Universitat Politècnica de Catalunya

Master Thesis

SMILE: Towards a Data Analytics System for Low-Resource Edge Devices

Author:
Ferran Olivera Corpas

Advisors:
Josep Lluís Larriba Pey
Arnau Prat Pérez

A thesis submitted in fulfillment of the requirements for the degree of Master in Innovation and Research in Informatics

October 2018

Abstract

Modern databases are intended for being run in machines with high performance capabilities such as servers. However, there is a set of non-traditional and smaller equipment that could be benefited in terms of decentralization of processes, more autonomous behavior, better bandwidth and latency decrease if they could manage their own lightweigth database, instead of managing requests to an external one. This is the case of, for instance, fog devices [23], smartphones or simple boards like a Raspberry Pi [24].

The lack of a simple database management system that can be easily adapted to the needs of both high-end computers and the aforementioned group of devices motivates our research. In the long term, we plan to deliver such a system, but, due to time constraints, in this work we focus on implementing what should be the core of our database: its buffer pool.

Hence, in this thesis, we review the state-of-the-art for buffer pool designs and, then, we describe the architecture followed by our model and its main optimizations. Finally, we provide an extensive evaluation of our work, testing it in two different environments – a server and a small board – with different buffer pool configurations.

Acknowledgements

First of all, I would like to deeply thank my master thesis advisors Josep Lluís Larriba Pey and Arnau Prat Pérez for their direction and advices. They have assisted me during the whole research and this work could not have been completed without their help.

I would also like to extend my gratitude to my entire family and closest friends for all the support received during these years.

Finally, I would like to acknowledge HP Inc. for financially supporting my master's degree and for offering me the opportunity to work in a place with such diversity which keeps me motivated to continue learning new things every day.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this chapter, we cover the motivation behind our research and the main contributions resulting from it. Moreover, we summarize our objectives and state how this document is structured.

1.1 Motivation

Modern databases are designed and intended for being used on top of high-end machines with lots of processing power and memory as, for instance, servers. Nevertheless, there is a set of non-traditional and smaller appliances that currently manage requests to an external database (DB) and that could be benefited from running its own one. This would allow for decentralization of processes, more autonomous behavior, latency decrease and better bandwidth. Some examples of this kind of devices are smartphones, simple boards like a Raspberry Pi and fog devices in general

This last set of devices come from the concept of Fog Computing. In such paradigm – also known as Edge Computing –, communication, computation and storage resources distribution are done on or close to systems in the control of end-users [2, 11]. As one could expect, this implies having a three hierarchy as the one depicted in Figure 1.1, where Cloud, Fog and Mobile represent the three different existing layers in a top-down vision of the paradigm.

The Internet of Things (IoT) is a common application example of Fog Computing and consists in a network of physical devices with some kind of connectivity which are also embedded with software and electronics such as sensors or actuators. The design of these devices allows to collect and share data from several sources with ease, resulting in a scalable model used to increase efficiency, create better experiences or contribute with new solutions

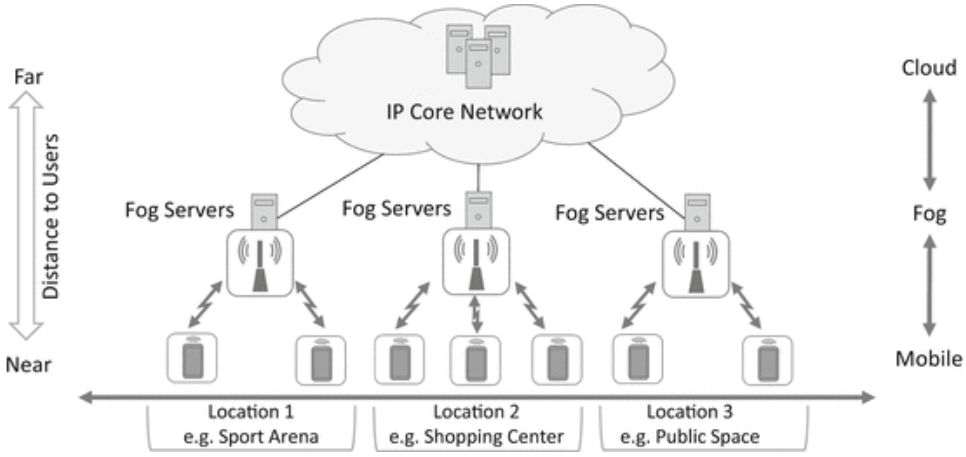

Figure 1.1: Fog Computing architecture [22].

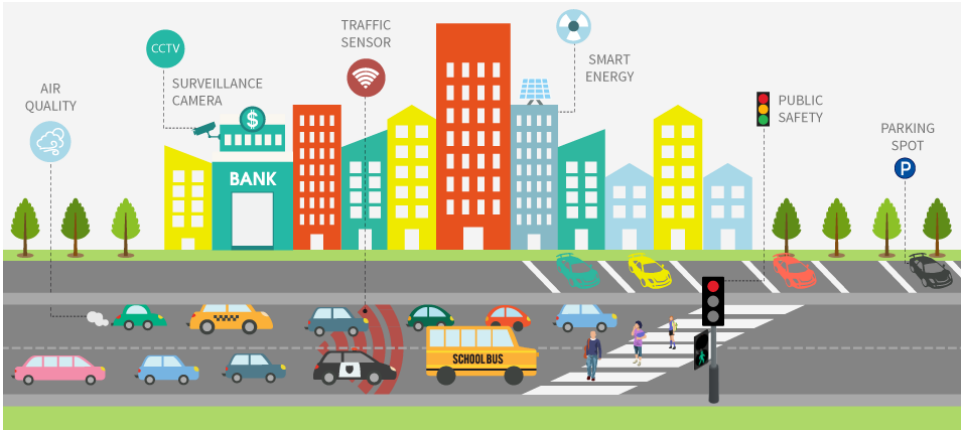

Figure 1.2: IoT use cases in a smart city [14].

for multiple real-world problems or situations [1, 19]. In that sense, home automation, wearable fitness trackers or autonomous cars are just some examples of what this technology can grant us. The field, in fact spans many industries as domotics, transports, enhanced learning and many more (Figure 1.2).

1.1.1 Databases in Edge Devices

Some edge devices may require of constant database query to retrieve updated information, log own behavioral data, etc. Typically, those queries are issued by a device and send over the Internet until it reaches one of its respective datacenters and is then interpreted, executed and a response is given back to the client [7]. As a result of this process, a non-negligible

latency between queries and limited bandwith exist. Moreover, external factors to the device like the saturation level of the server can even increase this delay. Due to this, the possibility of moving the responsibility of the query execution from centralized datacenters to the fog or to the device itself by running a mobile database is quite appealing. Nevertheless, the architectures of both kinds of systems present important differences, especially at processing and memory levels where traditional datacenters are equiped with poweful processors and make use of Hard Disk Drives (HDD) to keep data, whereas commodity hardware runs low-power CPUs and Solid State Drives (SSD) as persistent memory.

It is quite important to notice that edge devices are not only less powerful than servers, but that usually represent simple devices such a mini-computer that may be exposed to critical situations. For instance, think of a Raspberry Pi placed in a strategic location in a forest with a couple of sensors to measure the temperature and the humidity of the environment in order to track its variations during the year and powered by a small battery charged by means of a naive solar panel. Such a device could suffer from adverse weather conditions and/or be long time periods with no human intervention, all which could leave it without battery or even ruin it. In this kind of device, being able to run a database system to maintain information secure from power failures and allow to decouple it from the need of Internet connection to send periodically the measures to a server, would be really beneficial. Overall, the characteristics of the fog devices which could benefit from an adapted database are the following.

- Simple low-power multicore processors, frequently using an ARM architecture.
- Limited memory resources.
- Use of Flash NAND-based techonology as persistent storage.
- Unstable energy sources or adverse climatic conditions.

