrity since it does not provide create, update, and delete queries. PiCO QL's implementation ensures that data structures can only be read since callback function implementations for modifying queries (INSERT, UPDATE, DELETE, DROP) return with an error. SQLite does not support ALTER table queries for virtual tables and PiCO QL blocks CREATE virtual table queries from the user interface.

Concerning confidentiality, the kernel provides facilities for securing a module's use and access. First, loading and unloading a kernel module requires elevated privileges. Second, the `/proc` file system, PiCO QL's interface, provides an API for specifying file access permissions.

Our access control security policy covers interface access control and kernel access control, that is, access from other kernel modules. The mechanism we provide to restrict access to the kernel is based on the ownership of the `/proc` file. This should be configured to belong to a user or group with elevated privileges, who will be allowed to execute PiCO QL queries. When creating the PiCO QL `/proc` entry with `create_proc_entry()` we specify the file's access permissions for granting access to the owner and the owner's group. Additionally, the `/proc` interface provides a number of callback functions for manipulating an entry's behavior. Kernel modules communicate with each other and with the kernel leveraging exported symbols, that is, functions and variables. PiCO QL exports none, thus no other module can exploit PiCO QL's symbols.

Attacks to PiCO QL's availability can take two forms. Buffer overflows would allow arbitrary code execution but this is not evitable since we do not use any of the vulnerable functions to read user queries into our module. Even if we did, our module accepts and evaluates SQL queries only. Another

possible attack is to issue a query that engages in an infinite loop, however the available SQL dialect does not provide this vector.

4.7.3 Synchronized access to kernel data structures

In this work we are particularly interested in obtaining read access to data structures while they are in a correct state. This is the meaning we give to synchronized access. Ensuring synchronized access to kernel data structures is an important requirement for our work. In what follows we define consistency in the context of PiCO QL and describe how we treat synchronization. Finally, we discuss limitations of our work regarding synchronization and consistency.

4.7.3.1 Definition of consistency

We define consistency as the guarantee that all data structures referenced in a query do not change state during query evaluation. Consequently, our definition refers to a part of the kernel's current state for specific data structures. Synchronized access is a requirement for achieving consistency. It allows obtaining a kernel view in a safe manner because the kernel is at a correct state. Consistency adds that a query result should mirror a view of the system's current state. This capability supports important applications, such as the verification of invariants.

Consistency requires that a kernel state view is extracted while the referenced data structures are in the same consistent state as if by acquiring exclusive read access to them atomically, for the whole query evaluation period.

4.7.3.2 Synchronization in PiCO QL

PiCO QL provides consistent results transparently for data structures protected in critical sections provided these sections are not allowed to block. We discuss this limitation in Section 4.7.3.3. Transparently means that a query does not need to protect data structure accesses with synchronization. PiCO QL achieves transparent consistency by leveraging an existing mechanism, called *stop_machine()*,[3] of the Linux kernel, which allows a user-defined function to execute alone across the system's CPUs. This mechanism takes the query execution function as an argument and executes it with preemption and interrupts disabled while the remaining CPUs of the system are occupied by an idle thread. Essentially, the function executes in a non-blocking isolated setting until it finishes.

Our approach relieves relational interface designers of the PiCO QL Linux kernel interface from the burden of specifying locks and the order of lock acquisitions because the execution setting makes the necessary arrangements. Having worked with previous versions of PiCO QL that relied on holding and releasing locks for each virtual table we attest that this can be very difficult to get right in non-trivial cases, e.g., when data structures protected through blocking synchronization primitives are combined in a query. That said, this approach is much heavier than the actual locking requirements of a query since it is equivalent to holding every spinlock in the kernel. The performance related implications are explored in Section 5.2.3.

4.7.3.3 Limitations

Our approach does not work when a thread is allowed to block in a critical section, for instance by sleeping or waiting for an I/O operation. Although this is not allowed when using spinlocks in the Linux kernel, it is possible e.g., with semaphores. Semaphores allow a thread (including all other types of contexts in the Linux kernel) to leave the processor while in a critical section. In this case, a query may result in an inconsistent view. In addition, if a thread is blocked in a critical section, a query may result

[3] include/linux/stop_machine.h

in deadlock if it calls kernel code that uses the same semaphore, directly or indirectly. Note that this is no worse than our previous approach before adopting *stop_machine()* [FSLB14] that required the queries to use proper synchronization. In summary, our approach works with all types of spinlocks and read-copy-update and partially with semaphores. Currently the PiCO QL kernel interface does not reference kernel data structures protected through semaphores and does not invoke kernel code that uses semaphores so that all possible queries run normally.

We are currently examining possibilities for providing consistency transparently when kernel threads block in critical sections, but a good solution to this problem is not obvious. With the introduction of the *stop_machine* interface, synchronization is provided by the environment. It's important that the solution to the problem of supporting blocking synchronization primitives preserves this advantage. The candidate solution considered so far is to hold and release locks for data structures protected by blocking synchronization primitives. However, this solution drives us back to our previous design with explicit statements for handling synchronization provided through the USING LOCK clause of the PiCO QL DSL. In addition, this solution does not address indirect references to blocking synchronization primitives contained in kernel code that may be called in the course of a query.

Given that when PiCO QL queries are executed all other pointers are outside critical sections, any pointers used by PiCO QL should have valid values. However, to provide additional protection for pointers manipulated outside critical sections, e.g., in case of kernel bugs and erroneous virtual table definitions, PiCO QL checks pointers using the *virt_addr_valid()* kernel function before they are dereferenced to ensure they fall within some mapped range of page areas. Invalid pointers show up in the result set as INVALID_P. This protection measure is not perfect; pointer errors can still cause a kernel crash or wrong query results via, e.g., mapped but incorrect pointers.

4.7.4 Reliability

To validate the robustness of our implementation we used regression testing and the Linux Test Project [LTP14]. LTP contains an extensive suite of tests for the Linux kernel and related features. As regression testing dictates, first we executed the LTP tests on the system when PiCO QL was not loaded and gathered the results. A total of 1410 tests completed after more than 10 hours of operation; LTP reported 53 failed tests. Then, we executed the tests again having a set of 26 PiCO QL queries fire non-stop. The two LTP reports were alike, that is, PiCO QL did not introduce any side-effects to the system. In addition, the kernel's runtime lock validator [I. 14], which was active during the testing period, did not issue any complaints. Finally, PiCO QL sustained out of memory errors gracefully.

4.7.5 Deployment and maintenance

PiCO QL, similar to any loadable kernel module that is not built into the kernel, requires recompilation and reloading with each freshly installed version of the Linux kernel. The process is simple:

```
sudo make && insmod[4] picoQL.ko
```

Maintaining the PiCO QL kernel module in sync with kernel updates requires modifying PiCO QL's virtual relational schema through the PiCO QL DSL, not the kernel module's source code. A number of cases where the kernel violates the assumptions encoded in a struct view will be caught by the C compiler, e.g., data structure field renaming or removal.

Maintenance costs comprise adding C-like macro conditions in parts of the DSL data structure specifications that differ across kernel versions (Listing 4.5). The macros reference the kernel's version where the data structure's definition differs. Then they are interpreted by the DSL compiler, which generates code according to the underlying kernel version and the kernel version tag it sees on the macro. Thus, the cost of evolving PiCO QL as the kernel changes over time is minimized.

[4] `insmod` is the Linux kernel's facility for dynamic loading of kernel modules.

Listing 4.5: C-like macro condition configures the relational interface according to the underlying kernel version.

```
# if
KERNEL_VERSION > 2.6.32 pinned_vm BIGINT FROM pinned_vm,
#endif
```

4.7.6 Portability

PiCO QL requires a kernel's extensibility, processing monopolisation, and a way to communicate with user space. Many modern operating system kernels provide kernel modules, like UNIX's variants Linux, Solaris, Darwin, and FreeBSD, and Microsoft Windows. Processing monopolisation, that is code execution in only one of the system's CPUs at one time, is useful for performing tasks like garbage collection and system suspension in a clean fashion. Linux, for instance, employs the `stop_machine()` family of functions for this cause. FreeBSD implements the `stop_all_proc()` function, which stops all processes and `thread_single(struct proc *p, int mode)`, which allows the single-threaded execution of a process p across the system if mode is SINGLE_ALLPROC. Although modern operating systems provide features, such as hibernation, it is difficult to say how closed source operating systems like Windows and MacOSX implement them.

Kernels provide interfaces that loadable modules can use to collect input and publish output. A popular interface for Linux, Solaris and FreeBSD is the `/proc` interface. Darwin modules can interact with the outside world as pseudo-devices using for instance `/dev/random`. In a similar manner Windows modules register input/output callback functions among others and can reserve a place in the device stack. Applications communicate with them through standard file I/O system calls.

4.8 Implementation for the Valgrind framework

The main challenge in producing a working implementation is fitting SQLite to the Valgrind framework. This included omitting support for threads, eliminating SQLite's file I/O activity, and substituting standard library function calls with corresponding ones implemented by Valgrind.

Thread support for SQLite can be easily controlled through compile flags. File I/O is unnecessary since PiCO QL only stores the virtual table schema. SQLite's in-memory mode alleviates this complexity except for the journaling mechanism, which can be turned off with a pragma query following the establishment of a database connection. Thankfully, SQLite's flexible configuration removed significant roadblocks with minimal effort.

The C library facilities available within Valgrind are often customised. We matched some user space library function calls through a macro definition to implemented Valgrind functions. This is the case for the memory allocation routines. We substituted a few other library function calls with dummy macro definitions and implemented a couple of required functions by combining Valgrind functions.

Chapter 5

Empirical Validation

We use PiCO QL [FSL16] in three application domains [FSL15] and two system settings [FSLB14, FSL19]. Concerning the application domains (Section 5.1), QLandKarte is a GIS application that visualises GPS data, such as bike or running routes. The second one, Stellarium, is a virtual real time observatory of stellar objects. Finally, CScout [Spi10a] is a source code analyser and refactoring browser for collections of C applications. Within systems, PiCO QL serves as an ad-hoc diagnostic tool with a high-level programming language. We test its capabilities within the Linux kernel (Section 5.2) and the Valgrind instrumentation framework (Section 5.3).

5.1 Data analysis of C++ applications

We evaluate PiCO QL within three C++ applications based on the goal-question-metric method. We present the method (Section 5.1.1), the applications (Section 5.1.2), the measurements (Section 5.1.3), and the outcomes (Section 5.1.4).

5.1.1 Method

PiCO QL's evaluation follows the Goal-Question-Metrics approach [BCR94]. The evaluation's *goal* is to show how the current work compares to alternatives with respect to three important properties of query systems: programming language expressiveness, temporal efficiency, and spatial efficiency. Hence, three *questions* will drive the answers required to achieve the goal:

1. How does the current work compare to alternatives in terms of expressiveness?
2. How does the current work compare to alternatives in terms of runtime temporal efficiency?
3. How does the current work compare to alternatives in terms of runtime spatial efficiency?

Finally, the selected *metrics* consist of:

lines of code for measuring expressiveness

CPU *time* in seconds for measuring temporal efficiency.

storage space required to store the data in memory during execution for measuring spatial efficiency.

The properties mentioned are necessary and sufficient for drawing evaluation conclusions. Expressiveness marks a user's effort in writing queries; temporal efficiency quantifies query performance; spatial efficiency measures memory consumption in query processing. Memory consumption can potentially create problems in the face of large data sets or limited memory space.

Table 5.1: Projects used in evaluation

Project	Description and URL	Container		Number of measurements
		Type	Size	
Stellarium	Stellarium is a popular open source virtual real time observatory of stellar objects. It presents a realistic 3D sky view through its user-driven GUI. http://www.stellarium.org/	STL, Qt	100 (small)	100 runs
QLandKarte	QLandKarte is an open source GIS application that displays GPS data on a variety of maps. http://www.qlandkarte.org/	Qt associative	4K (medium)	10 runs
CScout	CScout [Spi10a] is a source code analyzer and refactoring browser for collections of C programs. http://www.spinellis.gr/cscout/	STL	1.1M (large)	5 runs

5.1.2 Use cases

PiCO QL has been evaluated on three large C++ projects, Stellarium, QLandKarte, and CScout (see Table 5.1). The instrumented containers are part of the standard STL library and the Qt framework.[1] The Qt framework provides a data model and optimized containers for GUI visualization.

SQL queries in Stellarium's use case concern planets and meteors stored in container classes of the Qt framework to ease management of the graphical aspects of the elements. Each container contains approximately 100 elements. The query interface for the Stellarium application is a nice companion for the GUI since it allows users to spot stellar objects on screen using an object's position, which is modelled as a 3D vector. For many stellar objects the GUI draws no markers or labels due to the former's small size.

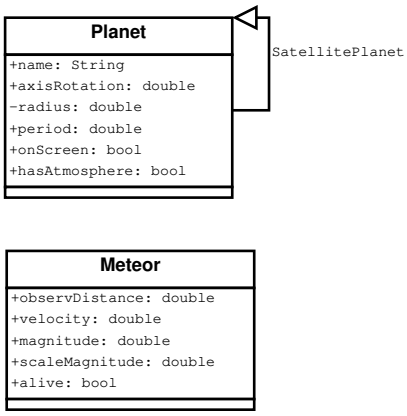
SQL queries in QLandkarte's use case concern waypoints of four thousand data elements stored in associative Qt framework containers. The query interface in QLandkarte allows users to analyze training data, such as heart rate and pace, as well as geographical characteristics, such as azimuth and altitude, collected by GPS tracking devices.

To ensure synchronized access to data we introduce mutex locks in the QLandkarte application's source code, which control access to the track, track waypoint, and map waypoint containers. This effort amounts to a handful lines of code and took a few minutes for each container. For the Stellarium and CScout applications we use PiCO QL single-threaded setting and occasionally poll for queries.

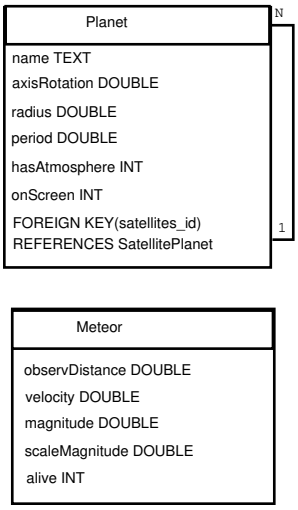
CScout can process workspaces of multiple projects (we define a project as a collection of C source files that are linked together) mapping the complexity introduced by the C preprocessor back into the original C source code files. Cscout can dump all identifiers of a C programming project's entire workspace with their relationships in the form of an SQL script containing the schema and data for the corresponding workspace. This script can then be uploaded into a relational database for further querying and processing. PiCO QL provides an SQL query interface to CScout's containers, which can be permanently stored in serialised format. This approach saves data import time compared to a database alternative. In this case study we use PiCO QL, C++, and MySQL to perform sophisticated queries on the extracted identifiers, files, functions, and function-like macros of the Linux kernel. MySQL is used on data dumped by CScout. CScout containers store 1.1 million identifiers and 89 thousand functions and function-like macros.

The evaluated projects form a good basis for performance evaluation due to their varying con-

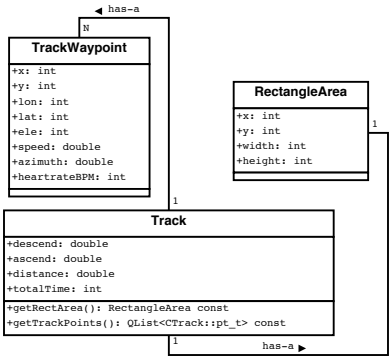
[1] http://qt-project.org/



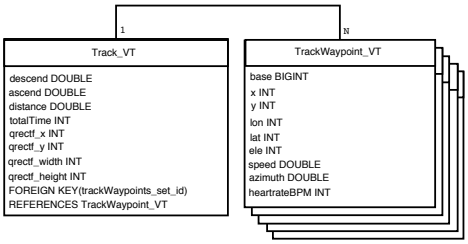
(a) Stellarium: class diagram



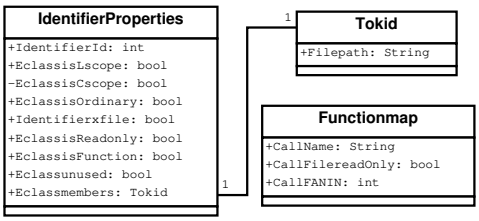
(b) Virtual table schema



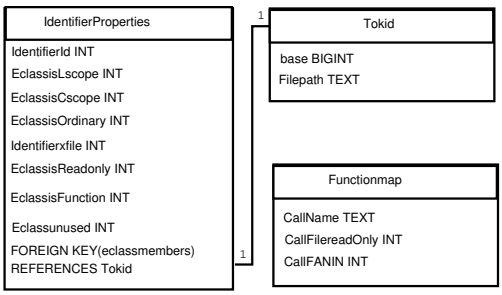
(c) QLandkarte: class diagram



(d) Virtual table schema



(e) CScout: class diagram



(f) Virtual table schema

Figure 5.1: Class diagram and virtual table schema for the Stellarium, QLandkarte, and CScout applications

tainer sizes and composite data models. With increasing container size the problem size increases. Thus, queries provide a good measure of PiCO QL's performance and its scalability. They indicate how gracefully PiCO QL consumes computer resources with respect to the evaluation criteria set, that is temporal and spatial efficiency. The projects' composite data models allow for sophisticated queries and, as such, provide a good testbed for measuring expressiveness.

The chosen projects are representative of situations where PiCO QL is most valuable. They carry out heavy online processing and access data stored as files on disk. Cscout can dump its containers to a relational database but PiCO QL provides SQL queries saving data import time compared to a database alternative. Overall, these projects store a wealth of information in main-memory containers, which can be queried to verify their state or extract knowledge out of the data.

PiCO QL requires some effort to embed in an application — recall Section 4.6. Assuming familiarity with the application, the basic cost involves writing the relational representation. Figure 5.1 shows the class diagram and relational representation of the Stellarium, QLandKarte, and CSCout applications respectively. Detailed guidance on embedding PiCO QL in applications is available online.[2]

Table 5.2: Use case (UC) identifiers and descriptions and respective PiCO QL queries for each project

#	Project	UC description	PiCO QL query
UC1	Stellarium	Retrieve meteors that are active at the moment and their observable distance from Earth is smaller than any other planet's distance from Earth	**SELECT** observDistance, velocity, magnitude, scaleMagnitude **FROM** Meteor **WHERE** alive **AND** observDistance < (**SELECT MIN**(distance) **FROM** Planet **WHERE** name **NOT LIKE** 'Earth');
UC2	Stellarium	Retrieve planets ordered by descending rotation that are currently on screen and rotate faster than their satellite planets	**SELECT** P.name, P.axisRotation, **MIN**(SP.axisRotation) **FROM** Planet **AS** P **JOIN** SatellitePlanet **AS** SP **ON** SP.base = P.satellites_id **WHERE** P.onScreen **AND** P.axisRotation > SP.axisRotation **GROUP BY** P.name **ORDER BY** P.axisRotation **DESC**;
UC3	Stellarium	Retrieve planets currently on screen whose satellites have atmosphere	**SELECT** P.name, P.radius, P.period, **COUNT**(*) **FROM** Planet **AS** P **JOIN** SatellitePlanet **AS** SP **ON** SP.base = P.satellites_id **WHERE** P.onScreen **AND** SP.hasAtmosphere **GROUP BY** P.name;
UC4	QLandkarte	Retrieve track points per azimuth ordered by maximum speed	**SELECT** name, azimuth, **max**(speed) **FROM** Track **JOIN** TrackWayPoint **ON** TrackWayPoint.base = Track.trackwaypoints_id **GROUP BY** azimuth **ORDER BY max**(speed);

[2] https://github.com/mfragkoulis/PiCO_QL/wiki/the-PiCO-QL-C%2C-CPP-app-tutorial

Table 5.2: Use case (UC) identifiers and descriptions and respective PiCO QL queries for each project

#	Project	UC description	PiCO QL query
UC5	QLandkarte	Retrieve track points per heart rate and elevation having more than 10 km/h average speed ordered by descending maximum speed	**SELECT** name, HeartRateBpm, ele, **max**(avgSpeed) **FROM** Track **JOIN** TrackWayPoint **ON** TrackWayPoint.base = Track . trackwaypoints_id **GROUP BY** heartRateBpm, ele **HAVING max**(avgSpeed) > 0 **ORDER BY max**(avgSpeed) **DESC**;
UC6	QLandkarte	Retrieve track points and map points having elevation above 20 meters	**SELECT** name, lon, lat , ele **FROM** Track **JOIN** TrackWayPoint **ON** TrackWayPoint.base = Track . trackwaypoints_id **WHERE** ele > 20 **UNION** **SELECT** name, lon, lat , ele **FROM** MapWayPoint **WHERE** ele > 20;
UC7	CScout	Retrieve local scope, non-class scope, ordinary, non-read only, non-function, used identifiers ordered by id	**SELECT** Identifierid **FROM** IdentifierProperties **WHERE** EclassisLscope **AND NOT** EclassisCscope **AND** EclassisOrdinary **AND NOT** Identifierxfile **AND NOT** EclassisReadonly **AND NOT** EclassisFunction **AND NOT** Eclassunused **ORDER BY** Identifierid ;
UC8	CScout	Retrieve file path of local scope, unused, non-read only tokens ordered by file path	**SELECT DISTINCT** Filepath **FROM** IdentifierProperties **LEFT JOIN** Tokid **ON** Tokid.BASE = IdentifierProperties .Eclassmembers **WHERE** EclassisLscope **AND** Eclassunused **AND NOT** EclassisReadonly **ORDER BY** Filepath;
UC9	CScout	Retrieve functions that are not called from read-only files and take no arguments ordered by their name	**SELECT** Callname **FROM** Functionmap **WHERE NOT** CallFilereadOnly **AND** CallFANIN=0 **ORDER BY** Callname;

5.1.3 Presentation of measurements

We compare PiCO QL queries to equivalent queries expressed using C++ constructs with respect to lines of code (LOC), CPU execution time, and query memory use. For the CScout case, we also carry out the measurements in a MySQL database with a default configuration and enabled indexes. Measurements of PiCO QL and C++ queries for the Stellarium and QLandkarte took place at another machine,[3] under identical (mostly idle) load. Each measurement represents the mean value obtained over 10 runs. Measurements of the PiCO QL, C++ and MySQL queries for the CScout case study took place at the

[3] Mac OS X 10.6.8, 2.4 GHz intel Core 2 Duo, 2 GB 667 MHz DDR2 SDRAM

same machine,[4] under (mostly idle) identical load.