Whilst there has been some previous work on the effects of SSDs in enterprise database applications [16], concurrency in databases [21] or even different DB data-storage paradigms [18], the concept of fog-adapted databases has not been really developed yet. Furthermore, there is still a lack of feasibility studies about it that help discern the main pros and cons so that this DB technology can be conveniently evaluated. Due to this and the existing needs concerning edge devices presented, our research revolves around the possibility of developing SMILE, a light database able to run in a fog/mobile system that can benefit, especially, from improved bandwidth and little latency, as well as in traditional DB devices, such as a normal computer. In the long term, we plan to deliver such a system, but, due to time constraints, we

will be focusing this work on implementing what should be the core of our database: its buffer and storage managers.

1.2 Contributions

The major contribution of this study is the description and analysis of a buffer pool intended to be part of a lightweight database suited for mobile/fog devices as long as regular computers. As covered in section 1.1, adapted on-disk databases could grant to a set of fog devices the ability to work with autonomy, lesser latencies and better bandwidth and enable new applications. The buffer pool is an essential part of a DB. It consists in a software system that is used for caching table and index data pages as they are modified or read from disk. Thus, its primary purpose is to reduce database file I/O and improve the response time for data retrieval. We also develop a disk-subsystem in charge of representing the needed interface between the buffer pool and the computer's filesystem.

We conduct an extensive evaluation in order to understand which is the most adient buffer pool condiguration and which differences exist between particular systems. We design our buffer pool keeping in mind the possibility of adjusting it by setting some parameters depending on our needs or the resources that are available. On the one hand, the user can specify the size of page to be used by the DB, this is: the basic unit of data that the system is able read or write. The bigger it is, the less reads need to be issued to scan some amount of data, however, it also implies having to load more memory which may not be useful in case of executing random accesses. As always, discovering the correct trade-off is a key point. On the other hand, buffer pool size can be defined too. In this case, having as much as possible is something preferable. Nonetheless, good performance with limited memory is a must for edge equipment. Finally, we also provide some optimizations in form of partitioning, NUMA-awareness and a prefetcher that can be enabled or not and configured at convenience. Each one can grant relevant speedups depending on the workload type and the underlying machine.

All this factors are considered during the experiments and the diverse outcomes obtained are compared. An important aspect here is that our work is tested in two dispare environments. The first one is a regular server equipped with HDD and high-end processors and the other is a low-consumption ARM device using SSD as primary storage. This way, we have access to a clear view of how our database performs when used as an ordinary one versus when used in a fog ecosystem. This aims to help discerning the feasibility of runnning lightweight databases on edge devices. Moreover, the server results allow us to make an idea of the possibilities of our design in the future, when

an environment similar to that of a server can be found on more portable devices like the ones we target.

In order to validate the correct functioning of our design, we give several unit tests and some others aiming for representing realistic high-level operators that stress the buffer pool with different workloads and that force it to serve requests from multiple threads simultaneously. Group By is one of them and is often used to group the result-set of a query by one or more columns. Hash Join is also part of our operations and is a type of join algorithm which finds, for each distinct value of the join attribute, the set of tuples in each relation which have it. The latter one is a Breadth-First Search (BFS) algorithm, which is used for traversing or searching tree or graph data structures, starting from some arbitrary node and exploring all the neighbors at the present depth before moving on to the nodes at the next depth level. We also define tests that hugely stress the DB in order to profile the buffer pool operations and be able to perform a good evaluation of it.

Last but not least, we freely deliver the buffer pool's code so that everyone can check it, use it or expand it with new features if desired. A little contribution but that may be useful for future testing or research.

1.3 Objectives

The main goals of this master thesis are the following:

- Depict the buffer pool's architecture of a state-of-the-art database.
- Develop an operative lightweight DB with its own buffer pool, storage layer and the ability to issue background tasks to a thread-pool. Also, implement unit tests to check validity and some complex tests to be able to stress the system.
- Analyze the performance of the designed database when using different configuration parameters.
- Evaluate the behavior of the DB when run on top of a mobile-like device compared to when it is used in regular server with high computing power.
- Summarize the results obtained and propose next steps to guide future research.

1.4 Structure

This document is divided in several chapters, each one covering different aspects of the work done. Thereby, the global structure of the thesis is as follows:

- In chapter 2, it is presented current state-of-the-art database architecture, its key technology features and significant concepts related to this work. This serves as the starting point for our contributions.
- In chapter 3, we describe the implemented DB architecture, discuss the most important design decisions and give a general overview of how it operates. Special emphasis is put on explaining the memory management performed by the buffer pool, this is: page allocation, pin and unpin operations and its internal structures. In another vein, data persistence, some introduced optimizations and other mechanisms needed for a correct behavior of the system are conveniently detailed.
- In chapter 4, we reveal which experiments have been done in order to evaluate the performance of our database when run in distinct systems and the influence of using different sets of configuration parameters. The analysis is carried out by stressing the system with workloads tailored and implemented specifically for it.
- In chapter 5, we highlight how future work could help to expand and/or complement this research.
- In chapter 6, we finally deliver our conclusions regarding the work done and the results obtained in the evaluation detailed in chapter 4.
- Finally, in chapter 7, we list a bibliography with all the references that have been consulted during the whole research process.

Chapter 2

Background

In this section we review the state of the art for modern buffer pools and Database Management Systems (DBMS) in general. A key performance factor for databases is how memory management is produced, especially how the buffer manager operates data transfers between main memory and the persistent storage. For the sake of explaining how it works in detail, we will primarily focus on the efficient implementation found in PostgreSQL. PostgreSQL – also refered just as Postgres – is one of the most advanced open source object-relational DBMS available [20] and with great standards compliance. As a DBMS, it can handle from small applications workloads to high-end applications with many concurrent users. Moreover, we will also take a closer look to the NUMA-awareness, thread and fiber concepts and see how are these related with our work.

2.1 Architecture of a DBMS

In this section, we give an overview of the traditional architecture of a DBMS. The whole description will be guided with figure 2.1, where single boxes represent system components and double boxes in-memory data structures. Furthermore, solid lines stand for control and data flow, whilst dashed lines indicate data flow only.

2.1.1 DDL Commands and Query Processing

Generally, in databases, the information is kept in the secondary storage and it is loaded into main memory when any operation on it must be performed. The storage manager controls storage on disk and keeps track of the location of files. In this way, it is able to obtain the pages containing a given file

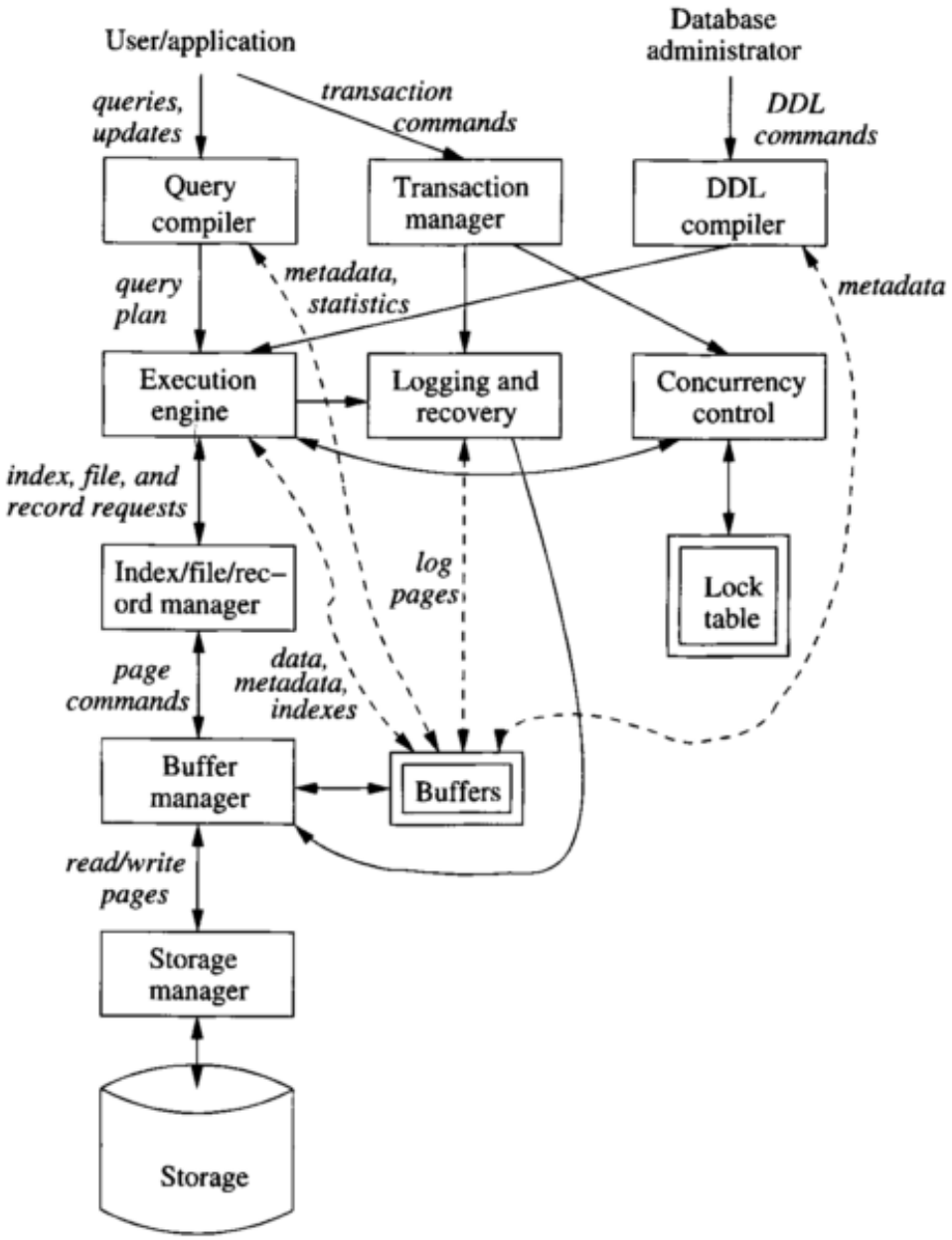

Figure 2.1: Overview of the architecture of a DBMS [7].

requested by the buffer manager. The buffer manager takes care of dividing the main memory into several buffers of page size, so that queries can operate directly with them.

In order to operate or manage this data, multiple DBMS commands exists. Overall, we can classify such commands into two differentiated groups:

- Applications or regular users asking for data or modifying it.
- The DB administrator in charge of the schema of the database.

Administrator commands are the ones that create, modify or delete the structure or constraint information relative to any table or relation, data which is all part of the schema of the database. These schema-altering data-definition language (DDL) commands are parsed by a DDL processor and send to the execution engine, which – going through the index/file/record manager – updates the database schema information. This process can be seen in the path starting in the top right of figure 2.1.

However, the major part of the commands take the path on the top left side of the figure. When the user or application starts any transaction using the data-manipulation language (DML). Such commands do not alter the schema but can consult or modify database content. DML commands are managed by two subsystems.

- **Query Answering**. A query is always parsed and optimized by a query compiler and the resulting actions go to the execution engine, which takes care of requesting the data to the buffer manager. The later loads into the main-memory buffers the requested information from disk via the storage manager, which manages pages as the basic unit of transfer.
- **Transaction Processing**. Queries are grouped into atomical units – named transactions – whose effect must be granted to be preserved upon completion. This is achieved thanks to the concurrency-control and recovery managers.

2.1.2 Transaction and Query Processors

As explained before, database operations are normally grouped into transactions, which are executed isolated from other transactions. Moreover, they have guaranteed that, if they complete, their work will be durable, that is, it will not be lost. The transaction manager takes transaction commands from client applications, which inform it about the begin and end of transactions and works alongside other transaction processor's components to accomplish the following tasks.

- **Logging**. Database changes are logged in buffers following a well-defined policiy and these are written to disk at a given time interval. Thus, in case of a system crash, the recovery manager can check such change-log and be able to restore the system to a consistent state.
- **Concurrency control**. The concurrency-control manager helps to

make sure that all transactions are performed in an order such that its effect is the same as if they were executed one after the other. This is usually achieved by maintaining database locks that prevent two transactions from accessing the same piece of data simultaneously.

- **Deadlock resolution**. Since transactions compete for resources between them, a deadlock situation can occur sometimes. In such cases, the transaction manager is able to cancel the necessary transactions to unlock the processing of the rest.

The last remaining part of a DBMS is the query processor, which have a great impact on the database overall performance. It is basically implemented by two components.

- **Query compiler**. It is in charge of translating queries into sequences of operations, which tend to be relational algebra operators. Typically, the compiler is divided into a pipeline formed by a parser, a preprocessor and an optimizer.
- **Execution engine**. It is responsible of executing each one of the operations specified by the compiler, interacting with the buffers and other aforementioned DB components to allow logging all changes, locking the data that needs to be accessed, etc.

2.2 Buffer Manager Architecture

As has already been explaineed, databases store information in persistent storage devices such as HDDs or SSDs. However, data is cached in main memory to speed up the process just like how our operating system does. Since databases are complex systems, they themselves implement mechanisms to optimize this procedure and be as efficient as possible without neglecting performance. This task is developed by the already introduced buffer manager, which is composite by three different layers.

- **Buffer pool**. It is basically an array divided in chunks where each slot is used to store data from a file page. The set of array indices are referred as buffer IDs.
- **Buffer descriptors**. Is an array of buffer descriptors with the same number of elements as buffer pool slots exist. Each descriptor can correspond to one of those buffer pool slots at a time and hold special metadata.
- **Buffer table**. Is a hash table that holds the translation from page IDs – the identification value of a file page stored in disk – to the buffer IDs of the buffer slots that have such pages loaded.

Next, we describe with detail the implementation of these structures in the case of PostgreSQL.

2.2.1 Buffer Pool

The buffer pool is a region of memory where data from disk is actually cached. It can be interpreted as a long memory section divided in hundreds, if not thousands, entries of page size called buffers and identified by a corresponding buffer ID. When schema elements such as tables or indexes are loaded from the persistent memory devices, their data file pages are stored in one or more buffers – depending on the buffer size (normally 4-8 KB) as well as their own size.

2.2.2 Buffer Descriptors

This layer is an array of very important metadata that defines the global status of the buffer pool. At each array position we can find a single buffer descriptor structure holding data related to its corresponding buffer pool slot. Although this kind of structure contains a myriad of fields, we next describe only the most relevant ones [25].

- **Empty bit**. A bit representing whether the corresponding buffer pool slot is empty – no page is cached, so information in this descriptor can be obviated – or not.
- **Pinned bit**. A bit representing whether the page cached by the corresponding buffer pool slot is currently being accessed by a PostgreSQL process or not.
- **Tag**. Buffer tag of the page stored in the corresponding buffer pool slot.
- **Buffer ID**. Identification value of the buffer pool slot currently associated to the descriptor.
- **Reference count**. Number of processes currently accessing the associated page. When it reaches zero, the page is automatically unpinned.
- **Usage count**. Number of times the corresponding buffer pool slot has been accessed since it was cached.
- **Dirty bit**. A bit indicating if the associated page is dirty – has been modified in main memory but not updated in disk – or not.
- **Valid bit**. A bit indicating if the associated page is valid – a non-valid page can not be read or written– or not.

- **I/O in progress bit**. A bit indicating that the associated page is being either read from storage or written to storage.
- **Free next**. A pointer to the next free descriptor. When Postgres starts, all descriptors are empty and linked forming a list of free descriptors (freelist). Then, as long as pages are requested, descriptors are retrieved from the freelist and associated to pages.

2.2.3 Buffer Table

The buffer table consists of a hash function, the hash bucket slots and multiple entries. In Postgres, each data file page is assigned a unique label named buffer tag, which is the code used to request the buffer manager an exact page. The hash function is in charge of mapping those buffer tags to the hash bucket slots. Collisions are resolved by means of linked lists which store the entries mapped to the same bucket slot. Each entry in the table holds two values: the buffer tag of a page and the buffer ID of the descriptor holding the page's metadata.