The PiCO QL evaluation queries for the case studies are presented in Table 5.2. We select these evaluation queries on two grounds: (a) to show how PiCO QL can provide meaningful data extraction, and (b) to achieve coverage of SQL's operators, which is important for the evaluation. Evaluation queries include nested subquery, GROUP BY, HAVING, ORDER BY, join, and set operations.

The PiCO QL DSL, C++ and the MySQL queries are available online[5,6,7] except for CScout's C++ queries, which are embedded in CSout's plain query facility and are reproducible from CScout's GUI interface.

5.1.3.1 LOC measurements

Depending on the programming language, there are a number of ways to count lines of code. In fact, there is no standard way to count LOC for SQL queries. To compare LOC between SQL and C++ queries in an equitable manner, we use logical LOC for C++, which measures executable statements, and decompose an SQL query in lines, each line starting with a language keyword as can be seen in Table 5.2. How LOC for C++ are measured is presented in Appendix A.1 where *x* in comments denotes that the corresponding lines have not been accounted for the program's lines-of-code metric.

The code query size is listed in Table 5.3. The cost of supplying a relational representation for querying the application's data structures is accounted separately from PiCO QL measurements. It is located next to each application's name in the table header inside parentheses because it is a one-off cost amortized over use. CScout contains three interfaces for performing queries on processed identifiers, files, and functions. The measurements amount to the C++ code used for calculating each of the three evaluation queries. The code does not include the generic code base shared by all three interfaces (203 LOC) and the presentation layer.

5.1.3.2 CPU execution time measurements

In calculating CPU execution time (Table 5.3) we use the C *time* library function in PiCO QL and C++ queries and the time reported after each query for MySQL queries. MySQL time measurements are carried out with cold cache and exclude data import time.

The implementations of C++ queries in Stellarium and QLandKarte case studies leverage appropriate containers and algorithms where available. Specifically, we manage groupings in C++ using an associative container, such as a `map`. An additional group-by term requires an additional container embedded in the first. A `multimap` is convenient for accommodating a second group-by term, because it is ordered and provides the opportunity to group values of the second term for which the first term has the same value. We then use the STL library's `equal_range` algorithm to manage groups in the multimap. These containers are heavily used in evaluation queries along with provided algorithms. Evaluation queries do not make use of C++11 features, such as lambda functions.

5.1.3.3 Query memory use measurements

For calculating memory space during query processing (Table 5.3) we use the *ltrace* library call tracing program for PiCO QL and C++ queries in CScout under Linux, the maximum resident set size reported by the GNU time utility for MySQL queries, and the maximum resident set size reported by GNU *rusage* in Stellarium and QLandKarte tested under Mac OS X. Each MySQL query run took place on a freshly started database server without cleaning the machine's buffer cache; each measurement represents

[4] Linux 2.6.32-5-amd64, 4 Dual Core AMD Opteron(tm) Processor 880 CPUs, 16 GB RAM

[5] https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/Cscout

[6] https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/QLandKarte

[7] https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/Stellarium

Table 5.3: Query evaluation measurements

Case	Stellarium (25)			QLandkarte (24)			CScout (32)		
Listing	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9
LOC measurements									
PiCO QL	7	8	7	6	7	9	10	8	5
C++	14	22	20	37	49	23	425	425	433
MySQL	N/A	N/A	N/A	N/A	N/A	N/A	17	13	7
CPU execution time									
PiCO QL	749μs	1538μs	1020μs	74ms	68ms	45ms	2190ms	2220ms	270ms
C++	363μs	774μs	739μs	54ms	31ms	14ms	1070ms	1010ms	260ms
MySQL	N/A	N/A	N/A	N/A	N/A	N/A	123.30**s**	3950**ms**	570**ms**
Query memory use									
PiCO QL	61kB	59kB	41kB	171kB	167kB	166kB	12kB	102kB	6161kB
C++	4kB	12kB	49kB	90kB	90kB	72kB	7kB	6kB	6kB
MySQL	N/A	N/A	N/A	N/A	N/A	N/A	320kB	364kB	7155kB
Marginal query code size									
PiCO QL	234kB	234kB	234kB	201kB	201kB	201kB	941kB	941kB	941kB
C++	47kB	61kB	34kB	19kB	36kB	32kB	N/A	N/A	N/A
MySQL	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

the observed peak resident set size at the database client. Similarly, to get a reliable measurement of the peak resident set size from GNU time and GNU resource usage we restarted the application after each run.

5.1.4 Results

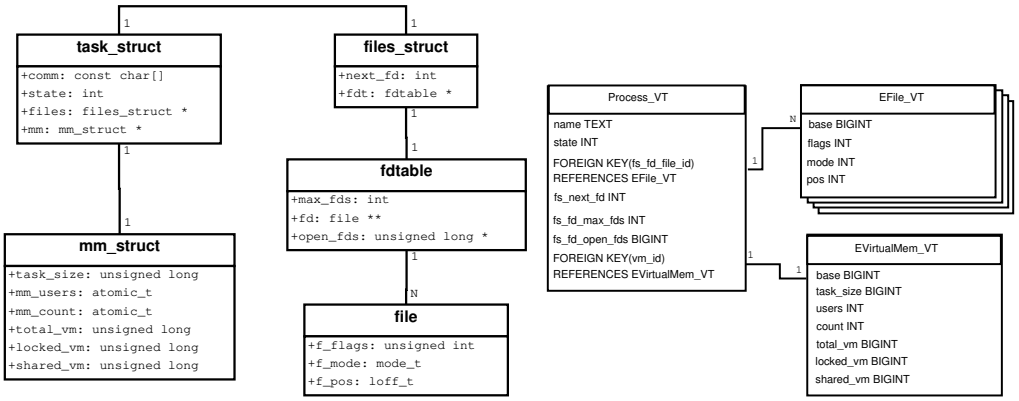
PiCO QL reduces the amount of code required for expressing a data analysis operation and even seems to have enhanced expressive power compared to the SQL statements used for querying the MySQL database. Each line in a PiCO QL query corresponds to four lines of C++ code on average. GROUP BY clauses cause a significant fraction of this expressiveness gap; LOC ratio explodes to 6:1 for GROUP BY queries (UC4, UC5). Notably, C++ data type definitions account for 1/4 of the overall expressiveness gap.

The difference, in favour of PiCO QL, that we observe between PiCO QL and MySQL queries is explained by the reduced normalization opted for in PiCO QL queries. Specifically, we have chosen to model 1:1 associations in the same virtual table (recall section 3.1.1 for further explanation of PiCO QL modelling), whereas in a typical relational schema there would be a table for each entity participating in the association and a primary key / foreign key relationship instance to accommodate the association. As a result, each PiCO QL evaluation query saves two joins or four LOC on average.

CPU execution time measurements show that querying using C++ programming constructs is generally more efficient, but PiCO QL performance scales as well as C++ code with increasing data sizes. C++ is twice as fast as PiCO QL on average regardless of a query's operations. MySQL with indexes enabled is efficient but for some operations object-oriented query processing with PiCO QL is much faster (UC7). This stems from the way PiCO QL is overlaid on the CScout data model. In the last two CScout evaluation queries, PiCO QL is twice as fast as MySQL capitalizing on its in-memory operation and native query processing against application data structures.

PiCO QL is slightly outperformed by the C++ query implementation for the query presented in UC9. Because the corresponding query returns a large result set ($>$ 20000 records), a possible explanation is that PiCO QL closes the performance gap by leveraging database result set presentation techniques.

Query memory use measurements show that MySQL queries consume most memory and C++ queries consume minimum space for simple cases. PiCO QL stands in between having a modest mem-



(a) Data structure model (b) Virtual table schema

Figure 5.2: Linux kernel models

ory footprint.

In sum, our measurements show that the use of PiCO QL is practical in terms of temporal and spatial efficiency. PiCO QL's query execution time scales well with increasing data input; comparably to C++ code. Besides interactivity, PiCO QL provides a competitive solution with respect to the properties examined in the evaluation.

Marginal query code size is considerably larger for PiCO QL compared to each C++ query but this is actually the space for the library as a whole. The cost is amortized over an arbitrary number of queries.

5.2 Diagnostics in the Linux kernel

We evaluate PiCO QL within the Linux kernel for diagnosing issues with its operation [FSLB14]. We present the method (Section 5.3.1), the use cases (Section 5.2.2), the measurements (Section 5.2.3), and the outcomes (Section 5.3.4).

5.2.1 Method

PiCO QL's evaluation in the Linux kernel involves two axes of analysis, use cases (Section 5.2.2) and quantitative (Section 5.2.3). Use cases examine how PiCO QL can aid system diagnostics. Figure 5.2 presents a part of the data structure model and the respective relational representation. The quantitative evaluation presents the execution cost for a variety of queries with diverse computation requirements and the overhead of executing queries to user processes.

5.2.2 Use cases

The following use cases demonstrate how PiCO QL can be used to aid system diagnostics activities in three areas: operation integrity, security audits, and performance evaluation.

5.2.2.1 Operation integrity

PiCO QL aids operation stability and data retrieval by providing a high level SQL interface to query the live state of data structures at arbitrary times and identify possible problems and deficiencies. The

Linux kernel bug tracking system[8] hosts an extensive list of bugs. The two use cases UC1 and UC2 referenced in Table 5.4 are associated with bugs that come from that list.

Table 5.4: Use case (UC) identifiers and descriptions and respective PiCO QL queries

#	UC description	PiCO QL query
UC1	Child process dies due to `kill` signal that did not originate from its parent process. [https://bugzilla.kernel.org/show_bug.cgi?id=43300]	**SELECT** P.name, P.pid, P.state, EP.name, EP.pid, EP.state **FROM** Process_VT **AS** P **JOIN** EProcess_VT **AS** EP **ON** EP.base=P.parent_id **WHERE** P.pdeath_signal = 9 **AND** EP.pdeath_signal = 0 **AND** EP.exit_state = 0;
UC2	Aggregate thread I/O statistics in parent. [https://bugzilla.kernel.org/show_bug.cgi?id=10702]	**SELECT** P.name, P.pid, P.tgid, PIO.read_bytes_syscall, PIO.write_bytes_syscall, **SUM**(ETIO.read_bytes_syscall), **SUM**(ETIO.write_bytes_syscall), PIO.read_bytes_syscall + **SUM**(ETIO.read_bytes_syscall), PIO.write_bytes_syscall + **SUM**(ETIO.write_bytes_syscall) **FROM** Process_VT **AS** P **JOIN** EIO_VT **AS** PIO **ON** PIO.base=P.io_id **JOIN** EThread_VT **AS** ET **ON** ET.base=P.thread_group_id **JOIN** EIO_VT **AS** ETIO **ON** ETIO.base=ET.io_id **GROUP BY** ET.tgid;
UC3	Identify normal users who execute processes with `root` privileges and do not belong to the `admin` or `sudo` groups.	**SELECT** PG.name, PG.cred_uid, PG.ecred_euid, PG.ecred_egid, G.gid **FROM** (**SELECT** name, cred_uid, ecred_euid, ecred_egid, group_set_id **FROM** Process_VT **AS** P **WHERE NOT EXISTS** (**SELECT** gid **FROM** EGroup_VT **WHERE** EGroup_VT.base = P.group_set_id **AND** gid **IN** (4,27))) PG **JOIN** EGroup_VT **AS** G **ON** G.base=PG.group_set_id **WHERE** PG.cred_uid > 0 **AND** PG.ecred_euid = 0;

[8] https://bugzilla.kernel.org

Table 5.4: Use case (UC) identifiers and descriptions and respective PiCO QL queries

#	UC description	PiCO QL query
UC4	Identify files open for reading by processes that do not currently have corresponding read access permissions.	**SELECT DISTINCT** P.name, F.inode_name, F.inode_mode&400, F.inode_mode&40, F.inode_mode&4 **FROM** Process_VT **AS** P **JOIN** EFile_VT **AS** F **ON** F.base=P.fs_fd_file_id **WHERE** F.fmode&1 **AND** (F.fowner_euid != P.ecred_fsuid **OR NOT** F.inode_mode&400) **AND** (F.fcred_egid **NOT IN** (**SELECT** gid **FROM** EGRoup_VT **AS** G **WHERE** G.base = P.group_set_id) **OR NOT** F.inode_mode&40) **AND NOT** F.inode_mode&4;
UC5	Retrieve executable program binary formats not used by any processes.	**SELECT** load_bin_addr, load_shlib_addr, core_dump_addr **FROM** BinaryFormats_VT **WHERE** load_bin_addr **NOT IN** (**SELECT** load_bin_addr **FROM** Process_VT **JOIN** ProcessBinaryFormat_VT **ON** base = binfmt_id);
UC6	Return the current privilege level of each KVM virtual online CPU and whether it is allowed to execute hypercalls.	**SELECT** cpu, vcpu_id, vcpu_mode, vcpu_requests, current_privilege_level , hypercalls_allowed **FROM** KVM_VCPU_View;
UC7	Return the contents of the PIT channel state array.	**SELECT** status_latched , status , mode, bcd, gate , count_load_time, count_latched **FROM** KVM_View **AS** KVM **JOIN** EKVMArchPitChannelState_VT **AS** APCS **ON** APCS.base=KVM.kvm_pit_state_id;
UC8	Present fine-grained page cache information per file for KVM related processes.	**SELECT** name, inode_name, file_offset , page_offset , inode_size_bytes , pages_in_cache, inode_size_pages, pages_in_cache_contig_start , pages_in_cache_contig_current_offset , pages_in_cache_tag_dirty , pages_in_cache_tag_writeback, pages_in_cache_tag_towrite , **FROM** Process_VT **AS** P **JOIN** EFile_VT **AS** F **ON** F.base=P.fs_fd_file_id **WHERE** pages_in_cache_tag_dirty **AND** name **LIKE** '%kvm%';
UC9	Present a view of socket files' state.	**SELECT** name, pid, gid, utime, stime, total_vm, nr_ptes, inode_name, inode_no, rem_ip, rem_port, local_ip , local_port , tx_queue, rx_queue **FROM** Process_VT **AS** P **JOIN** EVirtualMem_VT **AS** VM **ON** VM.base = P.vm_id **JOIN** EFile_VT **AS** F **ON** F.base = P.fs_fd_file_id **JOIN** ESocket_VT **AS** SKT **ON** SKT.base = F.socket_id **JOIN** ESock_VT **AS** SK **ON** SK.base = SKT.sock_id **WHERE** proto_name **LIKE** 'tcp';

The first query, UC1, finds child processes terminated due to a *kill* signal that did not originate from their parent process; child processes should follow their parent process's termination. The kernel's built-in tool infrastructure (`/proc`, *top, ps*) does not associate parent processes to children processes in order to reason about their state and identify this bug. Even if it did, the benefit of PiCO QL is that it provides a high-level language to express readable analysis tasks. Specifically, the query constraints mirror the bug's characteristics while the SELECT clause, especially column *state*, reflects the bug's outcome. With the use of relational views, as exemplified in Figure 3.10, query programming effort can shrink significantly.

The second query, UC2, accumulates thread I/O in the parent process, a view that has only recently been offered by the kernel's utilities under the `/proc` file system. The `/proc` interface[9] supplies such statistics but had failed to accumulate thread I/O in the parent process due to a design oversight. At the time of this writing accumulated thread I/O without taking into account the parent's I/O is not readily available from the `/proc` interface, and information is provided exclusively for each process. This example highlights PiCO QL's ability to combine bits of information in powerful ways in order to construct ad-hoc queries that reflect a user's objective. One could use scripting languages, such as AWK and Python for tailoring the output of kernel tools but these are limited to the data made available by the tools themselves. In addition, PiCO QL supports a single homogeneous interface to retrieve and analyze kernel data.

5.2.2.2 Security audits

PiCO QL can improve a system's security by expressing queries that spot suspicious behavior. For example, normal users, who do not belong to the `admin` or `sudo` groups, should rarely be able to execute processes with `root` privileges. The query in UC3 displays processes that do not obey this rule. This information is available through the kernel interface by reading `/proc/pid/status`, but to achieve the same objective post-processing is required to extract the desired credential values and express the constraints on them. An existing alternative for post-processing the provided information is to use the AWK programming language in conjunction with a shell script; see Figures A.11 and A.12 in Appendix for the respective scripts.

The PiCO QL query presented in UC3 is more concise and reflects the purpose of the analysis in a declarative manner compared to the AWK script alternative. Specifically, PiCO QL allows users to focus on their objective by presenting a relational data model of the kernel and providing a query language that avoids the use of low-level programming constructs, such as variables and assignments. Diagnostic tasks through PiCO QL hide the implementation details of their execution by leveraging the capabilities of a high-level declarative interface that is suitable for data management.

Another security requirement is verifying that processes have rightful access to the files they open. The query in UC4 reports processes that have a file open for reading without the necessary read access permissions. This query returns forty four records for our Linux kernel (Table 5.5). It can identify files whose access has leaked (unintentionally, e.g., through a race condition, or by design) to processes that have dropped elevated privileges.

The same check can be performed with SystemTap. Figure A.13 in Appendix presents the corresponding SystemTap script. SystemTap allows users to instrument any kernel function and examine the kernel's flow of execution based on user-defined filters. It can also capture the state of a function's internal variables and do user-specified aggregations. In addition, since embedded C is an option and a SystemTap script is compiled into a loadable kernel module, all of the kernel's public symbols are readily usable in scripts. SystemTap's probes span over the kernel and, as such, achieve instrumentation coverage.

SystemTap is particularly useful for kernel problems that require the analysis of execution flow.

[9] accumulated I/O: `/proc/pid/io` and each thread's I/O: `/proc/pid/task/tid/io`

In terms of access to data and computability, there is nothing that PiCO QL does that SystemTap cannot do; SystemTap provides the capability of writing embedded C in scripts. But when the objective requires kernel state examination rather than event-based analysis PiCO QL provides an advantageous alternative. Consider the query in UC4 compared to the SystemTap script in Figure A.13. A first observation is that the PiCO QL query is more compact because it does not use local variables, assignments, and print statements. It is also more readable and expressive owing to the query language's high-level syntax. A third observation is that to retrieve the desired information from the kernel through a SystemTap script either functions written in embedded C or composite instrumentation, that is, the activation of multiple kernel probes, are often required. Our SystemTap script uses embedded C functions.

The query and the script may seem computationally equivalent but they are not. The PiCO QL query traverses the accounting list of processes and checks a process's permissions against its open files while the SystemTap script instruments the *vfs_read()* function and reports unauthorized access to files by running processes for as long as the script executes. Thus, the SystemTap script only validates what is happening at the time it executes instead of the accounted state; the script did not show any unauthorized accesses within a time window of a few seconds.

Furthermore there is a quality difference between PiCO QL queries and SystemTap scripts. PiCO QL examines a consistent function of the system. The system is in a consistent state when a PiCO QL query starts to execute and it remains in that state until query evaluation completes. SystemTap, on the other hand, collects data as the system's state progresses and, thus, does not provide consistent results. This is further exemplified in Figure A.14. The script there records and filters the state of kernel data structures reflecting the PiCO QL query in UC4. The script is evaluated only once at the very beginning of the instrumentation, because it only activates the special probe *begin*, and exits immediately after its evaluation. The script does not provide a consistent view of the state because the system's state is evolving at the same time the script is executing; exclusive locking, which could provide consistency, only covers a fraction of the kernel's data. In root-cause analysis it is often useful to extract data while ensuring that the system state does not change in the interim.

Queries like the above could be used to detect the potential presence of a rootkit by retrieving details of processes with elevated privileges and unauthorized access to files. Advanced rootkits use dynamic kernel object manipulation attacks [PFWA06] to mutate non-control data, that is, kernel data structures. Baliga *et al.* [BGI08] present a number of such possible attacks, which tamper kernel data structures.

One attack involves adding a malicious binary format in the list of binary formats used by the kernel to load binary images of processes and shared libraries. A new format is added at the head of the list of binary formats so when a process is created the malicious handler is the first that is checked for its appropriateness to load the binary. The malicious code is executed and returns ENOEXEC error code. Then the kernel tries the next handler in the list. In UC5 PiCO QL queries the list of formats and crosschecks each with the format used by each process in the system. A malicious handler that implements this attack would not match any processes.