A light-weight lock called BufMappingLock is used to restrict the access to the buffer table. When a process is looking for a given buffer tag in the table, inserting a new entry or deleting an existing one, it must acquire the BufMappingLock. This lock can be divided into multiple partitions to reduce the table contention when several processes are operating simultaneously. Thereby, each BufMappingLock partition is in charge of a portion of the hash bucket slots.

2.2.4 Buffer Replacement Policies

The correct definition of which block must be thrown out of the buffer pool when an empty buffer is needed, is a critical design choice. This behavior is given by the buffer-replacement strategy that the system uses. In fact, some of these are familiar from other application types or operating system scheduling policies. In this section we take a closer look to the most important ones.

2.2.4.1 Least Recently Used (LRU)

The LRU rule states that the thrown out block must be the one that has not been accessed for the longest time. This technique requires the buffer manager to keep an updated table showing the time of last access of each buffer in the system. Note that this forces each database access to log its time and destination buffer in such table, incurring in a significant maintenance

effort. In any case, LRU is a quite effective approach because buffers with long time since its last access are less likely to be reclaimed in a near future than those recently cached.

2.2.4.2 Firs In First Out (FIFO)

FIFO policy says that, under a new buffer request, the buffer returned should be the one that has been more time occupied by a previous page. The data needed for this policy is easier to compute than in LRU. Here, the buffer manager only needs to know the time at which current pages were loaded into each one of the buffers. In order to do so, a table can be used with an entry per each buffer that is updated when a block is read from disk and stored in it. So, there is no need to modify the table per each access, making FIFO more painless to maintain than LRU. However, it also comes with some disadvantages: we could have a block that is frequently used and, eventually, it would become the oldest block in the buffer pool; forcing to write it back to disk and reread it in the next request, adding serious overhead to the process.

2.2.4.3 Clock Algorithm

Also known as Second Chance algorithm, it is a common efficient approximation to LRU [7]. A general implementation of this method can be as follows.

- Imagine all the buffers arranged in a circle-style fashion and a clock hand pointing to one of them – as shown in Figure 2.2 – that rotates clockwise if it needs to find a buffer to place a disk page.
- Each buffer has an associated flag that can be set to 0 or 1. Buffers marked with a 0 are susceptible to be replaced, whilst buffers with a 1 are not.
- When a buffer is accessed or contents of a page are cached to it, its correspondingflag is set to 1.
- When looking for a buffer to load new content, the buffer manager scans for the first 0 it can find. Each buffer flagged as 1 visited while moving clockwise during this process is set to 0. This action is repeated until the clock hand reaches a zero-marked buffer, which is then chosen for eviction.

This is the method actually implemented by PostgreSQL, however it presents slightly differences, since, as explained before, the presented technique is just a generic view of the clock algorithm. More specifically, instead of tagging

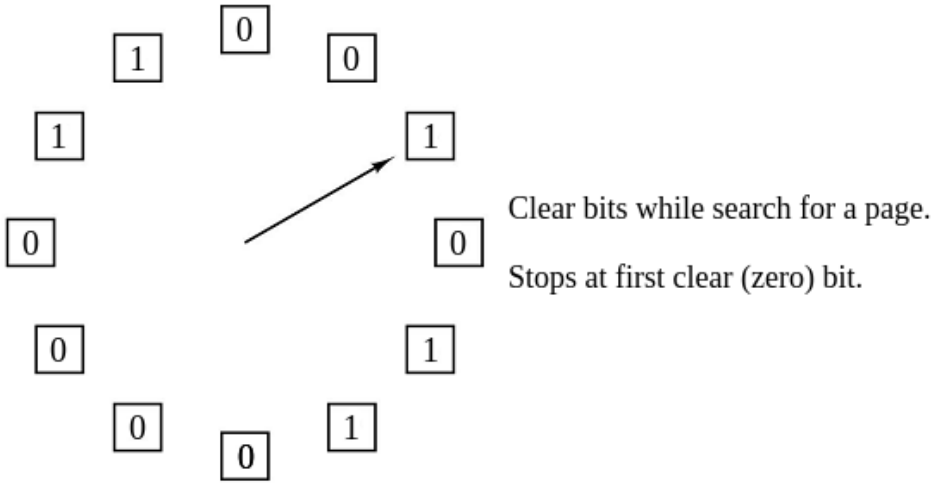

Figure 2.2: Clock algorithm example.

Algorithm 1 PostgreSQL implementation of the clock algorithm

```
1: while true do
2:     candidate ← buffer descriptor pointed by nextVictimBuffer
3:     if candidate is unpinned then
4:         if candidate's usage_count = 0 then
5:             return candidate's buffer_id
6:         else
7:             Decrease candidate's usage_count by 1
8:     Advance nextVictimBuffer
```

buffers with 0's and 1's, PostgreSQL takes profit of the pinned/unpinned and usage count information available in the buffer descriptors to do so. Algorithm 1 summarizes the pseudocode that is used. Hence, PostgreSQL maintains a pointer called *nextVictimBuffer* to the next buffer to evaluate and is used to check whether it is pinned or not. In case it is, the *nextVictimBuffer* is advanced to the next position beacuse we do not want to evict pages currently being used by a process. Contrary, if it is unpinned we also check if its usage count has reached zero and, if so, we return its buffer ID to free it. Notice that, if the usage count is greater than zero, it is decremented and the *nextVictimBuffer* is advanced to the next candidate to allow keeping recently accessed pages.

Although the basic implementation of the clock algorithm is quite good yet, in the literature we can find some variations of it. Examples of this are the WSclock [4] or the CLOCK-Pro [12]. For instance, in the later, everytime a page is accessed, it is assigned a reuse distance, which is the number of

other different pages requested since its last access. Such reuse distance is used to categorize them as cold or hot pages, depending of whether its reuse distance is big or small respectively. All pages are placed into a single list, ordered by recency. Then, cold pages are granted a test period and, if they are re-accessed during it, they become a hot page. When a page must be evicted, a cold one is chosen.

2.2.5 Running Example

In order to give an overview of how the different parts of the buffer manager interact, in this section we present an example of the behavior of a pin operation where all buffer slots are already occupied but the requested page is not stored. Following, we describe the steps of the process and number them, so that the explanation matches the descriptive pictures found in Figures 2.3 and 2.4.

1. Suppose that a backend process requests a page whose buffer tag is Tag_M. So, the system looks up the buffer tag in the buffer table, but, as said, we assume that it can not be found.
2. Since all buffer slots are being used, a buffer slot must be chosen as victim to be evicted by means of the clock algorithm. In this case, the algorithm says that the victim must be the buffer descriptor number 5, which is associated with the buffer tag Tag_F and the buffer slot 5.
3. If the dirty flag of the victim's buffer descriptor is set, the page must be written back to storage before loading the new one.
4. The partition lock that covers the slot containing the old entry is acquired.
5. The new table entry formed by the requested buffer tag – Tag_M – and the victim's buffer ID is generated. Then, the partition lock that covers the slot corresponding to the new entry is acquired and the entry is inserted into the buffer table.
6. The old entry from the buffer table is deleted and its partition lock released.
7. The requested page is loaded from storage to the victim's buffer slot. After, the flags of the buffer descriptor are updated with the current information (buffer ID = 5, dirty = 0, etc.).
8. The partition lock corresponding to the new entry is released.

9. The backend process can now access the loaded data directly from the buffer pool slot with buffer ID = 5.

2.3 NUMA Aware Allocation and Execution

Modern multiprocessor computers tend to use a memory design that is called Non-Uniform Memory Access (NUMA), where the access time depends on the relative location of the memory respect to the processor. In other words, a processor can access its local memory faster than the local memory of another processor. Figure 2.5 exemplifies how the NUMA architecture of a computer with eight CPUs could look like. In that simple case, processors are equally grouped into two sockets which are interconnected. Each socket has its own memory controller and its CPUs are able to access its 32 GB of local memory quite fast. However, in case a CPU would need to read or modify some data stored in the memory under the other socket, a remote access would be performed, which is, by design, significantly slower since some communication between both NUMA nodes is required.

Basically, modern computers are becoming a kind of network, since the cost of accessing specific data depends on which chip are located both the information itself and the running thread. Thus, parallelization in multi-core systems needs to take into account the different cache and main memory hierarchies. More specifically, RAM's division into several NUMA nodes has to be considered to make sure that threads work on NUMA-local data.

A state-of-the-art approach for databases against such issues is the known as *morsel-driven* query execution [17]. Under this model, query processing takes small portions of input data (morsels) and schedules them to worker threads that run entire operator pipelines until a pipeline breaker. Moreover, the parallelism degree can be adjusted elastically, so the dispatcher can adjust resources dynamically to fit arriving queries in the workload (work-stealing). Moreover, the dispatcher tries to respect the NUMA-locality of the morsels data, so executions tend to be NUMA-aware, thus better performance is achieved.

Since we plan to make SMILE NUMA-aware, this type of proposals are really interesting and will be possibly adapted and implemented in the future. In section 3.3.1.5 we depict the already done adjustments in SMILE towards an efficient NUMA management.

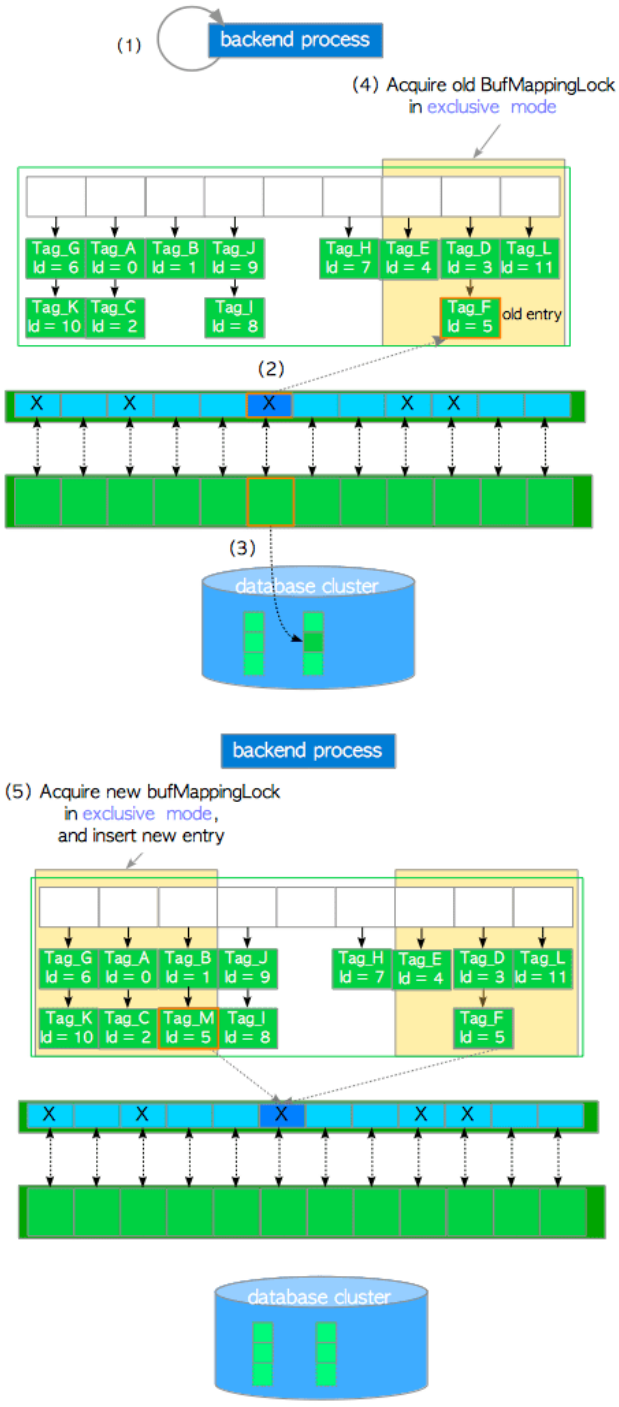

Figure 2.3: Pinning a page example [25].

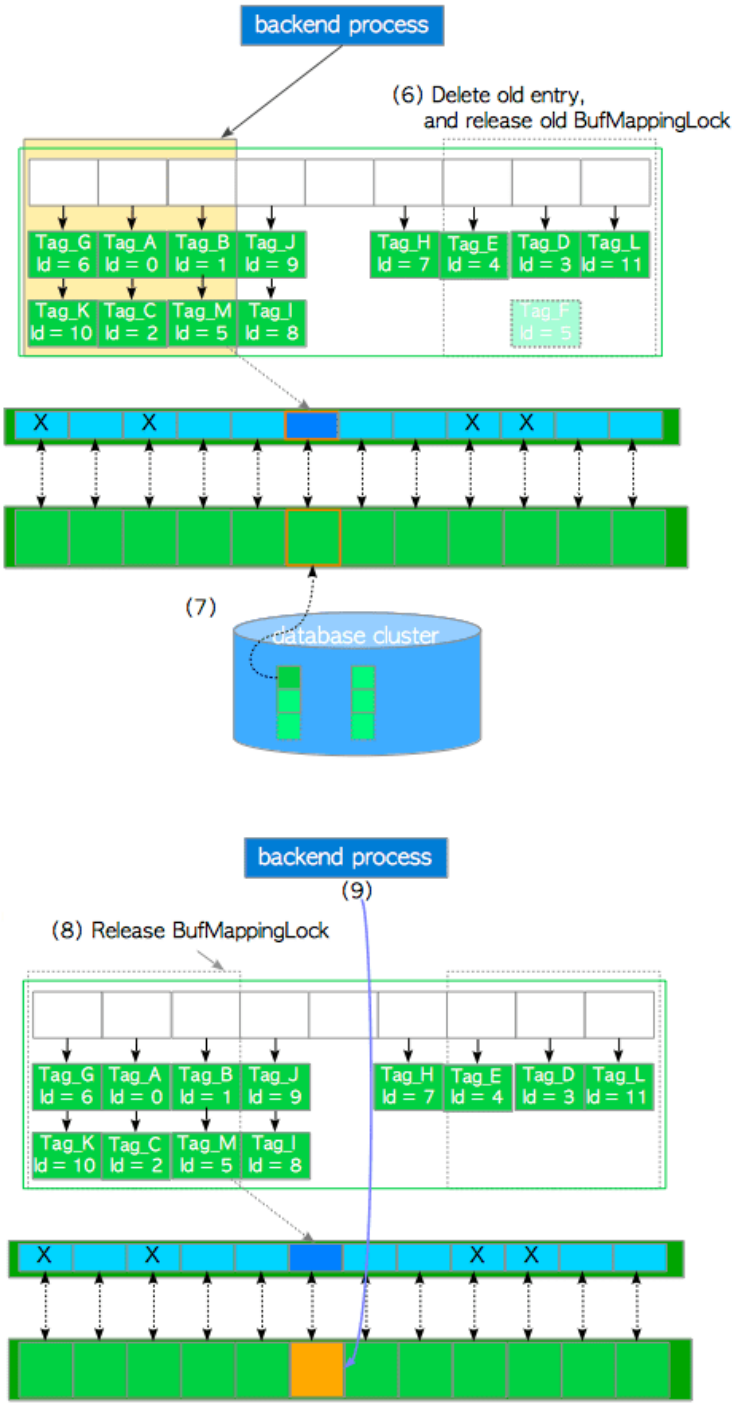

Figure 2.4: Pinning a page example (continued from Figure 2.3) [25].

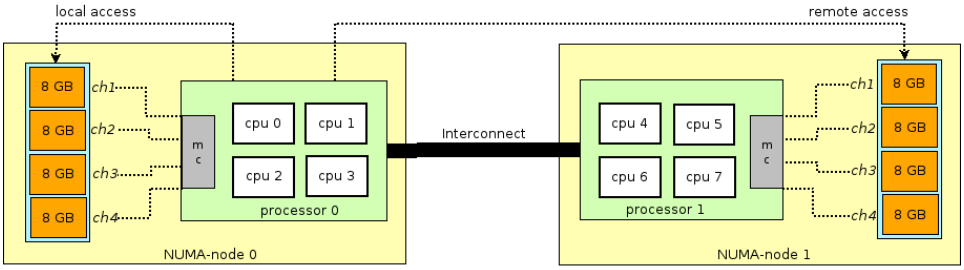

Figure 2.5: Example of a 2-node NUMA architecture with 8 CPUs [3].