Hardware virtualization environments suffer from vulnerabilities as well [PBSL13]. CVE-2009-3290 [Nat09] describes how guest VMs, operating at Ring 3 level, abuse hyper calls, normally issued by Ring 0 processes, to cause denial of service or to control a KVM host. The SQL query in UC6 retrieves the current privilege level of each online virtual CPU and its eligibility to execute hypercalls, and displays VMs violating hypercalls. This query can be used to detect the possibility of the corresponding attacks.

Perez-Botero *et al.* [PBSL13] report vulnerabilities in the KVM hypervisor due to lack of data structure state validation (CVE-2010-0309 [Nat10]). Lack of data validation in the programmable interval timer (PIT) data structures allowed a malicious VM to drive a full host operating system to crash by attempting to read from `/dev/port`, which should only be used for writing. The PIT channel state array

Table 5.5: Present SQL query execution cost for 10 diverse queries.

PICO QL query	Query label	LOC	Records returned	Total set size (records)	Execution space (KB)	Execution time (ms)	Record evaluation time (μs)
Figure 3.15	Relational join	10	80	683929	1667.10	231.90	0.34
UC6	Join – virtual table context switch ($\times 2$)	3(9)	1	827	33.27	1.60	1.94
UC7	Join – virtual table context switch ($\times 3$)	4(10)	1	827	32.61	1.66	2.01
UC3	Nested subquery (FROM, WHERE)	13	0	132	27.37	0.25	1.89
UC4	Nested subquery (WHERE), OR evaluation, bitwise logical operations, DISTINCT records	13	44	827	3445.89	10.69	12.93
UC8	Page cache access, string constraint evaluation	6	16	827	26.33	0.57	0.69
UC9	Arithmetic operations, string constraint evaluation	11	0	827	76.11	0.59	0.71
SELECT 1;	Query overhead	1	1	1	18.65	0.05	50.00

mirrors the permitted access modes as array indexes; read access is masked to an index that falls out of bounds and triggers the crash when later dereferenced. The query in UC7 provides a view of the PIT channel state array where each tuple in the result set mirrors a permitted access mode; read access is not included. Accessing this information in the form of a simple SQL query can help with automatic validation of data structure state during testing and prevent vulnerabilities of this kind.

5.2.2.3 Performance

Regulating system resources is a requirement for system stability. PICO QL can provide a custom view of a system's resources and help discover conditions that hinder its performance. The query in UC8 extracts a view of the page cache detailing per file information for KVM related processes. It shows how effectively virtual machine I/O requests are serviced by the host page cache assuming that the guest operating system is executing direct I/O calls.

One of PICO QL's advantages is its extensible ability to offer a unified view of information across the kernel. The combined use of diagnostic tools can point to the solution in most situations, but for some of those situations, PICO QL provides the answer without recourse to external tools. For instance, consider UC9, which shows how to combine data structures to extract detailed performance views of and across important kernel subsystems, namely process, CPU, virtual memory, file, and network. Adding to this list means only expanding the representation of data structures. Operating system utilities, such as *netstat*[10] or *lsof*,[11] can also combine data originating from different kernel subsystems, but their capabilities are not extensible and manipulating the retrieved data in non-trivial ways requires post-processing.

5.2.3 Presentation of measurements

PICO QL's quantitative evaluation presents the execution efficiency of SQL queries in the Linux kernel (Section 5.2.3.1) and the performance impact of queries on system operation by conducting overhead measurements on an array of macro-benchmarks (Section 5.2.3.2).

[10] *netstat -p -e*
[11] *lsof -iTCP*

5.2.3.1 Query execution efficiency

We compute the query execution time as the difference with cycle-accurate precision between two timestamps, acquired at the end and at the start of each query respectively. Tests are carried out in an otherwise idle machine (1GB RAM, 500GB disk, 2 cores) running the Linux kernel (v3.6.10). The mean of at least three runs is reported for each query.

The rows in Table 5.5 contain queries with diverse characteristics and columns contain query properties and measurement metrics. Specifically, columns include references to queries presented in the paper, a label characterizing each query's execution plan, the lines of SQL code for expressing each query, the number of records returned, the total set size evaluated, the execution space and time, and the average time spent for each record.

Table 5.5 indicates the programming effort for writing some representative queries in terms of LOC. As there is no standard way to count SQL lines of code, we count logical lines of code, that is, each line that begins with an SQL keyword excluding AS, which can be omitted, and the various WHERE clause binary comparison operators. At a minimum, SQL requires two lines of code to retrieve some information (SELECT...FROM...;). The PiCO QL evaluation queries listed in Table 5.5 require between six and thirteen LOC. This is not a small figure, but it is a price we pay for sophisticated queries, such as most of those used in our evaluation. Moreover, a large part of this programming cost can be abstracted away because queries can be composed from other queries leveraging standard relational views (Figure 3.10). This is the case with UC6 and UC7 whose LOC drop to less than half of the original. Finally, PiCO QL provides an advantage of qualitative nature: it allows users to focus on the information retrieval aspects of an analysis task. This can not be measured by LOC.

5.2.3.2 Impact on system performance

We study the performance impact of PiCO QL queries firing at fixed frequency on system operation. For this purpose we use the Phoronix test suite [The14b], which contains macro-benchmark tests targeting specific kernel subsystems such as processor, memory, and disk. We measure the performance overhead in time compared to native execution, with and without PiCO QL. The evaluation is conducted on version 3.14.4 of the Linux kernel. To provide a better understanding of the performance overhead we vary the frequency of firing queries. We use a particular query instance in these tests, UC9. This is small sized taking approximately 0.6 milliseconds to run. Then we select the query firing frequency based on this query. Table 5.6 presents our measurements.

The first table column lists the macro-benchmark test that we execute, the second one provides which kernel subsystem the test stresses, the third shows the query frequency set for this test, the fourth presents the test's average execution time, the fifth shows the bandwidth, the sixth states the standard deviation between subsequent runs of this test reported as a percentage over the mean, and the seventh presents the overhead of the queries on the system's performance, as additional time over native execution. The reported measurements denote the average of at least three runs unless otherwise noted at the standard deviation column.

Query execution frequency can increase up to a limit. Although the overhead is high, firing queries at high frequency still leaves the system in a usable state and can be justified in one-off debugging situations. Obviously, the frequency can not be higher than the duration of the queries being fired and the cost of setting up the isolated execution environment is an important cost attribute of the query roundtrip especially in SMP systems. The preparation cost rises with increasing number of online CPUs since it takes longer to wait for all of them to complete non-interruptible code execution.

In Figure 5.3 we measure the cost of the *stop_machine()* utility for three reasons: a) to provide a complete picture of a PiCO QL query roundtrip, which is the cost of running *stop_machine()* plus the cost of the actual query execution, b) to inform how the cost of running *stop_machine()* evolves in systems with varying number of CPUs, and c) to show the level of query execution frequency we can

Table 5.6: Present SQL query overhead on system performance.

Test name	Kernel subsystem	Query frequency (q/sec)	Average execution time (sec)	Bandwidth (MB/sec)	Standard deviation (%)	Overhead (%)
MrBayes (analysis)	processor	0	60.91		0.11	0
MrBayes (analysis)	processor	2.5	64.30		0.09	5.56
MrBayes (analysis)	processor	5	65.31		0.06	7.22
MrBayes (analysis)	processor	10	69.40		0.26	13.94
MrBayes (analysis)	processor	100	103.48		0.52	69.89
Apache (compile)	processor	0	129.62		0.22	0
Apache (compile)	processor	2.5	134.00		0.07	3.38
Apache (compile)	processor	5	136.44		0.02	5.26
Apache (compile)	processor	10	144.94		1.2	11.82
apache (compile)	processor	100	201.40		0.06	55.37
LZMA (compress)	processor	0	286.73		0.3	0
LZMA (compress)	processor	2.5	293.47		0.14	2.35
LZMA (compress)	processor	5	298.81		0.33	4.21
LZMA (compress)	processor	10	309		0.09	7.76
LZMA (compress)	processor	100	404.94		0.2	41.22
SmallPT	processor	0	635		0.08	0
SmallPT	processor	2.5	660		0.09	3.94
SmallPT	processor	5	670		0.00	5.51
SmallPT	processor	10	704		0.08	10.86
SmallPT	processor	100	972		0.1	53.07
Open FMM Nero2D	processor	0	1539.81		1 run	0
Open FMM Nero2D	processor	2.5	1575.88		1 run	2.34
Open FMM Nero2D	processor	5	1591.97		1 run	3.39
Open FMM Nero2D	processor	10	1661.32		1 run	7.89
Open FMM Nero2D	processor	100	2104.92		1 run	36.70
Stream	memory	0	5.7	2608.80	0.05	0
Stream	memory	2.5	5.8	2606.68	0.09	0.08
Stream	memory	5	5.9	2606.12	0.11	0.1
Stream	memory	10	6	2578.50	1.25	1.16
Stream	memory	100	7.2	2007.90	7.79	23.03
RAMspeed (integer)	memory	0	817	2143.51	0	0
RAMspeed (integer)	memory	2.5	849	2058.56	0	3.96
RAMspeed (integer)	memory	5	888	2016.12	1 run	5.94
RAMspeed (integer)	memory	10	897	1949.13	1 run	9.07
RAMspeed (integer)	memory	100	1116	1580.36	1 run	26.27
RAMspeed (float)	memory	0	734	2342.44	0	0
RAMspeed (float)	memory	2.5	763	2253.30	0	3.80
RAMspeed (float)	memory	5	777	2222.88	1 run	5.10
RAMspeed (float)	memory	10	811	2133.76	1 run	8.90
RAMspeed (float)	memory	100	1011	1729.57	1 run	26.16
Linux-ker (unpack)	disk	0	23.68		3.17	0
Linux-ker (unpack)	disk	2.5	24.24		1.23	2.36
Linux-ker (unpack)	disk	5	25.16		1.92	6.25
Linux-ker (unpack)	disk	10	25.63		1.31	8.23
Linux-ker (unpack)	disk	100	31.94		0.66	34.88
Geometric mean		6.59	198.07	2166.68	0.39	4.63
Average		23.5	525.37	2189.17	0.48	11.49

achieve with respect to the number of CPUs. We take over measurements in a single idle system with 8 dual core CPUs by hot-plugging CPUs.[12]

5.2.4 Results

We present the outcomes of the measurements regarding query execution efficiency (Section 5.2.4.1) and impact on system performance (Section 5.2.4.2).

5.2.4.1 Query execution efficiency

Query measurements allow a number of observations. First, query evaluation appears to scale well as the total set size increases. The average amount of time spent for each record in the relational join query (Listing 3.15) is the smallest, beating even the average record evaluation time of the query performing arithmetic operations (UC9). The cartesian product evaluated for the former approximates 700,000 records. Second, nested subqueries (UC3) perform well as opposed to DISTINCT evaluation (UC4), which seems to be the source of large average execution time per record. Third, multiple page cache accesses (UC8) during a query evaluation are affordable, incurring the second best record evaluation time and topping even arithmetic operations (UC9), which, as expected, are very efficient.

According to the memory space measurements during query execution, PICO QL has modest memory space requirements for most query plans. We distinguish two queries due to their significantly

[12] `echo 0 > /sys/devices/system/cpu/cpuX/online`

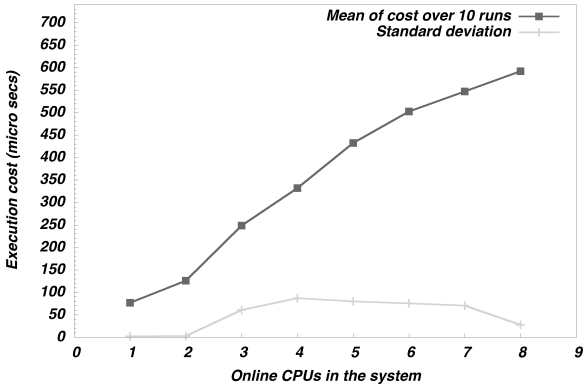


Figure 5.3: Fixed query cost depending on number of CPUs.

larger memory space footprint. The relational join query (Listing 3.15) requires 1.5MB but this can be justified by the evaluated total set size, which approximates 700K records. The second query with large memory space requirements, almost 3.5MB, is the one involving DISTINCT evaluation (UC4). The sizeable footprint is explained by the implementation of the DISTINCT algorithm to remove duplicates from the original result set of 827 records.

5.2.4.2 Impact on system performance

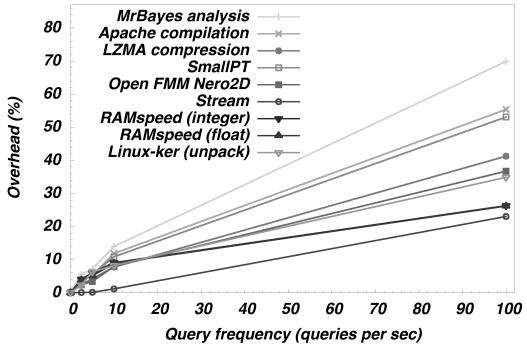
The measurements reveal that there is a piece-wise linear relationship between the tested query frequencies and the resulting overhead in high query frequency as Figures 5.4(a) and 5.4(c) show. In addition, PICO QL achieves lower overall overhead in computationally tough tests (Figures 5.4(d) and 5.4(c)). Specifically, the second toughest test incurs the third lowest overhead, the third toughest test ranks second in terms of low overhead, and the toughest test incurs the fourth lowest overhead.

The overhead of continuous PICO QL queries on the system is low up to a frequency of 5 queries per second (q/sec) allowing the regular background execution of queries. We see that above this rate the execution isolation feature of PICO QL kicks in doubling this overhead at the frequency of 10 q/sec (Figure 5.4(b)). Still, the overhead is modest at this point with a value below 10% for 6 out of 9 tests and a maximum value of 14%.

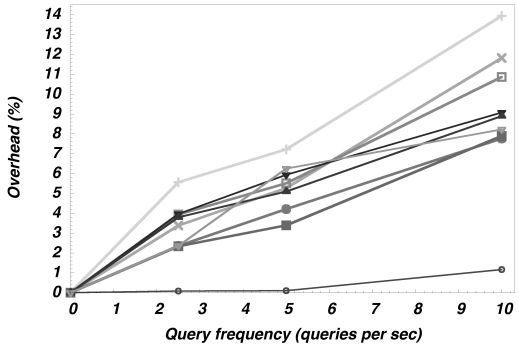
Tests that stress the processor are particularly important due to the nature of our approach. If we examine the overhead measured at the same query frequency between different processor tests, we observe that the overhead tends to decrease with longer computation periods. Examining the slope of the lines in Figure 5.4(a) we observe a rate of decrease in the segment between query frequency levels 2.5 and 5 q/sec compared to the initial one, then a rate of increase similar to the initial one up to 10 q/sec, then a rate of decrease up to 100 q/sec that is larger than the decrease rate in the segment between 2.5 and 5 q/sec.

Overall, the highest frequency level of 100 q/sec imposes a strong demand for computation on the system leading to high overhead. For example, this setting provides the capacity to detect changes to the system's state by querying it at high frequency.

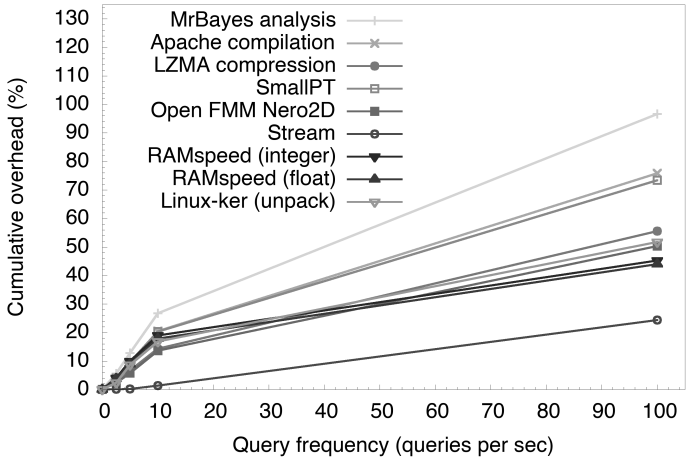
Regarding the cost of the `stop_machine` utility, we find that it rises approximately linearly as we add CPUs in the system (see Figure 5.3).



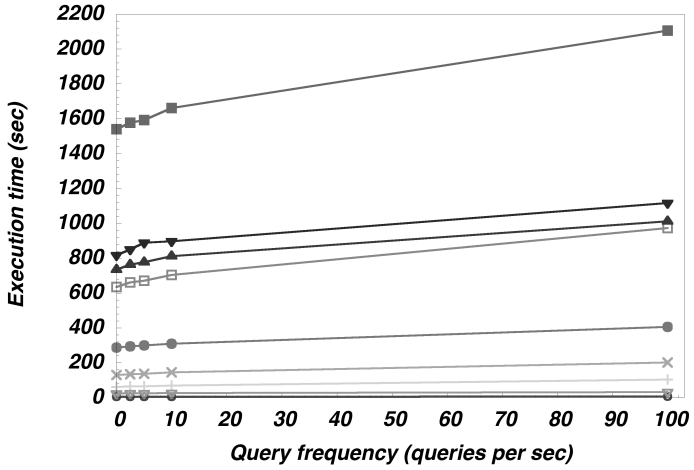
(a) Overhead - frequency plot



(b) Overhead - frequency plot zooming in 0-10 q/sec range for plot in Figure 5.4(a)



(c) Relationship between query frequency and cumulative performance overhead for processor tests



(d) Relationship between query frequency and execution time for processor tests

Figure 5.4: Impact on system performance

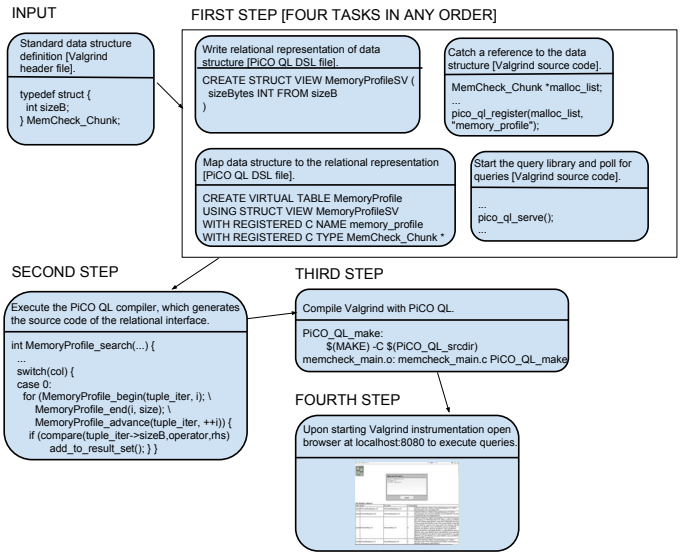


Figure 5.5: Workflow for embedding PiCO QL to Valgrind

5.3 Analysis of memory profiles in the Valgrind instrumentation framework

Figure 5.5 shows the setup workflow followed in order to use PiCO QL with Valgrind. We present the method (Section 5.3.1), the use cases (Section 5.3.2), the measurements (Section 5.3.3), and the outcomes (Section 5.3.4).

5.3.1 Method

We evaluate PiCO QL using three axes of analysis: use cases, performance measurements, and user experiment. Use cases (Section 5.3.2) show how PiCO QL can aid software testing and analysis by extracting valuable information from three Valgrind tools, Memcheck, Cachegrind, and Callgrind, through a high-level relational interface. Quantitative evaluation presents the execution time for a variety of queries with diverse computation requirements (Section 5.3.3). Finally, we conduct a user experiment to record users' evaluation of our interface compared to a popular alternative, Python scripting. We present the user study in a separate section (Section 5.4).

5.3.2 Use cases

The use cases show queries that leverage memory check metadata collected by Memcheck, cache analysis metadata gathered by Cachegrind, and function call history metadata accumulated by Callgrind. Two of the use cases show that our approach can be used to answer performance questions tackled in a well-received text in the field [Gre13].

For each tool we present first the PiCO QL representations we produced. Table 5.7 depicts the rules used to map data structure associations to each virtual table schema. Data structure names in the table map intuitively to virtual table names. Note that in our work with Valgrind we did not come across any *many-to-many* associations; had we found any, our approach would accommodate them.