2.4 Threads and Fibers

Threads consist in nothing else but a program counter indicating the next instruction to execute from a instruction sequence and a stack to store the needed variables. OS threads share address space and the majority of kernel's resources, which makes them lighter processes able to share data between them. Nevertheless, since they do not share CPU registers and have own stacks, they can execute different control flows concurrently.

The scheduler is in charge of swaping in and out of CPUs the running threads. Such preemption is typically performed when the thread has consumed its CPU quantum or when the code explicitly traps. In these cases, the CPU switches from user to kernel mode and transfers execution to the corresponding handler, which stores the CPU registers, the thread stack pointer and the program counter into a Thread Control Block. After so, the scheduler can pick the next thread to schedule and the kernel restores all its saved information, switches back to user mode and resumes the execution. Overall, threads allow for concurrently performing several operations, however switching between them – thus constantly changing between user and kernel mode – incurs in heavy performance penalty due to the unavoidable TLB flushes, cache misses and CPU pipeline stalls.

Fibers can be seen as a solution to cope with cases where the kernel may suppose a concurrency bottleneck. Indeed, fibers (or lightweight threads) are threads but, instead of being implemented by the OS, they are implemented in user mode [6]. Hence, while threads are slow to synchronize and to spawn, fibers are lightweight and can be synchronized with little overhead. Furthermore, fibers may use a specifically designed scheduler that is more appropriate for their use-case. So, they normally make use of non-preemptive schedulers and follow their execution path until they voluntarily yield.

Because of all this, fibers are well-suited to serve transactions, making them

quite useful for database operations. In the future, SMILE is expected to serve all buffer pool requests by means of fibers, allowing for asynchronous I/O and multiple simultaneous operations at a low context switching overhead.

Chapter 3

SMILE Architecture

The motivations depicted in section 1.1 have encouraged us to create a database, written in *C++*, suited for both traditional computers and small mobile/IoT devices. Since a DB is a big and complex software piece and we start from scratch, we determined to focus in a first implementation of the core of the database, the buffer manager. Thus, in this chapter we describe the design of SMILE – our database –, including the architecture of the buffer pool and the storage layers, as well as some of its features and basic operations supported.

3.1 Architecture Overview

Before describing at a low level the implementation and operations of each layer of our system, we want to provide a basic architecture schema of the current design used by SMILE. This aims to support and help understanding the rest of the definitions given in this chapter and is presented in figure 3.1.

SMILE has currently two well defined layers: the storage and the buffer manager. The first one is in charge of managing disk accesses, which arrive from the other layer in form of requests. Hence, data is kept safe into a single file called the data file, whilst configuration-specific information is placed into the metadata file. Overall, this is the least complex of the two.

The buffer manager layer is, indeed, the most relevant one and manages the data that is cached into main memory and serves the requests that are done by other processes. It contains the buffer slots where data is loaded into, a descriptor for each buffer holding metadata information and several data structures to track the current pages and buffers in use and how are these related. Furthermore, it also has a special structure named allocation table

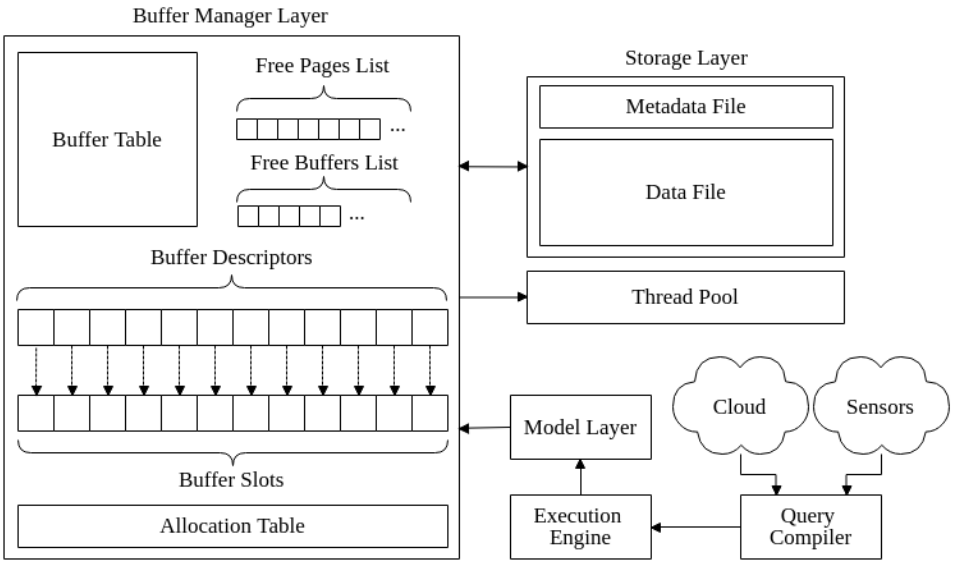

Figure 3.1: SMILE's architecture overview.

to know whether a disk page is being used or not, which is stored into disk upon checkpointing or powering off to be able to recover it.

SMILE also presents a thread pool that makes use of fibers with the objective of executing all the operations that are requested to the buffer manager, allowing to deliver asynchronous I/O and making the database to have better performance. Actually, the thread pool is partly-implemented and it is only used with prefetching purposes. Thence, since the it is still in development and has not much impact in the current design of SMILE, we have opted to skip describing it now and postpone its description until section 5.2, under the Future Work chapter. Other kind of features such as the possibility of recovering from an interrupted transaction are expected to be done in the future too.

In sake of giving a general vision of how the whole system would interact at a basic level, there are other layers named in the presented schema although they are not implemented yet. These are: the query compiler, the execution engine and the model layer. The task of the first two has already been explained in section 2.1, whilst the later would be used to decouple data managing from its internal representation in the buffer pool or the storage.

The idea is that this architecture serves to build up the database for small devices that we are proposing. Then, sensors and any other source of data could be connected with the buffer pool in the wake of storing and updating information. Other processes, such as an external cloud system, could

execute requests to SMILE to analyze or process all the gathered knowledge present in the device. In the next sections we give a deeper description of the storage and buffer manager layers.

3.2 Storage Layer

The storage layer is the one that works at the lowest level of abstraction. At its basics, it takes care of managing all the file input and output requests performed by the buffer pool. In this way, the buffer manager layer can delegate the implementation of the storage and focus in handling the access to the buffers and some other higher level tasks. This section will cover the principal operations available in this plane, however, prior to that, we need to explain the principal storage structures.

3.2.1 Data and Configuration Files

Clients constantly request information to the database, which, as explained in section 2.2, is always retrieved from a buffer in main memory. Still, main memory is limited – in our context – and volatile forcing to store all knowledge in a persistent place. In order to do so, we make use of files stored in the system which are managed by using the `std::fstream` class. Essentially, our model employs two distinct files: the configuration file and the data file.

On the one hand, the **metadata file** is the smallest one and its function is to hold the configuration parameters with which the database has been initialized. These parameters could just be assigned to variables, nevertheless, notice that since a DB can be powered off, it must somehow know its configuration details after a restart. Otherwise, the system could end up with a serious failure or a data corruption. The easiest solution to this situation is to keep that into a specific file. Currently, the page size is the only setting that is being stored, but our application is parameterized enough for allowing to increase the number of the configuration fields with no extra cost.

On the other hand, the **data file** is the biggest one and actually contains all the information of the database. The file is first created empty and as long as the buffer manager requests free space to the storage layer, its lenght is incremented by a fixed amount corresponding to the aforementioned page size used to configure the system. So, the the total length of the data file is logically divided into several pages, the basic unit of data that is managed between layers. Each page is assigned a page identifier (*pageID*) in progressive order and starting from zero, which is used to identify and be able to request them. Thus, the idea is to operate data at page level

using a write back policy [15]: provisioning main memory buffers with that exact amount of information and updating the data file pages only when their analogous cached page in the buffer pool is dirty (has been modified) and being evicted.

3.2.2 Operations

The objective of the storage is, as explained, to decouple the file management from the upper layers. To allow that, we provide an interface of simple methods that perform very specific actions, but that are enough for implementing all the data administration needed. We next mention and briefly describe each one of these operations.

- **Create**. This method is used when building up a full new database. It first generates the data and metadata files in a specified directoy path, initializes all the variables and finally reserves a page in the metadata file – which should be the first one – and writes into it all the system parameters to backup them as explained in section 3.2.1.
- **Open**. This operation is performed when initializing a database that has been previously closed, which should happen when powering on the system. Given a path, it checks that metadata and a data files exist and opens them. It also loads the storage configuration parameters by reading them from the config file and initializes all the variables.
- **Close**. It acts as the counterpart of the previous operation by closing both the data and the config files and checking that no errors are raised. It is requested upon powering off the database.
- **Reserve**. This utility is called by the buffer manager when a new page is needed. It increases the size of the data file in a given number of pages and returns the page identifier of the first allocated page in this process.
- **Read**. This method is employed to read a page from the data file and identified by the *pageID* indicated in the request into a memory address also specified. The buffer manager itself takes care that such address corresponds to the buffer pool slot where the page needs to be placed.
- **Write**. Given a *pageID* and a pointer to some data – usually a buffer pool slot containing a dirty page being replaced –, it writes a whole page from the pointed data to the corresponding data file page.
- **Size**. Returns the current size of the data file in terms of number of allocated pages. This information is needed to implement some

procedures in the buffer manager layer.

3.3 Buffer Manager Layer

The buffer manager provides database processes with the information they need to operate. It makes use of the storage interface defined in section 3.2 to simplify disk access and load data to its buffer pool in main memory. In this section, we first deal with the design and structures used in our implementation and later we focus in its accessible methods. In order to correctly understand the descriptions that are detailed in this passage, we encourage you to read, at least, chapter 2 first.

3.3.1 Design and Main Structures

Similarly to the storage layer, the buffer manager has its own parameters that can be adjusted as the situation requires and that may greatly influence the performance of the whole system. Such configuration options, alongside the different structures needed to correctly operate, suppose the main aspects to understand our implementation and are explained in this section.

3.3.1.1 Configuration Parameters

The buffer manager makes use of some configuration details in order to define how it must behave under certain circumstances. This data is introduced when creating or powering on the database and, like in the storage subsystem, our design allows to expand such set of options – if needed – in future development. However, we are currently using just three parameters which work as explained next.

- **Pool size**. Represents the total amount of memory, specified in KB, that will be used as buffer pool. The more memory is destinated to the pool, the more buffers the system will dispose, leading to more and better disk caching and better performance. There is only one limitation, the pool size must always be multiple of the page size used to configure the storage interface. This is done to guarantee that there is an integer number of buffers and avoid issues.
- **Prefetching degree**. It has to do with the prefetching feature of our database. Indicates the number of consecutive pages to prefetch. For instance: if it is set to 3, after requesting page X the system will prefetch pages $X + 1$, $X + 2$ and $X + 3$. More information regarding prefetching can be found at section 3.3.1.6.

- **Number of partitions**. It stands for the amount of portions in which the buffer pool structures are divided. This particularity is depicted in section 3.3.1.4.

3.3.1.2 Buffer Table and Descriptors

As explained in section 2.2, the buffer table's objective is to map the ID of a page with the ID of the buffer pool slot holding it. Whilst the PostgreSQL version of the table is quite complex, we have opted for a simpler implementation using a `std::unordered_map` where each entry is indexed by a page ID and contains a buffer ID, which are the types managed by our database. The set of page ID values ranges from zero to the number of pages in the data file minus one, representing each one of the existing pages in crescent order. Likewise, there are as much buffer IDs as slots in the buffer pool and are named in increasing form too. Each time the buffer manager caches a page, a new entry is added to the buffer table to track the buffer where it is loaded. When the buffer slot is released, the entry is also removed from the table, thus allowing the system to know which data is exactly in main memory at any given point in time.

The buffer table, combined with the buffer descriptors, signify the greatest source of information about the status of pages in database. As in Postgres, descriptors in our database are arranged in form of an array – they are held in a `std::vector` – and each one keeps metadata related to one single buffer pool slot. The fields that can be found in our descriptors are the following.

- **In use bit**. A bit representing whether the corresponding buffer pool slot is being used or not. If no page is cached in the slot, the information in this descriptor can be obviated and/or the slot can be reused in a future request.
- **Dirty bit**. A bit indicating if the associated page is dirty – has been modified in main memory but not updated in disk – or not.
- **Reference count**. Number of processes currently having a reference to the associated page. When the counter reaches zero, it means that the corresponding page is completely unpinned.
- **Usage count**. Number of times the stored page has been accessed since it was loaded in its current buffer slot.
- **Page ID**. Represents the page ID of the page currently cached in the corresponding buffer pool slot.

Field	Description
Buffer	A pointer to the buffer pool slot containing the requested page.
Buffer ID	The buffer ID identifying the slot containing the requested page.
Page ID	The page ID of the requested page.

Table 3.1: Description of the buffer handler fields.

- **Buffer pointer**. Pointer to the buffer pool slot associated with the descriptor.
- **Content lock**. It is a `std::unique_ptr<std::mutex>` used to protect the metadata held by the descriptor from simultaneous updates. It is acquired before any modification and released after so.

3.3.1.3 Buffer Pool

The buffer pool is the memory region where the database caches data from disk. In our case, its length depends on the pool size parameter previously introduced and it is managed in small portions of a fixed size (buffer pool slots) which is defined in the set of configuration parameters of the storage layer – section 3.2.1.

A process may ask the database for an empty page (to store new data) or specify a valid page ID for demanding an already existing one (to read or modify previous data). Those operations are respectively named alloc and pin and are described in section 3.3.2. In both cases, after a successful request, the client process is returned a special structure called **buffer handler** containing the fields detailed in table 3.1. By using these handlers, processes can access data directly using the Buffer field and ask again for the same page in the future thanks to the Page ID parameter – notice that a pin operation knows which page is requesting, but, when performing an alloc, the process does not know the ID of the page that will be granted. Finally, the Buffer ID is not used at all for managing the system, nonetheless, since we are still developing the database, it is helpful for debugging operations.

If there are no more empty slots when performing an alloc or a pin of a page which is not already loaded in the buffer pool, one slot must be chosen for eviction and its cached page must be written into disk if it has been modified. This selection procedure is done using always the same replacement policy: the clock algorithm, which has been previously introduced in section 2.2.4. Hence, our replacement strategy is the same one used in PostgreSQL, which is not difficult to implement since the buffer descriptors give us the necessary information to make the choice.

Notice that all the operations related with the buffer manager need to know constantly the status of disk pages and buffer pool slots, especially whether they are free or are being used. Having to collect that knowledge each time would be very time-consuming and would slow-down the system's performance. To cope with that, we have created two simple auxiliar structures but that really boost the operation time. They are the **free pages list** and the **free buffers list**. The first one is initialized with all the empty pages IDs found during startup and is updated progressively as requests are issued. The second one is filled also during system initialization with the IDs of all the buffer slots available and shrinks as long as the buffer manager caches data. When the buffer pool is full, the free buffers list is empty which makes easy to know when to launch a slot replacement.

3.3.1.4 Partitioning

Remembering the definition of our buffer decriptors, each one has an own lock to isolate simultaneous operations over it. However, we have not stated any measure against such multi-threading issues regarding the buffer table, the free pages lists and the free buffers lists. The problem with these structures is that, since its information is consulted or modified in almost all operations, if we make use of mutexes to protect them, a lot of contention will be generated when dealing with multiple processes demanding information to the database. Here is where the concept of partitions appear.