Table 5.7: Data structure association mapping according to our relational representation's rules

From	To	Rule
Memcheck		
memory_profile	execution_context	2b
Cachegrind		
cache_profile	cache_metrics	2b
cache_profile	code_location	2b
cache_profile	branch_metrics	2b
Callgrind		
application_thread	jump_call_cost_center	3
application_thread	basic_block_cost_center	3
jump_call_cost_center	basic_block_cost_center	2a
jump_call_cost_center	cost_id	2a
basic_block_cost_center	basic_block_cost_center (recursion)	3
basic_block_cost_center	basic_block	2a
basic_block_cost_center	cost_id	2a
basic_block	function_node	2a
function_node	function_context	2b

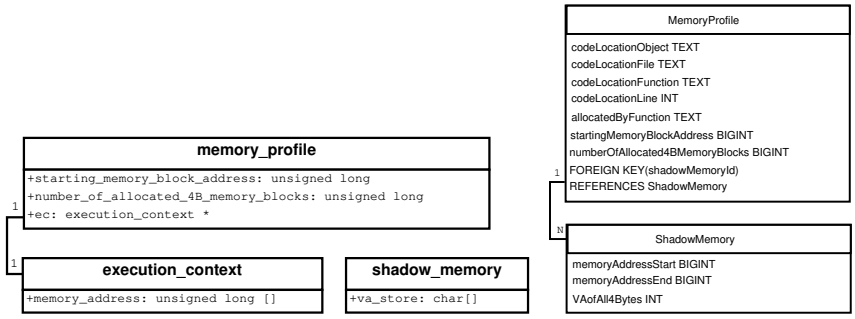
Rule No	**Rule description** (see Section 3.1.1)
1	scalar attributes of a class or struct become columns in the virtual table that represents the struct or class
2	*has-one* associations, which include nested data structures and references to nested data structures, can be represented in two ways, a) as columns in a separate virtual table that stands for the contained data structure or b) as columns within the parent data structure's virtual table
3	*has-many* associations, which include nested groups of data structures, such as an array or a list, and references to nested groups of data structures, are represented by an associated table that corresponds to the set of contained data structures.

5.3.2.1 Memcheck

Memcheck [SN05] is a memory checker that instruments an application's memory operations and prints memory errors during its execution. It also provides a synopsis of its findings at program termination.

Figure 5.6 shows Memcheck's data structure model and relational representation. At the time of application execution, Memcheck records memory blocks allocated as per application requests. Each allocation is requested from an execution context that corresponds to a specific instruction memory address. We use this information to extract the location of the instruction by calling Valgrind's public instrumentation interface that provides these metadata. This data extraction process is completely transparent to users who may ask for the object, source file, function, and line that an allocation happens as any other data. Because each memory block *has-one* execution context and there is no point in making this relationship explicit we apply rule 2b (see Table 5.7).

Memcheck's memory integrity checks rely on shadow memory, that is, on data structures of type *shadow_memory* that record the validity and addressability (VA) state of every byte of memory used by an application. Memcheck stores the VA information for each byte used by an application in a pair of bits. Consequently, each slot, that is 8 bits, in the *va_store* character array holds the VA state of 4 bytes of application memory. Although the data structure that stores the memory profile is not associated with the shadow memory data structure, the memory address range of each block in the memory profile can be used to find the right index in the shadow memory where each word's VA metrics are stored. We capitalise on this capability to design a relational representation of shadow memory that exposes VA metrics for each memory word at byte-level when it is associated to an application memory block.



(a) Memcheck model (b) Virtual table schema

Figure 5.6: Memcheck's data structure model and relational representation

Listing 5.19: Locate partialy defined bytes.

```
SELECT codeLocationFunction, codeLocationLine, startingMemoryBlockAddress,
  codeLocationFile , codeLocationObject, allocatedByFunction ,
  COUNT(*) AS PDBsPerMemoryAllocation
FROM MemoryProfile
JOIN ShadowMemoryView ON base=shadowMemoryId
WHERE VAof1stByte = ‘‘ partdefined ’ ’
OR VAof2ndByte = ‘‘ partdefined ’ ’
OR VAof3rdByte = ‘‘ partdefined ’ ’
OR VAof4thByte = ‘‘ partdefined ’ ’
GROUP BY startingMemoryBlockAddress
ORDER BY PDBsPerMemoryAllocation DESC;
```

Memcheck stores detailed metadata, such as code location, that enable it to carry out comprehensive integrity checks and discover errors. However, important information about the instrumented application's memory profile, hidden in the metadata, goes to waste. For instance, a use case concerns certain pieces of application code that are generating a large number of partially defined bytes (PDB). These are individual bytes in memory, which are not fully defined, and therefore not usable. PDBs may be the source of errors, which Valgrind spots, but in cases where PDBs are legitimate, they remain unnoticed. In large numbers PDBs may cause non-trivial waste of memory space and performance degradation. PiCO QL pinpoints the root of PDBs by mapping PDBs to source code locations through an appropriate SQL query shown in Listing 5.19.

Listing 5.19 references two virtual tables, MemoryProfile and ShadowMemory. MemoryProfile represents an array that keeps record of all memory allocations requested by a running application instrumented by Valgrind, along with related metadata. The query in Listing 5.19 joins each memory allocation (MemoryProfile) requested by the instrumented application with the shadow memory (ShadowMemory), looks for partly defined bytes recorded for an allocation in the shadow memory, counts them per memory allocation, and orders the tuples by descending number of partly defined bytes. The result set includes metadata corresponding to each allocation, namely the function, line number, and file name it took place, the start memory address of the allocated block, and the function it was allocated by the block.

ShadowMemory is not a mere virtual table, but a relational view, which is a popular feature of SQL implementations. Relational views are important because they can store recurring queries thereby relieving users from the strain of drafting them each time they want to use them. The relational views, which are non-materialized, can be defined in the DSL using the standard CREATE VIEW notation, as shown in Listing 5.20. The ShadowMemory view defines an alias for queries that retrieve shadow

Listing 5.20: Relational view definition stores query identifying shadow memory VA bits for 4 Byte words

```
CREATE VIEW ShadowMemoryView AS
 SELECT base, memoryAddressStart
  (SELECT CASE
   WHEN VAofAll4Bytes = 170 THEN ''defined''
   ELSE ''some_undefined''
   END) VAofAll4Bytes,
  (SELECT CASE
   WHEN VAofAll4Bytes & 3 = 0 THEN ''noaccess''
   WHEN VAofAll4Bytes & 3 = 1 THEN ''undefined''
   WHEN VAofAll4Bytes & 3 = 2 THEN ''defined''
   WHEN VAofAll4Bytes & 3 = 3 THEN ''partdefined''
   END) VAof1stByte
  (SELECT CASE
   WHEN (VAofAll4Bytes >> 2) & 3 = 0 THEN ''noaccess''
   WHEN (VAofAll4Bytes >> 2) & 3 = 1 THEN ''undefined''
   WHEN (VAofAll4Bytes >> 2) & 3 = 2 THEN ''defined''
   WHEN (VAofAll4Bytes >> 2) & 3 = 3 THEN ''partdefined''
   END) VAof2ndByte
  (SELECT CASE
   WHEN (VAofAll4Bytes >> 4) & 3 = 0 THEN ''noaccess''
   WHEN (VAofAll4Bytes >> 4) & 3 = 1 THEN ''undefined''
   WHEN (VAofAll4Bytes >> 4) & 3 = 2 THEN ''defined''
   WHEN (VAofAll4Bytes >> 4) & 3 = 3 THEN ''partdefined''
   END) VAof3rdByte
  (SELECT CASE
   WHEN (VAofAll4Bytes >> 6) & 3 = 0 THEN ''noaccess''
   WHEN (VAofAll4Bytes >> 6) & 3 = 1 THEN ''undefined''
   WHEN (VAofAll4Bytes >> 6) & 3 = 2 THEN ''defined''
   WHEN (VAofAll4Bytes >> 6) & 3 = 3 THEN ''partdefined''
   END) VAof4thByte
 FROM ShadowMemory;
```

memory validity and accessibility bits (VAbits).

The case statements in the view check the recorded state of a VAbits 8-bit set, which records the state of 4 memory bytes allocated to the instrumented application. The first case checks whether some of the 4 pairs of bits are undefined by testing whether the bitset is `10101010`, which amounts to the integer value 170. This form denotes four pairs of bits with the value 2, which encodes that a shadowed byte is fully defined.

Memcheck's analysis is crucial for debugging an application's memory errors but error analysis can be difficult if errors are numerous. Memcheck records twelve different kinds of errors modelled as a `union` structure since an error is of a specific kind. For each kind, the tool stores particular data related to this kind like leak size for a leak error. PiCO QL defines a relational representation for each error kind so that error kind information can be retrieved in a query through a join operation to the relational representation (Listing 5.21). Since an error can be of any kind, joining to the relational representation of each error kind is required. This is computationally cheap since only one join will be processed and this join bears the cost of a pointer traversal; however, from a programming point of view it is expensive. Thankfully, we can define a view for this query in the DSL description and hide this cost too. Finally, we can add semantics to extracted data to aid readability as is the case with an error kind.

A relational view of this sort provides us with the opportunity to analyse errors through SQL queries to pinpoint critical nodes in program memory errors. Listing 5.22 retrieves errors grouped by the functions they appear in and their kind and ordered by their accumulated appearances.

The size of allocated memory blocks during a program's run is important in tracking erroneous

Listing 5.21: Expose full information for error recorded and tag semantics to extracted data

```
SELECT
  (SELECT CASE WHEN kind = 0 THEN 'Err_Value'
   WHEN kind = 1 THEN 'Err_Cond'
   WHEN kind = 2 THEN 'Err_CoreMem'
   WHEN kind = 3 THEN 'Err_Addr'
   WHEN kind = 4 THEN 'Err_Jump'
   WHEN kind = 5 THEN 'Err_RegParam'
   WHEN kind = 6 THEN 'Err_MemParam'
   WHEN kind = 7 THEN 'Err_User'
   WHEN kind = 8 THEN 'Err_Free'
   WHEN kind = 9 THEN 'Err_FreeMismatch'
   WHEN kind = 10 THEN 'Err_Overlap'
   WHEN kind = 11 THEN 'Err_Leak'
   WHEN kind = 12 THEN 'Err_IllegalMempool'
   END) mcErrorTag, *
  FROM ErrorVT
  JOIN ErrorValueVT EV
  ON EV.base = value_id
  JOIN ErrorCondVT EC
  ON EC.base = cond_id
  JOIN ErrorCoreMemVT ECM
  ON ECM.base = coremem_id
  JOIN ErrorAddrVT EA
  ON EA.base = addr_id
  JOIN ErrorJumpVT EJ
  ON EJ.base = jump_id
  JOIN ErrorRegParamVT ERP
  ON ERP.base = regparam_id
  JOIN ErrorMemParamVT EMP
  ON EMP.base = memparam_id
  JOIN ErrorUserVT EU
  ON EU.base = user_id
  JOIN ErrorFreeVT EF
  ON EF.base = free_id
  JOIN ErrorFreeMismatchVT EFM
  ON EFM.base = freemismatch_id
  JOIN ErrorOverlapVT EO
  ON EO.base = overlap_id
  JOIN ErrorLeakVT EL
  ON EL.base = leak_id
  JOIN ErrorIllegalMempoolVT EIM
  ON EIM.base = illegalmempool_id;
```

patterns in the program's memory allocation operations. For instance, Listing 5.23 rounds memory allocation sizes at the megabyte (MB) level using integer division, then counts memory allocations per MB, and finally orders them by decreasing count frequency. Memcheck does not output detailed memory block allocation information. Gregg [Gre13, p.247] lists an identical task with DTrace to summarise the requested size of memory allocations for a specific process presented as a power-of-two frequency distribution format.

Moving on, Listing 5.24 reports the number of bytes in allocated memory blocks that are undefined and the accumulated bytes wasted for each memory block. These provide a measure to alignment holes caused by bad organisation of data structures and fragmentation introduced by the dynamic memory allocator in the course of a program execution.

A related analysis task involves retrieving stack trace metadata, such as function name and line number, for memory block allocations. Listing 5.25 presents such a query, which orders stack traces resulting to memory allocations that exceed 1MB of memory by decreasing size. Gregg [Gre13, p.248] describes a similar task with DTrace to summarise the requested size of memory allocations decorated with stack trace metadata.

Another potential task examines aggregated memory allocations in the context of specific functions or function families. Listing 5.26 retrieves the total size of memory allocated for each function

Listing 5.22: Group errors by function they appear in and sum their appearances and leak size

```
SELECT fn_name, addr, line_no, file_name,
 obj_name, thread_id, mcErrorTag,
 SUM(count) AS appear, SUM(leak_size)
FROM ErrorFullV
JOIN IPVT
ON base=execontext_id
GROUP BY fn_name, mcErrorTag
ORDER BY appear;
```

Listing 5.23: Retrieve number of memory allocations per MB ordered by decreasing size of memory allocation.

```
SELECT sizeBytes / 1000000
  AS memoryAllocationSizeInMB, COUNT(*) AS numberOfAllocationsAtMBSize
FROM MemoryProfile
GROUP BY memoryAllocationSizeInMB
ORDER BY memoryAllocationSizeInMB;
```

Listing 5.24: Count undefined bytes per memory allocation.

```
SELECT startingMemoryBlockAddress, sizeBytes, codeLocationFunction,
          codeLocationLine, SUM(bytesWasted)
FROM (
          SELECT startingMemoryBlockAddress, sizeBytes, codeLocationFunction,
                    codeLocationLine,
                    CAST( validityBitTag = 'undefined' AS INTEGER) AS bytesWasted
          FROM MemoryProfile
          JOIN ShadowMemoryView
          ON base=shadowMemoryId
          WHERE validityBitTag = 'undefined'
) BW
GROUP BY startingMemoryBlockAddress
ORDER BY SUM(bytesWasted) DESC;
```

whose name matches *BZ2_bzCompressInit* only. This is a function of `bzip2`, which takes care of memory allocations. We will meet it again in the evaluation outcomes.

5.3.2.2 Cachegrind

Cachegrind is a performance analysis tool for recording a program's interaction with the computer's cache hierarchy. Cachegrind provides cache performance metrics associated to the code that triggers a cache access. Figure 5.7 depicts its data structure model and virtual table schema. Branch metrics count conditional and indirect branching occurrences and mispredictions. Since all associations in the data structure model are intuitive *has-one* instances we opt for using rule 2(b) of Section 3.1.1 to avoid the expressional and computational cost of joins.

Cachegrind's bookkeeping allows PiCO QL to extract useful patterns regarding cache utilisation through its SQL interface. In addition, analysis can happen interactively at runtime avoiding the need to dump and read intermediate metadata.

A common aspect of studying cache utilisation is the ratio between cache accesses and cache misses, especially for code locations associated with a large number of cache accesses. Listing 5.27 retrieves source code lines tagged with more than 1M instruction cache read accesses ordered by descending level 1 instruction cache read misses and level 2 instruction cache read misses.

Another motivating example regards exploring the overall cache utilisation related to source code

Listing 5.25: Retrieve total size of allocated memory blocks per source code line whose size exceeds a MB ordered by decreasing total size.

```
SELECT codeLocationFunction, codeLocationLine, startingMemoryBlockAddress,
  codeLocationFile , codeLocationObject,
  SUM(sizeBytes) AS totalSizeBytesPerLOC
FROM MemProfile
GROUP BY codeLocationFile, codeLocationFunction, codeLocationLine
HAVING totalSizePerLOC > 1000000
ORDER BY totalSizePerLOC DESC;
```

Listing 5.26: Inspect the total size of memory allocations in Bytes for *BZ2_bzCompressInit*.

```
SELECT codeLocationFunction, codeLocationLine, startingMemoryBlockAddress,
          codeLocationFile , codeLocationObject, allocatedByFunction , sizeBytes
FROM MemoryProfile
WHERE codeLocationFunction LIKE 'BZ2_bzCompressInit'
ORDER BY sizeBytes DESC;
```

lines of specific function families to discover potential sources of bottlenecks. Listing 5.28 returns total cache metrics for distinct source code locations that belong to a function whose name matches *lookup* ordered by descending data cache read misses.

At a higher level, we might be interested to aggregate cache utilisation for distinct functions in order to discover bottlenecks owed to the implementation of specific functions or code blocks within them. Listing 5.29 returns total cache metrics for each source code function ordered by level 1 data cache write misses.

5.3.2.3 Callgrind

Callgrind is a function call history recording tool. Callgrind attributes costs to a program's basic blocks and to jump calls between those blocks. In addition, Callgrind records relationships between code basic blocks, which belong to specific functions. A basic block is a block of source code with at most one call instruction, e.g., function call. Since basic blocks can be executed in multiple contexts, they have multiple basic block cost centres associated to them. A separate layer of cost centres is used to store recursion costs. Callgrind can record the above information for each application thread separately. Although complex, this model fits well in a relational interface and provides a good context for demonstrating the advantages of a relational representation. Figure 5.8 shows a part of Callgrind's data structure model and its relational representation.

A common task in dynamic call graph analysis regards spotting methods that are never called in the course of a program's execution. Listing 5.30 inspects the execution counter of each basic block's execution contexts to discover basic blocks that are never executed.

To debug performance bottlenecks it often helps to observe execution counters related to a basic block execution and the accumulated cost of the associated execution context. Listing 5.31 selects the basic blocks and associated metadata that contribute the most instruction execution cost. Callgrind takes specific measures to handle recursion, that is, it models recursion for each basic block cost at a particular context as an array of basic block costs nested to the first basic block cost. These map to the recursion levels of each function. Thus PiCO QL uses an additional join operation (see *BasicBlockCostCenterRecursion*) to include recursion costs in query computation.

Sometimes jump calls, such as method calls between basic blocks account for a significant part of execution cost, for example due to repeated invocations of a method. Listing 5.32 groups recorded jump calls per basic block that initiates the call, aggregates them by the number of instructions fetched

Listing 5.27: Catch source code lines with over 1M cache instruction read accesses.

```
SELECT codeLocationFile , codeLocationFunction , codeLocationLine ,
  cacheInstructionReadAccesses , cacheInstructionReadMissL1 ,
  cacheInstructionReadMissL2 , branchConditionalTotal ,
  branchConditionalMispredicted , branchIndirectTotal ,
  branchIndirectMispredicted
FROM Cachegrind
WHERE cacheInstructionReadAccesses > 1000000
ORDER BY cacheInstructionReadMissL1 DESC,
  cacheInstructionReadMissL2 DESC;
```

Listing 5.28: Retrieve total cache metrics for source code locations of functions whose name matches 'lookup' ordered by descending data cache read misses.

```
SELECT codeLocationFile , codeLocationFunc, codeLocationLine ,
  SUM(cacheInstructionReadAccesses), SUM(cacheInstructionReadMissL1),
  SUM(cacheInstructionReadMissL2), SUM(cacheDataReadAccesses),
  SUM(cacheDataReadMissL1), SUM(cacheDataReadMissL2),
  SUM(cacheDataWriteAccesses), SUM(cacheDataWriteMissL1),
  SUM(cacheDataWriteMissL2)
FROM Cachegrind
WHERE codeLocationFunction LIKE '%lookup%'
GROUP BY codeLocationFile, codeLocationFunction, codeLocationLine
ORDER BY SUM(cacheDataReadMissL1) DESC;
```

Listing 5.29: Attribute total cache metrics to each function.

```
SELECT codeLocationFile , codeLocationFunction , codeLocationLine ,
  SUM(cacheInstructionReadAccesses), SUM(cacheInstructionReadMissL1),
  SUM(cacheInstructionReadMissL2), SUM(cacheDataReadAccesses),
  SUM(cacheDataReadMissL1), SUM(cacheDataReadMissL2),
  SUM(cacheDataWriteAccesses), SUM(cacheDataWriteMissL1),
  SUM(cacheDataWriteMissL2), SUM(branchConditionalTotal),
  SUM(branchConditionalMispredicted), SUM( branchIndirectTotal ),
  SUM(branchIndirectMispredicted)
FROM Cachegrind
GROUP BY codeLocationFile, codeLocationFunction
ORDER BY SUM(cacheDataWriteMissL1) DESC;
```

and data read accesses, and orders them by total number of times called. The query returns the top 20 basic blocs by using a LIMIT clause.

5.3.3 Presentation of measurements

We evaluated PiCO QL on three tools, namely `gzip`, `bzip2`, and `egrep`. The operations that the tools performed during Valgrind's recording were to compress a large archive of size 230MB and search for a word in the archive. The evaluation took place on a Mac OS X 10.6.8 with 2GB RAM and 2.6GHz Intel Core Duo processor. We measured query execution time in terms of CPU time provided by `ps`. Specifically, for each query we invoked `ps` at query start time and finish time and calculated the difference. Each time measurement provided in Table 5.8 stands for the mean of three runs. We considered three runs sufficient because the measurements showed small variance.

We select those tools and operations because they provide an insight on how PiCO QL handles varying input size. Total set size, that is, the total number of rows evaluated, in the cases of Listings 5.19 and 5.24 accounts for 4 byte memory words that were in use by the instrumented application at

Listing 5.30: Select basic block metadata for blocks that are never executed.

```
SELECT DISTINCT BB.memoryAddress, FN.codeLocationFunction,
  BB.codeLocationLine , FN. codeLocationFile , BB.codeLocationObject ,
  BB. codeLocationOffset , BB. instructionCount
FROM ApplicationThread
JOIN BasicBlockCostCenter BC ON BC.base=T. basicBlockCostCenterId
JOIN BasicBlock BB ON BB.base = BC. basicBlockId
JOIN FunctionNode FN ON FN.base = BB.functionNodeId
WHERE NOT BC.executionCounterSum
ORDER BY BB.codeLocationObject, FN. codeLocationFile , FN.codeLocationFunction;
```

Listing 5.31: Retrieve the most computationally expensive basic blocks ordered by descending execution cost.

```
SELECT BB.memoryAddress, FN.codeLocationFunction, BB.codeLocationLine ,
   FN. codeLocationFile , BB.codeLocationObject , BB. codeLocationObjectOffset ,
   BB. instructionCount , SUM(BC.executionCounterSum), SUM(FC.instructionFetches)
FROM ApplicationThread T
JOIN BasicBlockCostCentersAll BC ON BC.base=T. basicBlockCostCentersAllId
JOIN BasicBlockCostCenterRecursion R
  ON R.base = BC. basicBlockCostCenterRecursionId
JOIN FullCost FC ON FC.base = R. costId
JOIN BasicBlock BB ON BB.base = R. basicBlockId
JOIN FunctionNode FN ON FN.base = BB.functionNodeId
GROUP BY BC.basicBlockId
ORDER BY SUM(BC.executionCounterSum) DESC,
              SUM(FC. instructionFetches ) DESC;
```

the time the query took place. By firing queries occasionally during the instrumented execution of `gzip`, `bzip2`, and `egrep` we observe that `gzip` has a small memory footprint of 4.2KB early in the archiving process. Both `egrep` and `bzip2` use 1.5MB of memory on average, and `bzip2` utilise approximately 7.5MB. Notably, the total set size observed in Listing 5.19 is explained by the fact that each record represents shadow memory metadata, such as validity and addressability information for each memory byte associated with a source code location of allocation. Table 5.8 presents the evaluation measurements.

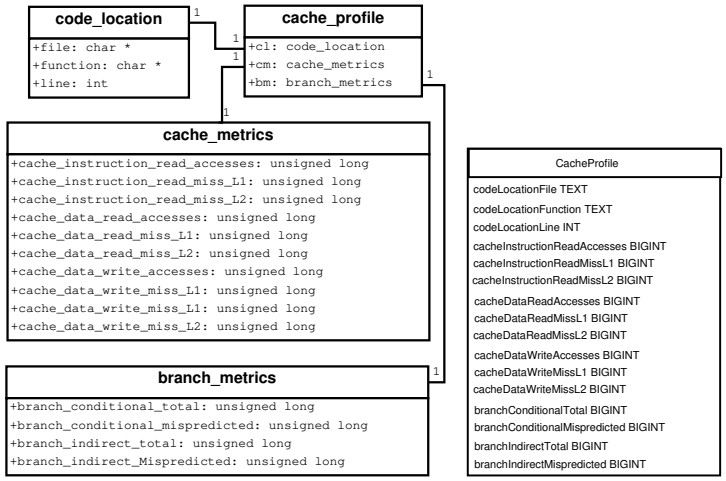
5.3.4 Results

Query measurements for Listings 5.19 and 5.24 are indicative of PiCO QL's scalability. The other queries that execute against a total set size of a dozen to a few thousand records incur roughly the same trivial cost, except for Listings 5.32 and 5.31 that combine a set size of a few thousand records and a *group by* clause. In fact, Listing 5.31 has approximately four times larger record set size than Listing 5.32 and its execution time is proportionally larger. In Listing 5.19, input size grows by a factor of eight from `egrep` to `bzip2` followed by a less than linear increase in computation time by a factor of seven. PiCO QL follows roughly the same performance trend in the query described in Listing 5.24, which is less computationally demanding.

Although PiCO QL seems to scale well as input size increases, querying the shadow memory that tracks large amounts of application memory becomes costly in terms of time. In an additional evaluation, apart from the ones in Table 5.8, we executed the query in Listing 5.19 for a total size of 18.2M records, that is 4B words of shadow memory, which account to 73MB of allocated memory space. It took PiCO QL 1 minute and 20 seconds to evaluate this query. In terms of computation efficiency, PiCO QL required 11 times more time to execute a query with 9 times bigger input size than the most demanding one depicted in Table 5.8. Figure 5.9 depicts PiCO QL query execution time on shadow

Listing 5.32: Retrieve cost factors of basic blocks according to the 20 most frequently executed jump calls initiated from those blocks

```
SELECT BB.memoryAddress, FN.codeLocationFunction, BB.codeLocationLine,
  BB.codeLocationObject, SUM(FC.instructionFetches ), SUM(FC.dataReadAccesses)
FROM ApplicationThread T
JOIN JumpCall JC ON JC.base=T.jumpCallCostCenterId
JOIN BasicBlockCostCenter BC ON BC.base=JC.fromBasicBlockCostCenterId
JOIN BasicBlock BB ON BB.base = BC. basicBlockId
JOIN FunctionNode FN ON FN.base = BB.functionNodeId
JOIN FullCost FC ON FC.base = JC. costId
GROUP BY BC.basicBlockId
ORDER BY SUM(JC.callCounter) DESC
LIMIT 20;
```



(a) Cachegrind model (b) Virtual table schema

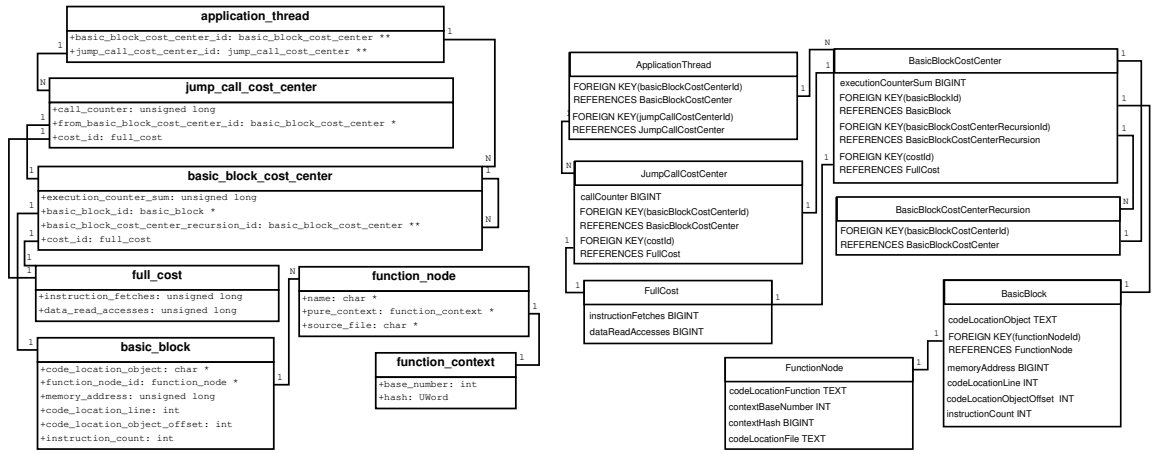
Figure 5.7: Cachegrind's data structure model and relational representation

memory as the amount of application memory increases.

The unintuitive difference that we observe in `gzip`'s case where the query in Listing 5.24 takes more time than the query in Listing 5.19 can be explained by the cost that the SQLite virtual machine imposes, which depends on the query's semantics and for small data sizes is an important factor. Indeed, the former is a sophisticated query with a nested subquery, GROUP BY, and double ORDER BY clauses.

By querying Memcheck's shadow memory (Listing 5.24) during `bzip2`'s instrumented execution we extracted that a single memory block of size 3600136 bytes allocated in *BZ2_bzCompressInit()* suffered a total of 900078 bytes wasted. This means that 900 KB in the same memory block allocated for `bzip2` have all four bytes undefined, which results in 12% memory space waste given that *bzip2* requires 7.5 MB for its operation in total. The issue persisted with a number of different compression tasks. Thus, there is an opportunity for tighter memory organisation.

We focused on *bzip2* to shed light on its symptom. We retrieved source code locations, such as function name and line number, for memory block allocations of significant size. Listing 5.25 presents such a query, which orders stack traces resulting to memory allocations that exceed 1MB of memory by decreasing size. For *bzip2*, *BZ2_bzCompressInit* stands out as expected accommodating 7.5MB of memory allocation, that is almost the total amount requested by *bzip2*. In this way we verified that this function plays the most important role in *bzip2*'s memory allocation operations. Gregg [Gre13,



(a) Callgrind model (b) Virtual table schema

Figure 5.8: Callgrind's data structure model and relational representation

Table 5.8: Query performance measurements

Tool	Memcheck				Cachegrind				Callgrind		
Listing	5.19	5.24	5.23	5.25	5.26	5.27	5.28	5.29	5.30	5.32	5.31
Rows returned											
gzip	1	2	1	12	0	1	6	363	0	871	20
egrep	2	20	9	346	0	7	6	417	0	1066	20
bzip2	1	180	7	123	1	9	11	716	2	1482	20
Total set size (records)											
gzip	1046	1046	2	12	2	425	425	425	3794	1040	3794
egrep	231 502	231 502	29	346	29	479	479	479	5132	1288	5132
bzip2	1 882 884	1 882 884	12	123	12	438	438	438	4378	1046	4378
Time measurements (milliseconds)											
gzip	27.0	11.2	10.65	13.69	14.67	10.73	20.69	9.62	13.57	30.14	100.17
egrep	1070.57	302.53	22.54	10.64	24.51	21.32	14.33	9.22	7.79	24.46	111.22
bzip2	7280.31	2522.54	12.49	17.12	15.23	21.20	16.88	9.37	17.47	32.2	113.72

p.248] describes a similar task with DTrace to summarise the requested size of memory allocations decorated with stack trace metadata.

Then we proceeded to further clarify the organisation of memory allocations for *bzip2*. The query in Listing 5.23 rounds memory allocation sizes at the megabyte (MB) level using integer division, then counts memory allocations per MB, and finally orders them by decreasing count frequency. Memcheck does not output detailed memory block allocation information. In the result set two memory allocation blocks are found in the range between 3 and 4 MBs, while the rest lay below 1 MB. Gregg [Gre13, p.247] lists an identical task with DTrace to summarise the requested size of memory allocations for a specific process presented as a power-of-two frequency distribution format.

Because *BZ2_bzCompressInit* is the key to *bzip2*'s memory allocation routines we examined it alone. The query in Listing 5.26 retrieves the allocated memory blocks for that function only. Four blocks were reported and two of them were distinct in size as also indicated by the result set of the query in Listing 5.23. The largest block's starting address matched the address where the 900KB of unusable memory were found. With a further query not presented in this paper we identified that all 900KB of unusable memory are contiguous, they are located at the tail of the block, and all individual bytes are tagged as undefined.

The previous three queries helped collect information about *bzip2*'s allocation operations, but they did not reveal the origin of the issue. Hence, we decided to examine the source code. Unfortunately, Memcheck did not recover source line number information, but by examining *BZ2_bzCompressInit*'s implementation in *bzip2 v1.0.6*'s source we concluded that the issue rests in the lines listed in List-

Listing 5.33: Source code lines in *bzlib.c* that trigger a large number of undefined or partially defined bytes.

```
      // BZ_N_OVERSHOOT = 34 (calculated from other constants)
      // blockSize100K = 9 (by default unless option is passed)
#176  n          = 100000 * blockSize100k;
#177  s->arr1 = BZALLOC( n * sizeof(UInt32) );
#178  s->arr2 = BZALLOC( (n+BZ_N_OVERSHOOT) * sizeof(UInt32) );
```

Table 5.9: Expensive jump calls between basic blocks

Tool	From function	Line	To function	Line	Execution Counter	Instruction Fetches
sort	*strcoll*	36	*strcoll_l*	473	27 007 779	33 406 087 951
	strcoll_l	593	*get_next_seq*	154	82 915 296	10 262 514 873
	strcoll_l	595	*get_next_seq*	154	82 915 295	10 267 926 563
	strcoll_l	504	*strlen*	66	27 007 778	1 674 860 884
	strcoll_l	505	*strlen*	66	27 007 778	1 674 836 034
uniq	*strcoll*	36	*strcoll_l*	473	16 804 718	5 165 064 625
	strcoll_l	593	*get_next_seq*	154	9 273 269	1 133 162 351
	strcoll_l	595	*get_next_seq*	154	9 273 269	1 130 978 530
	strcoll_l	504	*strlen*	66	8 402 359	537 732 818
	strcoll_l	505	*strlen*	66	8 402 359	537 732 818

ing 5.33. The data structures in lines 177 and 178 are used for performing block sorting during a compression operation. The two large blocks we identified with the queries were of size 3600136 and 3600000 bytes. By calculating the requested bytes for memory allocation from the source in Listing 5.33 we derive that the largest block of 3600136 bytes corresponds to line 178. This is the block that contains the large chunk of undefined memory.

After modifying the source code line responsible for this memory allocation we verified that the large amount of undefined bytes disappeared, Memcheck used 12% less memory, and continued to operate correctly in a number of different compression tasks that we tried including *bzip2*'s installation tests.

To discover performance-critical code we observe execution counters related to a basic block execution and the accumulated cost in terms of instruction fetches for the associated execution context. The query in Listing 5.31 selects the basic blocks and associated metadata that contribute the most instruction execution cost. For *sort* and *uniq* the basic blocks that stand out for the execution cost they incur are contained in functions *strcoll* and *get_next_seq*.

To go one step further we investigate jump calls, such as method calls between basic blocks that account for a significant part of execution cost, for example due to repeated invocations of a method. The query in Listing 5.32 groups recorded jump calls per basic block that initiates the call, aggregates the execution counter and number of instructions fetched for each block and orders the jump calls by execution counter of the initiating block. The query returns the top 20 basic block pairs by using a LIMIT clause. As Table 5.9 presents, we can identify two hot code execution paths, that is *strcoll* to *strcoll_l*, and then the latter calls either *get_next_seq* or *strlen*. Source code locations in Table 5.9 correspond to *glibc v2.19*. This finding is in accord with the result of the query in Listing 5.32. Indeed a later version of *glibc* (*v2.21*) has *get_next_seq* inlined to *strcoll_l* to improve the latter's performance (see `http://bit.ly/24zvVHM`).

5.4 User study

Our empirical study regards the user-based evaluation of PiCO QL compared to a popular alternative, Python scripting, on the Valgrind instrumentation framework. To provide essential background for

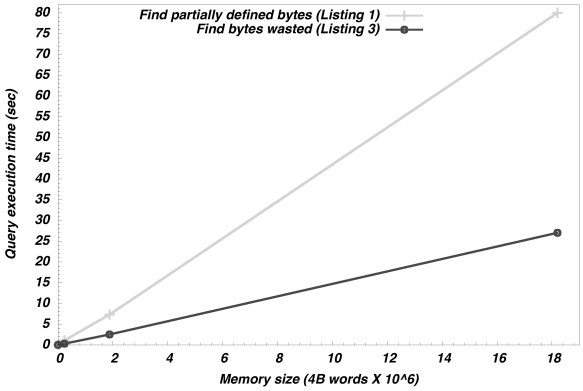


Figure 5.9: PICO QL's scalability

our tool, we position it in existing taxonomies of software (Section 5.4.1). We describe in detail the motivation behind our experiment and the questions we try to address (Section 5.4.2). Then, we explain the experiment's setting (Section 5.4.3), document the evaluation process of experiment data (Section 5.4.4), and present the results (Section 5.4.5). We finish with the presentation of user interface characteristics that our users consider important after performing the evaluation (Section 5.4.6).

5.4.1 Software typology

In the vast taxonomy of software, our software can be regarded as data management software in the category of application software according to the list of software categories in Wikipedia [Theb]. In a more fine-grained taxonomy, our tool would be placed in a subcategory titled query interfaces.

From the perspective of a query language interface taxonomy, our tool accepts queries input from keyboard using a keyword command language and presents output on screen according to the framework introduced in reference [JV85].

5.4.2 Motivation and questions

Our motivation for this empirical study stems from key differences between two general types of interfaces: declarative vs imperative interfaces and live interactive vs post-mortem interfaces.

Declarative interfaces express the logic of a task, that is, *what* is the desired objective. On the other hand, imperative interfaces provide a list of steps for performing a task, that is, *how* the desired objective will be achieved. Procedural interfaces belong to the imperative paradigm.

The booming of declarative interfaces for a variety of imperative programming models, such as object-oriented [Mei11, WPN06] and Map-Reduce [TSJ+09, ORS+08] implies advantages of these interfaces complementary to corresponding interfaces for these programming models. Advantages could be attributed for instance to the higher-level of abstraction declarative interfaces offer and the economies of scale in reusing a standard query language for data analysis.

Live interactive interfaces are an important feature of many successful lower-level tools, such as DTrace [CSL04], Systemtap [PCE+05], and GDB [SS96]. On the one end such tools typically dig into an application's or operating system's internals; on the other they present a safe, usable interface that users can interact with at runtime. Although a human factors experiment with these tools has not been reported, we think that their runtime interactive interface contributes significantly to their success.

The complementary advantages of declarative live interactive interfaces could further enhance the usefulness and usability of the Valgrind framework for its users. These measures have been es-

tablished as key determinants of user acceptance in the field of information systems [ANT92, Dav89, Ven00, VD00]. Usability measures the ease of use and learnability of a system.

In the absence of existing empirical evidence that suggests PICO QL is better than Python scripting or vice versa, we want to prove or disprove our claims by answering the following questions:

Q1. is PICO QL *more useful* than Python scripting on files?

Q2. is PICO QL *more usable* than Python scripting on files?

Human factors experiments on programming languages for data manipulation, hence query languages, have a long history, but there is no recent indication to describe the current situation. In addition, the results of studies from the distant past present conflicting results. One research stream in this area related to our study focuses on the effect of a query language's procedurality in user performance [WS81] and productivity [HM85] respectively. On this track, an experiment was performed between SQL and TABLET. TABLET, The Algebra Based Language for Enquiring of Tables, is a relationally complete query language that, like SQL, uses Codd's relational model. TABLET is more procedural than SQL because it uses procedural statements within a query, such as FORM to create a working table from specified columns of a table and PRINT to output the query's result set. TABLET proved advantageous for writing more difficult queries than SQL [WS81]. The study's result was attributed to the more procedural approach that characterised TABLET queries compared to the less procedural SQL queries. The authors developed a metric of procedurality for the experiment. Notably, despite having some level of procedurality, SQL is considered a nonprocedural language [Rei81]. On the other hand, COBOL, a third-generation procedural language, lagged with respect to productivity and efficiency when compared to a fourth generation non-procedural language [HM85].

A number of comparative experiments between query languages of different procedurality have also been conducted. We focus on two [YS93, BBE83] that regard SQL and query by example (QBE), a graphical database query language, which is less procedural than SQL. In an online testing setup carried out in [YS93] user performance was found unaffected of query language type. The other experiment, focused on the easiness of learning the two languages [BBE83]; SQL proved easier to learn except for queries that concerned more than one tables. Most users preferred SQL to QBE.

In addition to the above key characteristics we are also interested in rating developer productivity with the tools, that is, how close to achieving an analysis goal users came with each tool and how efficiently. We denote those two characteristics performance and effort respectively. Because a programming language's expressiveness affects user effort, we also examine the languages' expressiveness through the drafted code. McConnell [McC04] reports that higher-level languages, such as C++ and Java, are more expressive than lower-level languages, such as C. An empirical study examines programmer productivity and programming language expressiveness measured in lines of code with seven different programming languages [Pre00], but SQL is not included. Moreover, we found no recent empirical study between SQL or a declarative language and Python or other imperative language. In lack of stronger empirical evidence, we set out to answer the following questions:

Q3. is PICO QL *more expressive* than Python scripting on files?

Q4. do users require *less effort* with PICO QL than with the Python scripting approach?

Q5. do users *perform better* with PICO QL than with the Python scripting approach?

Thus, we measure, usefulness, usability, expressiveness, effort, and performance by means of proxies that we describe in Section 5.4.4. These are the dependent variables of the software used in the experiment. We measure expressiveness with lines of code after establishing ground rules about our measuring approach. Users rate effort in time units for each query immediately after drafting it.

With performance we measure a query's syntactic correctness and whether it reflects a given analysis task description. We choose the term performance because queries that are correct and reflect the rationale of a given analysis task help users achieve the goal of their analysis.

5.4.3 Experimental design

5.4.3.1 Users

The experiment took place after the spring semester of 2015. We considered final year undergraduate students that selected the elective course Advanced Topics in Software Engineering. Ten students volunteered for the experiment.

We followed a between groups experiment approach, where one group drafted SQL queries with PiCO QL and the other scripting with Python. Thus, students formed two groups of five students each. Students joined the group that most matched their core competences.

All students reported one to three years of experience with the target programming language of their group. All students had attended a university course on both Python and SQL. In addition, all members of the Python group had undertaken a Python project and two of those had a job as Python developers. At the SQL group, one student had worked in an SQL project and another worked as SQL developer.

5.4.3.2 Setting

We provided each group with a description of the SQL or Python data model accordingly. The experiment consisted of answering ten questions by writing queries in SQL or Python respectively. Both groups worked on the same set of ten questions.

In order to have an equal basis of comparison between the two groups, the Python group received setup assistance. Because Valgrind tools store a subset of their collected metadata in files, parsing of the files is then required before users can employ Python to analyse the metadata. We provided both Valgrind tools' reports and Python code for parsing them to the Python group. Users only wrote Python code for each query in boilerplate files, which they could then execute with the Python interpreter to examine the produced output. On the other hand, the PiCO QL group drafted SQL queries on PiCO QL's web interface, which interacted with an active instrumented execution of an application.

Before drafting queries, users filled in an online form regarding their level of competence in SQL or Python respectively. During the experiment, users provided each query corresponding to each task description on the online evaluation form mentioning also whether they used online help for a query and how much time it took them to answer it. After the experiment, users rated in Likert 1–5 scale each of the evaluation criteria, that is, usefulness and usability, of the tool they used by answering twenty questions, ten for each evaluation criterion. We adopted the questions for measuring usefulness from reference [Dav89] and usability from an online source.[13] The complete form for each group is available online. [14] [15]

5.4.3.3 Tasks

We provided tasks of varying difficulty to be able to examine thoroughly the quality characteristics of each of the two interfaces. Questions followed an easy/medium/hard division. Out of ten questions, two were easy, six were of medium difficulty, and two were hard. The division was based on the language features and combinations of those needed to use in order to answer a question.

[13] http://www.measuringu.com/sus.php
[14] https://docs.google.com/forms/d/e/1FAIpQLSdgSUCUL-RrTMepOhI5TI59etB5PsMkiGGfBXCY-l9L4sSeIQ/viewform
[15] https://docs.google.com/forms/d/e/1FAIpQLSejXMhnxFIRK7ceOKicWHMccpnCebgxXQWMKwZhWAvLq0QBtQ/viewform

Table 5.10: Query characteristics, expressiveness, and performance by level of difficulty for all tasks

Query characteristics		Expressiveness (Lines of code)		Correct queries (Number)	
PiCO QL	Python	PiCO QL	Python	PiCO QL	Python
Easy					
SELECT FROM	*for*	2.0	2.0	5	4
SELECT FROM	*for*	2.6	4.5	5	4
Medium					
COUNT GROUP BY	*for, sum, dict*	3.0	6.0	4	4
JOIN ON	*for, dict*	2.7	3.0	4	4
JOIN ON, SUM	*for, sum, if*	3.0	1.3	1	3
WHERE, ORDER BY DESC	*for, if, sorted, lambda*	4.0	6.6	4	5
AVG GROUP BY, ORDER BY ASC	*for, if, dict, sum, sorted, lambda*	4.0	16.0	4	1
SUM, WHERE LIKE, ORDER BY DESC	*for, if, dict, sorted, itemgetter*	4.7	8.0	4	4
Hard					
4 X JOIN ON, GROUP BY, ORDER BY DESC, LIMIT	*for, dict, sum, lambda, if*	0.0	0.0	0	0
5 X JOIN ON, GROUP BY, ORDER BY DESC	*for, dict, sum, sorted, itemgetter*	0.0	3.0	0	1

Table 5.11: Average student scores and p-value per evaluation criterion

Criterion	PiCO QL group					p-values			Python group					
	1	2	3	4	5	Avg	Mann-W	Levene	Avg	1	2	3	4	5
Usefulness	3.6	4	4.4	3.7	3.7	3.88	0.17	0.19	2.94	4.1	2.4	4	1.8	2.4
Usability	3.1	2.5	3.2	3.0	3.1	2.98	0.54	0.52	2.96	2.6	2.7	3.2	3.6	2.7
Effort	513.33	277.1	561.5	350.4	645.5	459.3	0.004	0.35	789.8	746.66	734	850.75	924	693.6
Performance	4.4	7.6	5.9	7.9	6.8	6.52	0.21	0.83	5.42	6.3	2.8	6.4	4.6	7.0
Expressiveness	3.50	3.3	3	3.2	3.1	3.23	0.30	0.02	5.3	8	6	2.2	7.2	3

5.4.4 Experimental evaluation

Our evaluation scale to judge correctness of the queries was based on the method introduced in reference [WS81]. With the application of this scale we derived the scores of the performance dependent variable. Each answer matched one of the following categories: correct, minor language error, minor operand error, minor substance error, correctable, major substance error, major language error, incomplete, and unattempted. According to this method an answer placed within the top four categories was considered correct because it contained only minor mistakes. We adopted this approach in our study. Table 5.10 presents the language features, number of correct queries, and query language expressiveness per difficulty level.

We performed post-processing work on the experiment data to be able to run statistical tests. First, we evaluated the correctness of queries provided by users as answers to the given tasks according to the scale described in the previous paragraph and produced performance scores in scale 1–10. In addition, we manually transformed the time required to draft each query from the form that each user provided it to seconds and measured lines of code both in SQL queries and in Python scripts. In SQL queries we treated each SELECT, FROM, JOIN ON, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT clause as a separate line of code, while in Python scripts we treated as a line of code what the Python interpreter would take as one. For instance, we considered a sizeable Python comprehension that contained two for loops and two if conditions as one line of code, but we regarded an if condition with one statement as two lines of code.

Afterwards, we computed a user's average score across all a) ten questions for each of the evaluated criteria on the form, that is usefulness, and usability, b) tasks to measure performance, c) attempted tasks to measure effort, and d) correct tasks to measure expressiveness.

We then conducted a Kolmogorov-Smirnov test to identify whether the scores on each of the cri-

teria followed a normal distribution. Because they did not, we performed Mann-Whitney's unpaired non-parametric one-sided test at the 0.99 confidence level to check whether PiCO QL proved statistically better than Python on each of the dependent variables. We also carried out Levene's test to test equality of variances between the two groups' scores for each dependent variable. The input dataset for each group contains all average user scores on usefulness, usability, and performance. The dataset also includes users' average effort scores only for attempted tasks and average expressiveness scores only for correct tasks to avoid distortion.

5.4.5 Experimental results

Table 5.11 presents the datasets and the p-values provided by the statistical tests for each variable.

Q4. The between groups experiment revealed statistically significant differences in user effort with 99% confidence in favour of PiCO QL. A possible explanation for the differences is that writing SQL queries that execute on arbitrary data structures provides a more comprehensive abstraction for analysis than traversing the data structures using iterators or functional constructs; the latter impose implementation details that users do not need to know. In pair with the language, the relational data model might provide friendlier conceptual representation for users to grasp and use. In light of the statistical evidence, we accept this hypothesis.

Q1. Q3. Q5. Although PiCO QL had better average scores in usefulness, performance, and expressiveness, the differences were not statistically significant for our sample. This finding discourages our claims that PiCO QL is better than Python in the examined dimensions. Therefore, we reject hypotheses Q1, Q4, and Q6. A glimpse over the average user scores between the two groups for each of the discussed measures show PiCO QL scores have substantially lower variance than Python scores. Levene's test shows equality of variance for expressiveness scores between the two groups. This suggests that the observed differences might as well be a random effect. But for usefulness and performance where equality of variance is not proven it might be the case that for this level of knowledge an average user finds our tool more useful and performs better with our tool and is required to write fewer lines of code on average in less time than Python.

Q2. PiCO QL and Python received approximately the same scores on usability. Consequently the answer to Q2 is negative. This is not what we anticipated. We expected that PiCO QL's live interactive user interface with the data model representation on it would make a difference in these respects. We observed, however, elements of this difference from other experiment data presented in Section 5.4.6. According to a user interface characteristics rating presented in Section 5.4.6 the data model representation on the query interface received almost unanimous preference. Because users only evaluated one tool, as a next step the scores could be further clarified and validated with an additional experiment where users evaluate both. Another side note is that the between groups approach imposes an indirect comparison between the two competitors. If users evaluated both tools, the results would provide clearer evidence as to score differences.

5.4.6 User interface preferences

A further question we would like to answer is how good an interface our work provides for analysing an application's memory operations profile and how close our proposed interface falls to users' needs. For this reason at the experiment's end we asked participating users to select useful interface characteristics from a list in order to see what kind of interface the preferences sketch. Figure 5.10 presents user preferences.

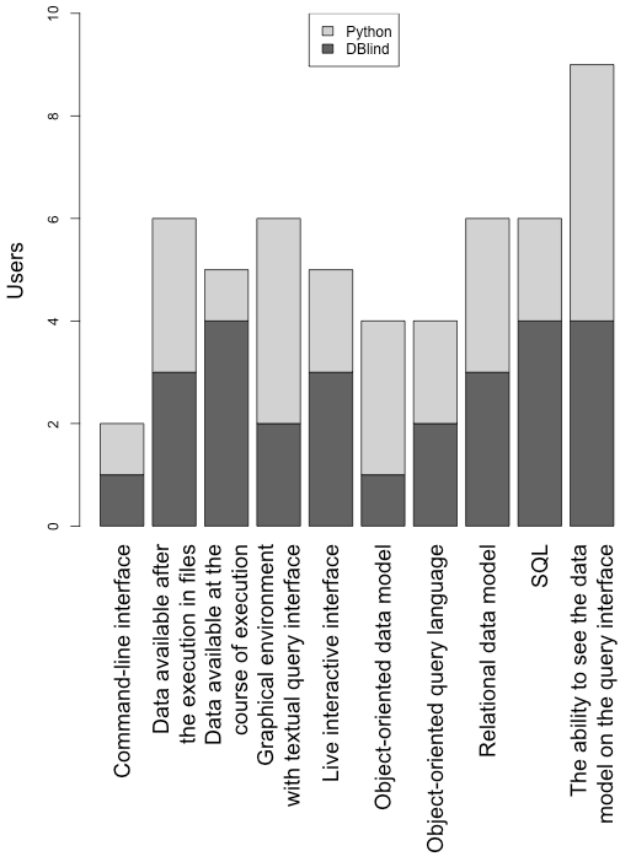


Figure 5.10: User preferences of interface characteristics

A straightforward and important finding is the need to present the data model on the query interface. All but one user agreed on this including the whole Python user group whose evaluation interface lacked this feature. Then there are four axes of comparisons. First, on command-line vs graphical environment with typed input the latter clearly wins. In fact it collects 4 out of 5 votes from Python's user group, which implies that they would prefer a graphical environment. Second, data availability for analysis during instrumented execution and post-mortem manipulation preferences are almost equally divided. The PiCO QL user group appreciates more the availability of live data than the Python group. A potential explanation for Python group's low interest is that their evaluation was completely disconnected from an instrumented execution in order to relieve users from a dump-parse-extract cycle of Valgrind data. On a related note, the availability of a live interactive interface gathered half of the total votes; we expected more especially from the PiCO QL group. Third, on object-oriented vs relational data model we identify a tangible precedence of the latter owed to the PiCO QL group. The Python group voted both data models equivalently. Fourth, the bars of object-oriented query languages and SQL show that SQL attracted more user votes. Compared to the data model comparison, votes are more clearly divided between the two groups. Even though only two members of the Python group voted for SQL, two members of the Python group, probably the same, in their reply to our thank you note for participating in the experiment commented that they missed SQL in the experiment process.

5.4.7 Threats to validity

There are three main threats to validity related to our study. First, our user sample is small and focused, thus it may not be representative of the population of programmers or data analysts. There are pros and cons in our sample's characteristics. As undergraduate students, users had a few years of programming experience, but for the same reason they an had open mind regarding the tools' evaluation.

Second, our study may be limited to the Valgrind instrumentation framework and its results may not be generalisable to other systems and platforms. Our study evolved around Valgrind, thus we cannot claim the application of our study's results to other systems. As a side note, users' interaction with Valgrind happened through the PiCO QL or Python interface and none of the users had ever used Valgrind; they carried out the experiment given a documentation of each Valgrind tool's PiCO QL or Python data model. Consequently, whether our findings present in other systems or not is an open question.

Third, the fact that we marked ourselves the SQL queries and Python scripts drafted by subjects in the user study threatens the validity of the study. The criteria for marking come from a published empirical study [WS81] and we provide the whole material of the user study, that is, the question forms and answers, available online to allow the reproduction of the study's results.

5.4.8 Limitations

We note two limitations regarding this work, queries to data structures of big size and synchronous live interactive queries.

The main limitation of this work is the inability to handle big data, such as querying byte to byte shadow memory information about allocated memory blocks when these account to hundreds of MBs, not to mention GBs. The computational cost becomes too high for thousands or more of millions of words in our single-threaded implementation. One alternative is to introduce multiple threads in order to achieve performance boost. In addition, we can specify the LIMIT clause in SQL queries to reduce the computational burden but this is not a generic solution as it would not restrict queries that would return fewer rows than what the LIMIT clause specifies. In addition, queries with ORDER BY and GROUP BY clauses have to be evaluated completely, grouped and/or ordered and then have their result set limited. Another alternative that can reduce computation is to introduce sampling according to a memory block's size and return an indication of the instrumented application's internal memory organisation. At the protection front, an emergency button in the user interface, which terminates a heavy duty query and returns the current result set, and perhaps a progress bar and a timeout mechanism are the most trustworthy solutions to this problem. Finally, since a selection of memory allocation blocks is very cheap in computational terms, it would allow PiCO QL to query shadow memory for these blocks with limited possibility to timeout.

Synchronous live interactive queries require synchronisation primitives, which the Valgrind framework does not use. If it did use, PiCO QL would be able to share them through simple definitions written in the DSL description. Currently PiCO QL's interface is configured to poll for queries. In this case, PiCO QL occasionally polls for a query and if one is in line, then it stops the world, performs the query, and handles control back to the Valgrind tool. Asynchronous queries provide an alternative with satisfying characteristics, that is, low overhead and programming effort to implement and interactivity with the evolving state of an instrumented application's memory profile given some, adjustable, latency. Another possibility is to use a snapshot with copy on write for querying. Finally, Valgrind can collaborate with GDB for debugging purposes.

Chapter 6

Conclusions and Future Work

This thesis presented the design and implementation of an approach for mapping imperative programming models to a relational query interface. In this chapter we summarise the results of our research, present the overall contribution of our work, discuss potential avenues for future work, and conclude our thesis.

6.1 Summary of results

Our approach allows SQL queries to execute on a program's main memory data structures through a relational representation that we introduce. The implementation, PiCO QL, delivers a usable SQL interface for interactive, ad-hoc queries to C/C++ application data structures. Its evaluation within three C++ applications shows query expressiveness, scalable query speed, and a low memory footprint. For applications that only require a DBMS's query facilities, a fully-fledged DBMS is superfluous. Introducing it would mean adding an intrusive dependency, extra overhead, and writing boilerplate code for interacting with the application, which would litter application code. For applications with heavy online processing, such as scientific computing and visualization applications, interactive queries are both tough and important.

We show that PiCO QL is advantageous in situations that require managing software state, such as operating system diagnostics. Our relational interface to accessible Linux kernel data structures delivers custom high level views of low level system data, which are hidden in complex data structures. The implementation is type-safe, secure, and provides consistent kernel state views for data structures protected in critical sections that do not involve blocking. We exemplify PiCO QL's contribution in diagnosing security vulnerabilities, operation bugs, and performance issues. Our evaluation demonstrates that this approach is efficient and scalable by measuring query execution cost and the impact on system performance. We present a small number of queries on top of PiCO QL's relational interface, which currently amounts to 50 virtual tables. However, the kernel's relational representation with PiCO QL is only limited by the kernel's data structures. In fact, it is easy for everyone to roll their own probes by following the tutorial available online [M. 13].

Finally, we apply our live interactive SQL interface on a running application's memory profile. We argue that our interface increases productivity in analysing an application's memory profile. First, our work reduces the analysis lifecycle significantly because it realises live interactive queries for Memcheck and Cachegrind. This is particularly useful in analysing long-running applications, such as Firefox and OpenOffice. Second, PiCO QL can reduce programming effort more without sacrificing either readability or expressiveness by expressing sophisticated queries that build on the shoulders of others using relational views. Finally, PiCO QL exposes a wealth of collected metadata hidden in Valgrind tools' data structures.

Our evaluation with Valgrind reveals a number of intended uses for PiCO QL; three of those follow.

First, we note the leveraging of Memcheck's shadow memory to examine the internal organisation of allocated memory blocks. A second practical use case is to observe the evolving cache utilisation cost ranks attributed to specific source code locations. Finally, we highlight the examination of evolving call history related costs and specifically computationally expensive jump calls.

According to the results of our user study, users want a data model representation on a graphical user interface and prefer querying a relational data model with SQL to an object-oriented alternative. The results are not clear on the relative importance between live and post-mortem data availability. Live and post-mortem interfaces seem to be complementary useful.

We list below the specific results of this dissertation.

- A method for representing main memory data structures in relational form [FSL15]
- An implementation for querying main memory data structures of C and C++ applications with SQL [FSL16]
- An evaluation of the expressiveness, temporal efficiency, and spatial efficiency of our approach within three C++ applications, a virtual observatory of stellar objects, a GIS application, and a source code analyzer.
- A Linux kernel diagnostic tool that provides SQL queries on the kernel's data structures
- A method for extracting consistent views of the Linux kernel's state transparently
- An evaluation of the usefulness of our kernel diagnostic tool for diagnosing problems with the system's operation. Our tool spots bugs, security vulnerabilities, and opportunities for performance optimisations. The overhead of our approach to the system's operation is acceptable [FSLB14]
- A diagnostic tool that extends the Memcheck, Cachegrind, and Callgrind tools of the Valgrind instrumentation framework [FSL19]. Our extension provides SQL queries to the tools' gaathered metadata for analysing memory profiles.
- An evaluation of the usefulness of our Valgrind diagnostic tool for analysing memory profiles, which shows that it identifies opportunities for performance optimisations in the way applications use the hierarchy of memory
- A user study that tackles the usefulness, usability, effort, performance, and expressiveness of our approach. We measure usefulness and usability through qualitative analysis of questionaires. We measure effort, performance, and expressiveness quantitatively through the evaluation of the code written. Two groups of six students each express data analysis tasks with SQL queries and Python code respectively. The results of the statistical tests on the students' scores show that user effort is lower with our approach. PiCO QL also scores higher in usefulness, performance, and expressiveness, but the differences are not statistically significant. Finally, in terms of usability both approaches score equally well.

6.2 Overall contribution

We describe the contribution of our work on three fronts: science and research, software community, and business. Regarding the first front, our work's research results are described in Section 6.1.

Our contribution to the software community is the availability of the PiCO QL library including the Linux kernel and the Valgrind diagnostic tools as open source software hosted at Github.[1] Table 6.1 shows the metrics of our library regarding its use.

[1] https://github.com

Table 6.1: Usage metrics of the PiCO QL software library

Downloads	182
Stars	8
Forks	3

On the business front, Genesys [2] a California-based company that offers telecommunications and customer experience services uses PiCO QL for debugging purposes. The specific use case regards issuing SQL queries against a hierarchical C++ data model where containers hold string elements and other containers thereby forming parent-child relationships.

The concept behind our work may have contributed to the birth of a similar diagnostic tool that has become very popular. A few months after the PiCO QL kernel module's publication, Facebook released *osquery*,[3] an open source software project that supports SQL queries to system data made available by the operating system system components, and applications. It uses SQLITE's virtual tables to create a relational interface to the various data sources. The specific common points and differences with PiCO QL are detailed in Section 2.3.1. Osquery has accumulated more than seven thousand stars and nine hundred forks so far at Github. According to its website, osquery is currently used by a number of very demanding enterprises including Facebook.

6.3 Future work

We consider two possible avenues for future work with PiCO QL. The first regards its use within embedded systems and the second its ability to perform stream processing.

Embedded systems usually have tight requirements regarding the computing resources they can provide and the software facilities they can support. PiCO QL combines native code, compact configuration, and low memory requirements at runtime. These characteristics make it a compelling alternative for use in an embedded system.

PiCO QL can deliver analytics and diagnostics services in the form of ad-hoc SQL queries. It can execute user-driven queries to the data that reside in an embedded system. In addition, PiCO QL can present views of the state of the embedded system using its data in order to unveil concerns with the system's health.

The second potential research avenue regards processing of structured data that flow in a stream with PiCO QL. Given an incoming stream, a relational specification of its data, and a window PiCO QL will generate the relational interface to the data, compile a minimal application, and execute it to present a live interactive relational interface to the streaming data. Users will be able to enter SQL queries to PiCO QL's web interface and visualise data produced by queries scheduled to run at fixed intervals. Our research will focus on stream processing aspects and methods for parsing the data stream into a structured form appropriate for querying with SQL.

6.4 Conclusions

This thesis documented the problems with the analysis of program and system data and an architecture that promises cost effective system management, simplified programming, and integrated data management. For the problems regarding the analysis of data we presented a method and an implementation for providing SQL queries on main memory data structures. We described the evaluation

[2] http://www.genesys.com
[3] https://osquery.io

of this work on three C++ applications, the Linux kernel, and the Valgrind instrumentation framework. The evaluation suggested that our work is:

- useful, through the identification of security vulnerabilities in the Linux kernel and opportunities for performance optimisations in applications instrumented by Valgrind
- usable, through a user study that compared PiCO QL with Python scripting

We hope that out work will help in the race towards more efficient processing and effective analysis of the available data.

Appendix A

Appendix

A.1 C++ and SQL code for the queries used in the evaluated applications

The mark *x* in comments denotes that the corresponding lines have not been accounted for the program's lines-of-code metric.

A.1.1 Stellarium

Listing A.1: UC1

```
/* C++ lines of code: 14
 * x: excluded
 */
void cpp_query1() {
  PlanetP iter;
  std::set<double> resultsetP;
  double min_distance;
  std::vector<Meteor*>::iterator it;
  for (int i = 0; i< 100; i++) {                          /* x */
    foreach (iter, solar->getAllPlanets()) {
      if (strcmp(iter.data()->getNameI18n().toStdString().c_str(), "Earth"))
        resultsetP.insert(iter.data()->getDistance());
    }                                                     /* x */
    min_distance = *min_element(resultsetP.begin(), resultsetP.end());
    if (meteor->getActive() && meteor->getActive()->size() > 0) {
      std::cout << "Min distance is :: " << min_distance
                << std::endl;                             /* x */
      std::cout << "observdistance | velocity | "         /* x */
                << " magnitude | scalemagnitude "         /* x */
                << std::endl;                             /* x */
      for (it = meteor->getActive()->begin(); it != meteor->getActive()->end(); it++) {
        if (((*it)->xydistance > min_distance) && ((*it)->alive)) {
          std::cout << (*it)->xydistance << " | "
                    << (*it)->velocity << " | "           /* x */
                    << (*it)->mag << " | "                /* x */
                    << (*it)-> distMultiplier             /* x */
                    << std::endl;                         /* x */
        }                                                 /* x */
      }                                                   /* x */
    }                                                     /* x */
  }                                                       /* x */
}                                                         /* x */
```

Listing A.2: UC2

```
/* C++ lines of code: 22
 * x: excluded
 */
void cpp_query2(SolarSystem *s) {
  PlanetP iterP, iterSP;
  float minAxisRotation, spAxisRotation;
  bool inserted;
  std::multimap<float, std::pair<float,std::string>> aggregate;
  std::pair<float, std::string> axisName;
  std::multimap<float, std::pair<float, std::string>>::reverse_iterator aggrRit;
  for (int i = 0; i < 100; i++) {                                    /* x */
    minAxisRotation = 100000000;                                     /* x */
    inserted = false;                                                /* x */
    foreach (iterP, s->getAllPlanets()) {
      inserted = false;
      minAxisRotation = 100000000;
      if (StelApp::getInstance().getCore()->getProjection(
          StelApp::getInstance().getCore()->getHeliocentricEclipticModelViewTransform())->checkInViewport(
              iterP->screenPos)) {
        foreach (iterSP, (*iterP).satellites()) {
          spAxisRotation = (*iterSP).axisRotation;
          if ((*iterP).axisRotation > spAxisRotation) {
            if (spAxisRotation < minAxisRotation) {
              minAxisRotation = spAxisRotation;
              inserted = true;                                       /* x */
            }                                                        /* x */
          }                                                          /* x */
        }                                                            /* x */
        if (inserted) {
          axisName=make_pair(minAxisRotation, (*iterP).getNameI18n().toStdString());
          aggregate.insert(std::pair<float, std::pair<float, std::string>>((*iterP).axisRotation, axisName));
        }                                                            /* x */
      }                                                              /* x */
    }                                                                /* x */
    std::cout << "name | PlanetRotation | "
              << "MinSatelliteRotation " << std::endl;               /* x */
    for (aggrRit = aggregate.rbegin(); aggrRit != aggregate.rend(); aggrRit++) {
      std::cout << (*aggrRit).second.second << " | "
                << (*aggrRit).first << " | "                         /* x */
                << (*aggrRit).second.first                           /* x */
                << std::endl;                                        /* x */
    }                                                                /* x */
    aggregate.clear();                                               /* x */
  }                                                                  /* x */
}                                                                    /* x */
```

Listing A.3: UC3

```
/* C++ lines of code: 20
 * x: excluded
 */
void cpp_query3(SolarSystem *s) {
  PlanetP iterP, iterSP;
  std::multimap<std::string, std::pair<int, Planet *> > aggregate;
  std::pair<int, Planet *> pSatellites;
  std::multimap<std::string, std::pair<int, Planet *> >::iterator aggrIt;
  int count = 0;
  Planet *currentP;
  for (int i = 0; i < 100; i++) {                          /* x */
    foreach (iterP, s->getAllPlanets()) {
      count = 0;
      if (StelApp::getInstance().getCore()->getProjection(
          StelApp::getInstance().getCore()->getHeliocentricEclipticModelViewTransform())->checkInViewport(
            iterP->screenPos)) {
        foreach (iterSP, (*iterP).satellites()) {
        if ((*iterSP).hasAtmosphere())
          count++;
        }                                                  /* x */
        if (count > 0) {
          pSatellites = std::make_pair(count, iterP.data());
          aggregate.insert(std::pair<std::string, std::pair<int, Planet*>>(
            (*iterP).getNameI18n().toStdString(), pSatellites));
        }                                                  /* x */
      }                                                    /* x */
    }                                                      /* x */
    std::cout << " name | radius | "
              << "period | albedo | NoSatellitesAtm "     /* x */
              << std::endl;                                /* x */
    for (aggrIt = aggregate.begin(); aggrIt != aggregate.end(); aggrIt++) {
      currentP = (*aggrIt).second.second;
      std::cout << (*aggrIt).first << " | "
                << currentP->getRadius() << " | "          /* x */
                << currentP->getSiderealDay() << " | "     /* x */
                << currentP->albedo << " | "               /* x */
                << (*aggrIt).second.first                  /* x */
                << std::endl;                              /* x */
    }                                                      /* x */
    aggregate.clear();                                     /* x */
  }                                                        /* x */
}                                                          /* x */
```

A.1.2 QLandKarte

Listing A.4: UC4

```
/* C++ lines of code: 37, x: excluded */
float fn(float max, float current) {
  return max >= current ? max : current;
}                                                    /* x */
void cTrackDB_cpp_query1(CTrackDB *c) {
  QMap<QString, CTrack**> m = (QMap<QString, CTrack *> *)&c->getTracks();
  QMap<QString, CTrack*>::iterator iter;
  QList<CTrack::pt_t>::iterator it;
  std::vector<std::multimap<double, float> > resultset;
  std::multimap<float, std::pair<double,QMap<QString, CTrack*>::iterator > > aggregate;
  std::vector<std::multimap<double, float> >::iterator rsIt;
  std::multimap<double, float>::iterator groupBy;
  std::pair<std::multimap<double,float>::iterator, std::multimap<double,float>::iterator > ret;
  float speed_max;
  double currentAz;
  std::pair<double, QMap<QString, CTrack*>::iterator > p;
  std::multimap<float, std::pair<double, QMap<QString, CTrack*>::iterator > >::iterator aggrIt;
  double azimuth, totalDistance, ascend, descend;    /* x */
  int totalTimemoving, totalTime;                    /* x */
  std::string name;                                  /* x */
  CTrack *curTrack;                                  /* x */
  for (int i = 0; i < 10; i++) {                     /* x */
    for (iter = m->begin(); iter != m->end(); iter++) {
      resultset.push_back(std::multimap<double, float>());
      for (it = iter.value()->getTrackPoints().begin(); it != iter.value()->getTrackPoints().end(); it++) {
        resultset.back().insert(std::pair<double, float>(it->azimuth, it->speed));
      }                                              /* x */
    }                                                /* x */
    speed_max = -2000000;
    iter = m->begin();
    for (rsIt = resultset.begin(); rsIt != resultset.end(); rsIt++) {
      for (groupBy = (*rsIt).begin(); groupBy != (*rsIt).end();) {
        currentAz = (*groupBy).first;
        ret = (*rsIt).equal_range(currentAz);
        if (ret.first == ret.second)
          speed_max = ret.first->second;
        else {
          for (std::multimap<double, float>::iterator it = ret.first; it != ret.second; it++)
            speed_max = fn(speed_max, it->second);
        }                                            /* x */
        p = std::make_pair(currentAz, (QMap<QString, CTrack *>::iterator) iter);
        aggregate.insert(std::pair<float, std::pair<double, QMap<QString, CTrack*>::iterator > >(speed_max, p));
        speed_max = -2000000;
        groupBy = ret.second;
      }                                              /* x */
      iter++;
    }                                                /* x */
    std::cout << "name | descend | ascend | distance | "
              << "totaltime | totaltimemoving | azimuth | "   /* x */
              << "max(speed)" << std::endl;                   /* x */
    for (aggrIt = aggregate.begin(); aggrIt != aggregate.end(); aggrIt++) {
      speed_max = (*aggrIt).first;                   /* x */
      azimuth = (*aggrIt).second.first;              /* x */
      curTrack = (*aggrIt).second.second.value();    /* x */
      totalTimemoving = curTrack->getTotalTimeMoving();  /* x */
      totalTime = curTrack->getTotalTime();          /* x */
      totalDistance = curTrack->getTotalDistance();  /* x */
      ascend = curTrack->getAscend();                /* x */
      descend = curTrack->getDescend();              /* x */
      name = curTrack->getName().toStdString();      /* x */
      std::cout << name << " | " << descend << " | "  /* x */
                << ascend << " | " << totalDistance << " | " /* x */
                << totalTime << " | " << totalTimemoving << " | " /* x */
                << azimuth << " | "                  /* x */
                << speed_max << " | " << std::endl;  /* x */
    }                                                /* x */
}                                                    /* x */
```

Listing A.5: UC5

```
/* C++ lines of code: 49, x: excluded */
void cTrackDB_cpp_query2(CTrackDB *c) {
  QMap<QString, CTrack*>* m = (QMap<QString, CTrack *>*)&c->getTracks();          /* x */
  QMap<QString, CTrack*>:: iterator iter;
  QList<CTrack :: pt_t >:: iterator it;
  // heartRateBpm, elevation                                                      /* x */
  std :: vector<std :: multimap<int, std :: pair<float, float> > > resultset;
  // maxAvgspeed, elevation, heartRateBpm                                         /* x */
  std :: multimap<float, std :: pair<float, std :: pair<int, QMap<QString, CTrack*>:: iterator > > > aggregate;
  std :: vector<std :: multimap<int, std :: pair<float, float> > >:: iterator rsIt;
  std :: multimap<int, std :: pair<float, float> >:: iterator groupBy, groupByNext, groupByEnd, mIter;
  std :: pair<std :: multimap<int, std :: pair<float, float> >:: iterator,
              std :: multimap<int, std :: pair<float, float> >:: iterator > ret;
  float maxAvgspeed = -100.0, currentAvgSp, currentEle = 0.0;
  int currentHr = 0;
  std :: multimap<float, std :: pair<float, QMap<QString, CTrack*>:: iterator > > mapEle;
  std :: pair<std :: multimap<float, std :: pair<float, QMap<QString, CTrack*>:: iterator > >:: iterator,
          std :: multimap<float, std :: pair<float, QMap<QString, CTrack*>:: iterator > >:: iterator > ret2;
  std :: pair<float, std :: pair<float, QMap<QString, CTrack*>:: iterator > > pairEle;  /* x */
  std :: pair<float, QMap<QString, CTrack*>:: iterator > pairAvg;                 /* x */
  std :: pair<int, QMap<QString, CTrack*>:: iterator > pairHr;                    /* x */
  std :: multimap<float, std :: pair<float, QMap<QString, CTrack*>:: iterator > >:: iterator mapEleIt, groupBy2;
  double totalDistance, ascend, descend;                                          /* x */
  int totalTimemoving, totalTime, heartRate;                                      /* x */
  float ele;                                                                      /* x */
  std :: string name;                                                             /* x */
  for (int i = 0; i < 10; i++) {                                                  /* x */
    for (iter = m->begin(); iter != m->end(); iter++) {
      resultset.push_back(std :: multimap<int, std :: pair<float, float> >());
      for (it = iter.value()-> getTrackPoints (). begin (); it != iter.value()-> getTrackPoints (). end (); it ++) {
        resultset.back (). insert (std :: pair<int, std :: pair<float, float> >( it ->heartReateBpm,
          std :: make_pair( it ->ele, it ->avgspeed)));
      }                                                                           /* x */
    }                                                                             /* x */
    iter = m->begin();
    for (rsIt = resultset.begin (); rsIt != resultset.end (); rsIt ++) {
      for (mIter = (* rsIt ). begin (); mIter != (* rsIt ). end ();) {
        currentHr = mIter->first;                                                 /* x */
        currentAvgSp = mIter->second.second;                                      /* x */
        currentEle = mIter->second. first;                                        /* x */
        ret = (* rsIt ). equal_range(currentHr );
        if (ret. first == ret.second) {
          if (currentAvgSp > 0) {
            pairHr = std :: make_pair (currentHr, iter );                         /* x */
            pairEle = std :: make_pair ( currentEle , pairHr );                   /* x */
            aggregate. insert (std :: pair<float, std :: pair<float,
                std :: pair<int, QMap<QString, CTrack*>:: iterator > > >(currentAvgSp, pairEle ));
            maxAvgspeed = -100.0;
          }                                                                       /* x */
        } else {
          for (groupBy = ret. first ; groupBy != ret.second; groupBy++) {
            pairAvg = std :: make_pair (groupBy->second.second, (QMap<QString, CTrack *>:: iterator ) iter );  /* x */
            mapEle. insert (std :: pair<float, std :: pair<float, QMap<QString, CTrack*>:: iterator > >(groupBy->second.first,
pairAvg ));
          }                                                                       /* x */
          for (mapEleIt = mapEle.begin (); mapEleIt != mapEle.end ();) {
            currentEle = mapEleIt-> first;                                        /* x */
            ret2 = mapEle.equal_range( currentEle );
            if (ret2. first == ret2.second) {
              maxAvgspeed = mapEleIt->second.first;
            } else {
              for (groupBy2 = ret2. first ; groupBy2 != ret2.second; groupBy2++) {
                if (maxAvgspeed < groupBy2->second.first) {
                  maxAvgspeed = groupBy2->second.first;
                  currentEle = groupBy2->first;
                }                                                                 /* x */
              }                                                                   /* x */
            }                                                                     /* x */
```

Listing A.6: UC5 (continue)

```
            if (maxAvgspeed > 0) {
              pairHr = std :: make_pair (currentHr, iter );                    /* x */
              pairEle = std :: make_pair ( currentEle , pairHr );              /* x */
              aggregate . insert ( std :: pair< float , std :: pair< float ,
                    std :: pair< int , QMap<QString, CTrack*>:: iterator > > >(maxAvgspeed, pairEle ));
            }                                                                  /* x */
            mapEleIt = ret2 . second;
            maxAvgspeed = -100.0;
          }                                                                    /* x */
          mapEle.clear ();
        }                                                                      /* x */
        mIter = ret . second;
      }                                                                        /* x */
    }                                                                          /* x */
    std :: cout << "Aggregate size : " << aggregate . size () << std :: endl;  /* x */
    std :: cout << "name | descend | ascend | distance | "
                << "totaltime | totaltimemoving | "                          /* x */
                << "heartRateBpm | elevation | "                             /* x */
                << " max(avgspeed)" << std :: endl;                          /* x */
    std :: multimap<float, std :: pair< float , std :: pair< int , QMap<QString, CTrack*>:: iterator > > >:: reverse_iterator aggrRIt ;
    CTrack *currentTrack ;                                                     /* x */
    for ( aggrRIt = aggregate . rbegin (); aggrRIt != aggregate . rend (); aggrRIt ++) {
      maxAvgspeed = (*aggrRIt ). first ;                                       /* x */
      ele = (* aggrRIt ). second. first ;                                      /* x */
      heartRate = (* aggrRIt ). second.second. first ;                         /* x */
      currentTrack = (* aggrRIt ). second.second.second.value ();              /* x */
      totalTimemoving = currentTrack ->getTotalTimeMoving();                   /* x */
      totalTime = currentTrack ->getTotalTime ();                              /* x */
      totalDistance = currentTrack ->getTotalDistance ();                      /* x */
      ascend = currentTrack ->getAscend();                                     /* x */
      descend = currentTrack ->getDescend();                                   /* x */
      name = currentTrack ->getName().toStdString ();                          /* x */
      std :: cout << name << " | "
                  << descend << " | "                                          /* x */
                  << ascend << " | "                                           /* x */
                  << totalDistance << " | "                                    /* x */
                  << totalTime << " | "                                        /* x */
                  << totalTimemoving << " | "                                  /* x */
                  << heartRate << " | "                                        /* x */
                  << ele << " | "                                              /* x */
                  << maxAvgspeed << " | " << std :: endl;                      /* x */
                  }                                                            /* x */
     resultset . clear ();                                                     /* x */
     aggregate . clear ();                                                     /* x */
   }                                                                           /* x */
}                                                                              /* x */
```

Listing A.7: UC6

```
/* C++ lines of code: 23
 * x: excluded
 */
void cpp_query3(CTrackDB *c, CWptDB *w) {
  QMap<QString, CTrack*>* m = (QMap<QString, CTrack *> *)&c->getTracks();                    /* x */
  QMap<QString, CTrack*>:: iterator iter ;
  QList<CTrack :: pt_t >:: iterator it ;
  std :: vector<std :: multimap<std:: string , CTrack :: pt_t*> > resultsetTracks ;
  std :: vector<std :: multimap<std:: string , CTrack :: pt_t*> >:: iterator iterTr ;
  std :: multimap<std:: string , CTrack :: pt_t *>:: iterator iterTrP ;
  std :: vector<CWpt*>:: iterator iterP ;
  QMap<QString, CWpt*>* p = (QMap<QString, CWpt *> *)&w->getWpts();
  QMap<QString, CWpt*>::iterator iterWpts;
  std :: vector<CWpt*> resultsetWpts;
  for (int i = 0; i < 10;i++) {                                                              /* x */
    for (iter = m->begin(); iter != m->end();iter++) {
      resultsetTracks .push_back(std :: multimap<std:: string , CTrack :: pt_t *>());
      for (it = iter .value()-> getTrackPoints (). begin (); it != iter .value()-> getTrackPoints (). end (); it ++) {
        if (it ->ele > 20) {
          resultsetTracks .back (). insert (std :: pair<std :: string , CTrack :: pt_t*>( iter .value()->getName(). toStdString (), &*it ));
        }                                                                                    /* x */
      }                                                                                      /* x */
    }                                                                                        /* x */
    for (iterWpts = p->begin(); iterWpts != p->end();iterWpts++) {
      if (iterWpts.value()->ele > 20) {
        resultsetWpts .push_back(iterWpts.value ());
      }                                                                                      /* x */
    }                                                                                        /* x */
    std :: cout << "name | lon | lat | ele" << std :: endl;
    for (iterTr = resultsetTracks .begin (); iterTr != resultsetTracks .end (); iterTr ++) {
      for (iterTrP = (* iterTr ). begin (); iterTrP != (* iterTr ). end (); iterTrP ++) {
        std :: cout << (* iterTrP ). first << " | "
                    << (* iterTrP ). second->lon << " | "                                    /* x */
                    << (* iterTrP ). second->lat << " | "                                    /* x */
                    << (* iterTrP ). second->ele << " | "                                    /* x */
                    << std :: endl;                                                          /* x */
      }                                                                                      /* x */
    }                                                                                        /* x */
    for (iterP = resultsetWpts .begin (); iterP != resultsetWpts .end (); iterP ++) {
      std :: cout << (* iterP )->getName().toStdString () << " | "                           /* x */
                  << (* iterP )->lon << " | "                                                /* x */
                  << (* iterP )->lat << " | "                                                /* x */
                  << (* iterP )->ele << " | "                                                /* x */
                  << std :: endl;                                                            /* x */
    }                                                                                        /* x */
    resultsetTracks . clear ();                                                              /* x */
    resultsetWpts . clear ();                                                                /* x */
  }                                                                                          /* x */
}                                                                                            /* x */
```

A.1.3 CScout

Listing A.8: UC7

```
/* SQL lines of code: 17 */
SELECT IDS.NAME
FROM (
    SELECT EID,FID,COUNT(FOFFSET)
    FROM TOKENS
    GROUP BY EID,FID
) AS A
JOIN IDS
ON A.EID=IDS.EID
WHERE IDS.LSCOPE
AND IDS.ORDINARY
AND NOT IDS.READONLY
AND NOT IDS.FUN
AND NOT IDS.UNUSED
AND NOT IDS.CSCOPE
GROUP BY A.EID
HAVING COUNT(A.FID)=1
ORDER BY IDS.NAME;
```

Listing A.9: UC8

```
/* SQL lines of code: 13 */
SELECT DISTINCT NAME FROM (
        SELECT FID FROM (
                SELECT EID FROM IDS
                WHERE LSCOPE
                AND UNUSED
                AND NOT READONLY
                ) AS U
        LEFT JOIN TOKENS
        ON TOKENS.EID=U.EID
        ) AS UNUSED
LEFT JOIN FILES
ON FILES.FID=UNUSED.FID
ORDER BY NAME;
```

Listing A.10: UC9

```
/* SQL lines of code: 7 */
SELECT FUNCTIONS.NAME
FROM FUNCTIONS
JOIN FILES
ON FILES.FID=FUNCTIONS.FID
WHERE NOT RO
AND FANIN=0
ORDER BY FUNCTIONS.NAME;
```

A.2 Identify possible privilege escalation attacks with AWK

Listing A.11: Shell script that loops through /proc/pid directories.

```
#!/bin/sh

find /proc -maxdepth 1 -name '[0-9]*' |
while read -r line ; do
  ./ps_perm.awk "$line/status"
done
```

Listing A.12: AWK script that identifies possible privilege escalation attacks.

```
#!/bin/awk -f

BEGIN {
  su_in_group = 0;
  u_to_su = 0;
  cred_uid = -1;
  ecred_uid = -1;
  name = "N/A";
}
{
  if ($1~/Name:/)
    name = $2;
  if ($1~/Uid:/) {
    cred_uid = $2;
    ecred_euid = $5;
    if ((cred_uid > 0) && (ecred_euid == 0))
      u_to_su = 1;
  }
  if ($1~/Gid:/) {
    n = split ($0, array, " ");
    for (i = 1; i <= n; i++) {
      if ((array[i] == 4) || (array[i] == 27)) {
        su_in_group = 1;
        break;
      }
    }
  }
}
END {
  if ((su_in_group == 0) && (u_to_su == 1))
    printf ("User process %s has performed privilege escalation.",
            name);
}
```

A.3 Dynamic diagnostic tasks via Systemtap scripts

Listing A.13: Systemtap script that instruments *vfs_read()* and checks file permissions of the executing process to identify unauthorised read access to files. For our kernel it did not return any.

```
function get_task_ecred_fsuid(t) %{
  struct task_struct *task = (struct task_struct *)STAP_ARG_t;
  if (task->real_cred)
    STAP_RETVALUE = task->real_cred->fsuid;
  else
    STAP_RETVALUE = -1;
%}

function found_gid_in_group(fgid, t) %{
  struct task_struct *task = (struct task_struct *)STAP_ARG_t;
  struct group_info *gi = task->real_cred->group_info;
  int i = 0;
  int ret = 0;
  for (i = 0; i < gi->ngroups; i++) {
    if (STAP_ARG_fgid == gi->small_block[i]) {
      ret = 1;
      break;
    }
  }
  STAP_RETVALUE = ret;
%}

probe kernel.function("vfs_read")
{
  if (@defined($file->f_path->dentry)) {
    f_mode = $file->f_mode
    f_owner_euid = $file->f_owner->euid
    ecred_fsuid = get_task_ecred_fsuid(task_current())
    f_inode_mode = $file->f_path->dentry->d_inode->i_mode
    if (f_mode&1 && (f_owner_euid != ecred_fsuid || !f_inode_mode&400)
        && (found_gid_in_group($file->f_cred->egid, task_current())
            || !f_inode_mode&40)
        && !f_inode_mode&4)
      printf("In %s: %s(%d)\n",
             probefunc(), task_execname(task_current()),
             task_pid(task_current()))
  }
}
```

Listing A.14: Systemtap script that traverses the accounting list of processes and checks file permissions of each process to identify unauthorized read access to files. It returns a number of results.

```
function traverse_process_list (p) %{
  char line [100];
  struct task_struct *t;
  struct file *f;
  int bit ;
  rcu_read_lock ();
  list_for_each_entry_rcu (t,
      &((struct task_struct *)(long)STAP_ARG_p)->tasks, tasks) {
    struct group_info *gi = t->real_cred->group_info;
    for (f = files_fdtable (t->files)->fd[
          bit= find_first_bit ( files_fdtable ((unsigned long *)t->files)->open_fds,
            files_fdtable (t->files)->max_fds)];
         bit < files_fdtable (t->files)->max_fds;
         f = files_fdtable (t->files)->fd[
           bit= find_next_bit ( files_fdtable ((unsigned long *)t->files)->open_fds,
             files_fdtable (t->files)->max_fds, bit + 1)]) {
      if (f && f->f_path.dentry) {
        int f_mode = f->f_mode;
        int f_owner_euid = f->f_owner.euid;
        int ecred_fsuid = t->real_cred->fsuid;
        int f_inode_mode = f->f_path.dentry->d_inode->i_mode;
        const char *f_name = f->f_path.dentry->d_name.name;
        int i = 0, found_gid = 0;
        for (i = 0; i < gi->ngroups; i++) {
          if (f->f_cred->egid == gi->small_block[i]) {
            found_gid = 1;
            break;
          }
        }
        if ((f_mode&1) && ((f_owner_euid != ecred_fsuid)
                           || (!(f_inode_mode&400)))
            && ((!found_gid) || (!(f_inode_mode&40)))
            && (!(f_inode_mode&4))) {
          sprintf (line, "%s(%d): %s\n", t->comm, t->pid, f_name);
          strlcat (STAP_RETVALUE, line, MAXSTRINGLEN);
        }
      }
    }
  }
  rcu_read_unlock ();
%}

probe begin { printf ("%s\n", check_privileges_process_list (task_current ()))}
```

A.4 Diagnostic tasks with PiCO QL, Systemtap, and DTrace

Query description	PiCO QL query	SystemTap script	DTrace script

Pages brought in memory per process	`SELECT name, nr_ptes`
	`FROM Process_VT AS PS_VT`
	`JOIN EVirtualMem_VT AS VM_VT`
	`ON VM_VT.base = PS_VT.vm_id;`
		`global pgfaults`
		`probe vm.pagefault`
		`{`
		`  pgfaults[execname()]<<<1`
		`}`
			`vminfo:::pgin`
			`{`
			`  @[execname] = count();`
			`}`

Number of read system calls for “cc1” and “ruby” processes	`SELECT name, syscalls_read`
	`FROM Process_VT AS PS_VT`
	`JOIN EIO_VT`
	`ON EIO_VT.base = PS_VT.io_id`
	`WHERE name = ‘‘cc1”`
	`OR name = ‘‘ruby”;`
		`global reads`
		`probe syscall.read`
		`{`
		` if (execname() == ”cc1” ||`
		`  execname() == ”ruby”)`
		`  reads[execname()]<<<$count`
		`}`
			`syscall::read:entry`
			`/ execname == ”cc1” ||`
			`  execname == ”ruby” /`
			`{`
			`  @counts[execname] = count();`
			`}`

Selected MIB TCP statistics	`SELECT MibTcpActiveOpens,`
	`  MibTcpAttemptFails,`
	`  MibTcpCurrEstab`
	`FROM Statistics_VT;`
		`global stats`
		`probe tcpmib.ActiveOpens,`
		`  tcpmib.AttemptFails,`
		`  tcpmib.CurrEstab`
		`{`
		`  stats[pn()]<<<op`
		`}`
			`mib:::tcpActiveOpens,`
			`mib:::tcpAttemptFails,`
			`mib:::tcpCurrEstab`
			`{`
			`  @[probename] = count();`
			`}`

CPU system time per process	`SELECT name, stime`
	`FROM Process_VT;`
		`function traverse_process_list (p) %{`
		`  struct task_struct *c; char line[100];`
		`  rcu_read_lock();`
		`  list_for_each_entry_rcu(c,`
		`  &((struct task_struct *)(long)STAP_ARG_p)->tasks, tasks) {`
		`  sprintf(line, ”%s: %d\n”, c->comm, (int)c->stime);`
		`  strlcat(STAP_RETVALUE, line, MAXSTRINGLEN);`
		`  }`
		`  rcu_read_unlock();`
		`%}`
		`probe begin {printf(”%s\n”, traverse_process_list(task_current()))}`	

Table A.1: Analysis tasks using PiCO QL, SystemTap and DTrace

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983.

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, November 1997.

[ACFS94] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2):72–109, 1994.

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: a strongly-typed, interactive conceptual language. *ACM Trans. Database Syst.*, 10:230–260, June 1985.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *SIGMOD Rec.*, 25:68–75, December 1996.

[AG89] R. Agrawal and N. H. Gehani. Ode (object database and environment): the language and the data model. *SIGMOD Rec.*, 18(2):36–45, June 1989.

[AGO91] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 565–575, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[AH15] Mohammad R. Azadmanesh and Matthias Hauswirth. Sql for deep dynamic analysis? In *Proceedings of the 13th International Workshop on Dynamic Analysis*, WODA 2015, pages 2–7, New York, NY, USA, 2015. ACM.

[AKW87] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *VLDB JOURNAL*, 4:319–401, 1995.

[ANT92] Dennis A. Adams, R. Ryan Nelson, and Peter A. Todd. Perceived usefulness, ease of use, and usage of information technology: A replication. *MIS Q.*, 16(2):227–247, June 1992.

[BBE83] James M. Boyle, Kevin F. Bury, and R. James Evey. Two studies evaluating learning and use of qbe and sql. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 27(7):663–667, 1983.

[BCMV03] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. In *In Proc. IEEE*, pages 34–53. IEEE Press, 2003.

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric approach. In *Encyclopedia of Software Engineering*, volume 1, pages 528–532. Wiley, 1994.

[BFH+92] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Arc hitectures*, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association.

[BGI08] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[Bis04] Matt Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[BK06] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications, revised edition, November 2006.

[BKFP08] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 103–116, Berkeley, CA, USA, 2008. USENIX Association.

[Bou86] Roger J Bourdon. *The PICK Operating System: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BW05] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 262–286, Berlin, Heidelberg, 2005. Springer-Verlag.

[CAC+84] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent object management system. *Software: Practice and Experience*, 14(1):49–71, 1984.

[CB00] R. G. G. Cattell and D. K. Barry, editors. *The object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, pages 383–394, New York, NY, USA, 1994. ACM.

[CDRS86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the exodus extensible database system. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 91–100, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[Col89] Latha S. Colby. A recursive algebra and query optimization for nested relations. *SIGMOD Rec.*, 18(2):273–283, June 1989.

[CPJ06] Dave Crane, Eric Pascarello, and Darren James. *Ajax in action*. Manning, Greenwich, CT, 2006.

[Cro06] D. Crockford. The application/json media type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. [July 2013].

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[Dav89] Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Q.*, 13(3):319–340, September 1989.

[DD97] C. J. Date and Hugh Darwen. *A guide to the SQL standard (4th ed.): a user's guide to the standard database language SQL*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[DD06] M. Desnoyers and M. R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS)*, 2006.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens añd Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system, 1994.

[DeT05] John DeTreville. Making system configuration more declarative. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.

[DH00] Alan Dearle and David Hulse. Operating system support for persistent systems: Past, present and future. In *Software - Practice and Experience, Special Issue on Persisten t Object Systems*, 2000.

[DKM10] Alan Dearle, Graham N. C. Kirby, and Ron Morrison. Orthogonal persistence revisited. In *Proceedings of the Second international conference on Object data bases*, ICOODB'09, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[DL87] L. Deshpande and P.A. Larson. An algebra for nested relations. Technical report, University of Waterloo, 1987. Tech. Report CS-87-65.

[DLAR91] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *Computer*, 24:34–44, November 1991.

[DRH+92] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan, and K. Maciunas. An examination of operating system support for persistent object systems. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume i, pages 779 –789 vol.1, January 1992.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 273–284, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby programming language*. O'Reilly, first edition, 2008.

[FSL15] Marios Fragkoulis, Diomidis Spinellis, and Panos Louridas. An interactive SQL relational interface for querying main-memory data structures. *Computing*, 97(12):1141–1164, 2015.

[FSL16] Marios Fragkoulis, Diomidis Spinellis, and Panos Louridas. Pico ql: A software library for runtime interactive queries on program data. *SoftwareX*, 5:134 – 138, 2016.

[FSL19] Marios Fragkoulis, Diomidis Spinellis, and Panos Louridas. Live interactive queries to a software application's memory profile. *IET Software*, 13:241–248(7), August 2019.

[FSLB14] Marios Fragkoulis, Diomidis Spinellis, Panos Louridas, and Angelos Bilas. Relational access to unix kernel data structures. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 12:1–12:14, New York, NY, USA, 2014. ACM.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, December 1992.

[GOA05] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[GRADAD08] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: a declarative file system checker. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association.

[Gre99] R. Greer. Daytona and the fourth-generation language Cymbal. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 525–526, New York, NY, USA, 1999. ACM.

[Gre13] Brendan Gregg. *Systems performance: entreprise and the cloud*. Prentice Hall, Upper Saddle River, NJ, 2013.

[GSS+13] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. Cod: Database / operating system co-design. In *CIDR, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[HAF+11] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 38–49, New York, NY, USA, 2011. ACM.

[HD98] David Hulse and Alan Dearle. Trends in operating system design: Towards a customisable persiste nt micro-kernel. Technical report, University of Stirling, 1998.

[HJ91] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[HM85] Elie C. Harel and Ephraim R. McLean. The effects of using a nonprocedural computer language on programmer productivity. *MIS Q.*, 9(2):109–120, June 1985.

[I. 14] I. Molnar and A. van de Ven. Runtime locking correctness validator, 2014. Available online Current September 2014.

[IJC01] A. Iyengar, Shudong Jin, and J. Challenger. Efficient algorithms for persistent storage allocation. In *Mass Storage Systems and Technologies, 2001. MSS '01. Eighteenth IEEE Symposium on*, pages 85 –85, april 2001.

[Jos] Josef Weidendorfer. KCachegrind. Available online `http://kcachegrind.sourceforge.net/html/Home.html` Current January 2014.

[JV85] Matthias Jarke and Yannis Vassiliou. A framework for choosing a database query language. *ACM Comput. Surv.*, 17(3):313–340, September 1985.

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Trans. on Electronic Computers*, 11:223–235, April 1962.

[KJA93] Arthur M. Keller, Richard Jensen, and Shailesh Agarwal. Persistence software: bridging object-oriented programming and relational databases. *SIGMOD Rec.*, 22(2):523–528, June 1993.

[KKL+07] A. Kivity, Y Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[KKS92] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. *SIGMOD Rec.*, 21(2):393–402, June 1992.

[KL89] Michael Kifer and Georg Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *SIGMOD Rec.*, 18(2):134–146, June 1989.

[KS09] M. Keith and M. Schincariol. *Pro JPA 2: Mastering the Java Persistence API*. Apress, Berkely, CA, USA, 1st edition, 2009.

[LB02] S. Lampoudi and D. M. Beazley. SWILL: A simple embedded web server library. In Chris G. Demetriou, editor, *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 19–27. USENIX, 2002.

[LC86] Tobin J. Lehman and Michael J. Carey. Query processing in main memory database management systems. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 239–250, New York, NY, USA, 1986. ACM.

[LHS03] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Eng.*, 10(1):39–74, January 2003.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, October 1991.

[LSZE11] Kyu Hyung Lee, N. Sumner, Xiangyu Zhang, and P. Eugster. Unified debugging of distributed systems with Recon. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 85–96, June 2011.

[LTP14] LTP developers. The Linux Test Project test suite, 2014. Available online Current September 2014.

[Lut03] Mark Lutz. *Learning Python*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2003.

[M. 13] M. Fragkoulis. The PiCO QL Linux kernel module tutorial, 2013. Available online September 2014.

[MAB07] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 461–472, New York, NY, USA, 2007. ACM.

[Mai90] David Maier. Advances in database programming languages. In François Bancilhon and Peter Buneman, editors, *Papers from DBPL-1*, chapter Representing database programs as objects, pages 377–386. ACM Press, New York, NY, USA, 1990.

[MBB06] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[MDRK93] Bruce R. Millard, Partha Dasgupta, Sanjay Rao, and Ravindra Kuramkote. Run-time support and storage management for memory-mapped persistent objects. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 508–515, 1993.

[Mei11] E. Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, October 2011.

[Mem13] MemSQL. The MemSQL database, 2013. Available online http://memsql.com/ [July 2013].

[MGL+10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[MJLF86] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program, April 1986.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[Moo01] Richard J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 297–308, Berkeley, CA, USA, 2001. USENIX Association.

[Mos92] J.E.B. Moss. Working with persistent objects: to swizzle or not to swizzle. *Software Engineering, IEEE Transactions on*, 18(8):657 –673, August 1992.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. *SIGPLAN Not.*, 21(11):472–482, June 1986.

[Mun93] David S. Munro. On the integration of concurrency, distribution and persistence, 1993.

[Nat09] National Institute of Standards and Technology. CVE-2009-3290, National Vulnerability Database, 2009. September 22, 2009.

[Nat10] National Institute of Standards and Technology. CVE-2010-0309, National Vulnerability Database, 2010. January 12, 2010.

[OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the ObjectStore database system. *SIGMOD Rec.*, 21(2):403–412, June 1992.

[Oik13] Shuichi Oikawa. Integrating memory management with a file system on a non-volatile main memory system. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1589–1594, New York, NY, USA, 2013. ACM.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[Owe06] M. Owens. *The Definitive Guide to SQLite (Definitive Guide)*. Apress, Berkely, CA, USA, 2006.

[PBSL13] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[PCE+05] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[PFWA06] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[Pre00] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000.

[Ree00] G. Reese. *Database Programming with JDBC and Java*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2000.

[Rei81] Phyllis Reisner. Human factors studies of database query languages: A survey and assessment. *ACM Comput. Surv.*, 13(1):13–31, March 1981.

[RIS+10] Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward E. Aftandilian, and Samuel Z. Guyer. What can the gc compute efficiently?: A language for heap assertions at gc time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 256–269, New York, NY, USA, 2010. ACM.

[RKB85] M.A. Roth, H.F. Korth, and D.S. Batory. Extended algebra and calculus for non-1nf relational databases. Technical report, University of Texas at Austin, 1985. Tech. Report TR-85-19.

[RKB87] Mark A. Roth, Henry F. Korth, and Don S. Batory. SQL/NF: A query language for non-1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[RPV09] S. N. Tirumala Rao, E. V. Prasad, and N. B. Venkateswarlu. Performance Evaluation of Memmory Mapped Files with Data Mining Algorithms. *International Journal of Information Technology and Knowledge Management*, 2(2):365–370, Jul-Dec 2009.

[RSI12] Christoph Reichenbach, Yannis Smaragdakis, and Neil Immerman. Pql: A purely-declarative java extension for parallel programming. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 53–78, Berlin, Heidelberg, 2012. Springer-Verlag.

[Rum87] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 466–481, New York, NY, USA, 1987. ACM.

[San98] R. E. Sanders. *ODBC 3.5 Developer's Guide*. McGraw-Hill Professional, 1998.

[SBRP11] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 119–132, New York, NY, USA, 2011. ACM.

[SC05] Michael Stonebraker and Ugur Cetintemel. "One size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[SDSD86] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[SKS06] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[SKS11] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 6 edition, 2011.

[SN05] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[Spi10a] Diomidis Spinellis. CScout: A refactoring browser for C. *Sci. Comput. Program.*, 75(4):216–231, April 2010.

[Spi10b] Diomidis Spinellis. Farewell to disks. *IEEE Softw.*, 27(6):82–83, November 2010.

[SS86] H J Schek and M H Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, April 1986.

[SS96] Richard M. Stallman and Cygnus Support. *Debugging with GDB : The GNU source-level debugger, GDB version 4.16*. Free software foundation, Boston, MA, 1996.

[ST06] Michal Spivak and Sivan Toledo. Storing a persistent transactional object heap on flash memory. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, LCTES '06, pages 22–33, New York, NY, USA, 2006. ACM.

[SZ90] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. In *proc. of the Persistent Object Systems Workshop*, pages 89–102, 1990.

[SZ10] D. Spiewak and T. Zhao. ScalaQL: Language-integrated database queries for Scala. In *Proceedings of the Second International Conference on Software Language Engineering*, SLE '09, pages 154–163, Berlin, Heidelberg, 2010. Springer-Verlag.

[TC03] C. Tunstall and G. Cole. *Developing WMI solutions: a guide to Windows management instrumentation*. The Addison-Wesley Microsoft Technology Series. Addison-Wesley, 2003.

[TF86] S.J. Thomas and P.C. Fischer. Nested relational structures. *Advances in Computing Research III, The theory of databases, P.C. Kanellakis, ed*, pages 269–307, 1986.

[Tha86] Satish M. Thatte. Persistent memory: a storage architecture for object-oriented database systems. In *Proceedings on the 1986 international workshop on Object-oriented database systems*, OODS '86, pages 148–159, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[Thea] The Perl Archive Network. The DBD::CSV driver. Available online `http://search.cpan.org/ hmbrand/DBD-CSV-0.41/lib/DBD/CSV.pm` Current January 2014.

[Theb] The Wikipedia community. List of software categories. Available online `https://en.wikipedia.org/wiki/List_of_software_categories` Current July 2015.

[The13] The SQLite team. The virtual table mechanism of SQLite, 2013. Available online `http://www.sqlite.org/vtab.html` [July 2013].

[The14a] The Facebook team. osquery:SQL powered operating system instrumentation and analytics, 2014. Available online Current October 2014.

[The14b] The Phoronix developers. The Phoronix Test Suite, 2014. Available online Current September 2014.

[The14c] The SQLite team. The SQLite Query Planner Release 3.8.0., 2014. Available online Current September 2014.

[TM99] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 117–130, Berkeley, CA, USA, 1999. USENIX Association.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[VD92] Francis Vaughan and Alan Dearle. Supporting large persistent stores using conventional hardware. In *In Proc. 5th International Workshop on Persistent Object Systems*, pages 34–53. Springer-Verlag, 1992.

[VD00] Viswanath Venkatesh and Fred D. Davis. A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Manage. Sci.*, 46(2):186–204, February 2000.

[Ven00] Viswanath Venkatesh. Determinants of perceived ease of use: Integrating control, intrinsic motivation, and emotion into the technology acceptance model. *Info. Sys. Research*, 11(4):342–365, December 2000.

[Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000.

[WPN06] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for Java. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP '06, pages 28–49, Berlin, Heidelberg, 2006. Springer-Verlag.

[WS81] Charles Welty and David W. Stemple. Human factors comparison of a procedural and a nonprocedural query language. *ACM Trans. Database Syst.*, 6(4):626–649, December 1981.

[YKC06] Yoshisato Yanagisawa, Kenichi Kourai, and Shigeru Chiba. A dynamic aspect-oriented system for OS kernels. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 69–78, New York, NY, USA, 2006. ACM.

[YS93] M. Y.-M. Yen and R. W. Scamell. A human factors experimental comparison of sql and qbe. *IEEE Trans. Softw. Eng.*, 19(4):390–409, April 1993.