A buffer pool partition is a structure containing its own buffer table, free buffers/pages lists and a lock that we implement with `std::mutex`. At system's initialization, as many partitions as indicated in the configuration parameters are created. From this point on, each page will be using always the same partition for any kind of need. More specifically, pages are assigned to partitions by a simple hash consisting in the operation: $pID \mod N$, where pID stands for the page ID and N for the number of partitions in the database. In the same way, each buffer slot will be tied to a partition following the rule: $bID \mod N$, where bID is the ID of the buffer.

Figure 3.2 naively represents the structure of a buffer pool with 4 partitions and X buffer slots. In such case, for instance, the first partition – number 0 – would be the only one whose free pages list and buffer table contain entries that satisfy: $pID \mod N = 0$. For the free buffer list entries, would apply: $bID \mod N = 0$. Moreover, before accessing any structure, its associated **partition lock** is taken, which prevents data race problems.

With this approach, we avoid the aforementioned contention problem, since the lock collision probability of two processes is inversely proportional to the total number of partitions in the system. Both in the partition and de-

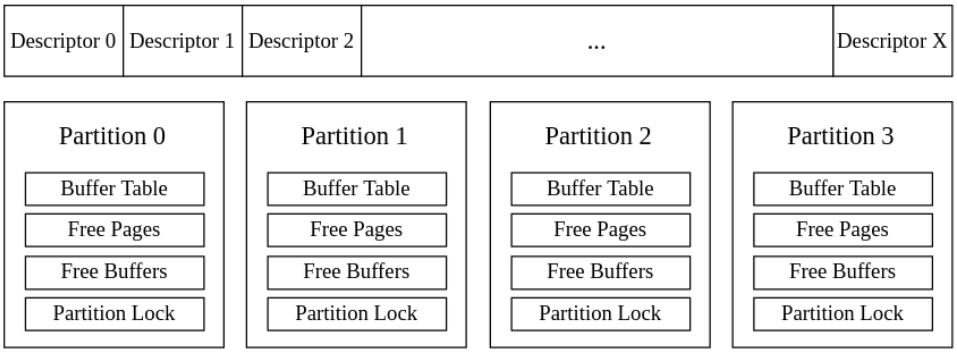

Figure 3.2: Example of a buffer pool with 4 partitions.

scriptor locks cases, we follow a RAII (Resource Acquisition Is Initialization) approach when dealing with them – implemented with `std::unique_lock` or `std::shared_lock` as corresponds –, which makes the code less error-prone. Thanks to partitions and the division of buffer descriptors, our system is able to serve several requests at the same time.

3.3.1.5 NUMA Awareness

As seen in 3.3.1.4, the system is prepared to attend several processes simultaneously. Nevertheless, there could still exist a limitation that has not been mentioned. In a naive implementation, the buffer slots – where pages are fetched – could consist in a large array allocated consecutively in memory. This could create some scalability problems. For instance, following previous figure 2.5 example, the array of buffer slots could be placed in some memory region under the scope of the NUMA node 0. Now imagine a workload that would just use CPU 0 for reading or writing information from the database. Instead, if CPUs 0 and 1 are used, performance could ideally be doubled. Likewise, using CPUs 0-3 could speedup the execution up to 4 times faster. Nonetheless, making use of CPUs 0-7 could not grant us an 8X performance, which is due to the fact that CPUs 4-7 would start generating overhead for each database access since an extra communication effort would be needed to access the buffer pool slots under NUMA node 0.

In order to cope with that, we have followed a NUMA aware design that can be enabled or disabled just with a compilation flag. Thence, when it is activated and by making use of the `numa.h` library, the number of available NUMA nodes is detected and the buffer slots array is divided in as many chunks and each one is allocated into a different node. After so, buffer IDs are then assigned in a circular fashion between the NUMA nodes. If we reason a little bit the configuration parameters of the buffer pool, we can

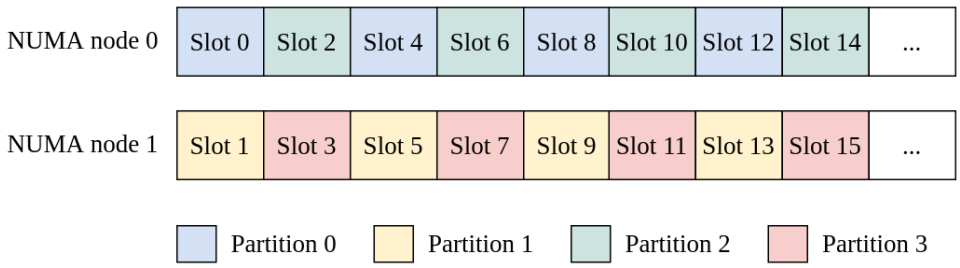

Figure 3.3: Buffer slots distribution with 2 NUMA nodes and 4 partitions.

also easily achieve that buffer slots relative to a same partition are placed into the same NUMA node. Figure 3.3 shows how would be distributed the first 16 buffer slots of a buffer pool with 4 partitions running into a machine with 2 NUMA nodes. This feature helps us to overcome the NUMA communication overhead and be able to scale better in the aforementioned cases because we can adjust the workload so that CPUs center their requests into pages that are from partitions that are managed in their own NUMA node.

3.3.1.6 Page Prefetcher

We have included a **prefetching mechanism** to speedup how the database operates at least in some cases. Although we plan to deliver a more complex one as future work, we have started by defining a basic next page(s) prefetcher. Thus, the idea is that when a page pin is issued, a secondary thread can be designated for loading the next consecutive page(s) into the buffer pool without blocking the main thread execution. In this way, if the same or another process tries to pin such next page, it will already be in main memory and the time required for reading it from disk will have been overlapped with the execution of other tasks. Obviously, this benefits are limited to workloads with sequential access patterns that retrieve information with no randomness.

Instead of creating dynamically the prefetching threads, prefetches are issued by means of the thread pool that is instantiated before running the database. Then, when the system needs to prefetch some data, it appends a pin request for the needed page into the task queue of one of those threads, which, when free, starts executing the pin. In this first implementation of the buffer pool, prefetches are assigned to threads in a circular way. With this, we try to balance the workload among the worker threads.

The way prefetches work can be adjusted or even disabled thanks to a configuration parameter explained before in this section: the prefetching degree.

Such parameter denotes the number of next consecutives pages to prefetch (in case it is set to zero no prefetching is done).

3.3.1.7 Allocation Table

The last part that we are presenting regarding the buffer manager is the allocation table. As explained in 3.2.1, the data file keeps the database pages in disk. It starts with no pages and, as long as alloc operations are performed, it grows a page at a time. Nevertheless, data is not always kept indefinetely and pages may be set as released, meaning that can be assigned in a future alloc request instead of reserving new space in disk. Thence, we need some place to store information about whether which pages are currently allocated and which not, this is respectively: which ones keep relevant data and which others are marked for being reused to allocate new values. This is the goal of the allocation table.

The table is implemented by using a `boost::dynamic_bitset`, which is an array containing as much bits as number of pages are in the data file and where each element tells us if the respective page is allocated (1) or not (0). That array is allocated in main memory when the database is powered on, updated accordingly when allocating or releasing pages and finally saved when closing the system. Notice that this information is filled at runtime and that if it is not correctly stored into disk, it could not be restored after a power off which would cause the loss of the allocation table and the system would not be able to determine whether pages are being used or not. So, the buffer manager takes profit of the data file and reuses some of its pages to store the table. These pages are named *protected pages* and are not allowed to be pinned by any external process in order to keep the integrity of the data.

That knowledge must be stored following a particular pattern so that the system can modify and recover it easily. Particularly, our approach is shown in figure 3.4. Supposing a 4 KB page size, page zero of the data file would store the allocation bits of the first 32768 pages (4 KB $\rightarrow$ 32768 bits), including itself. Once all the pages were in use and a new allocation was issued, page 32768 would be reserved by the database to store the status of pages between 32768 and 65536 (both ends included) and page 32769 would be returned to the client process requesting the new page. This pattern is followed indefinitely as long as new petitions are done and allows the buffer manager to know beforehand where to check the allocation table.

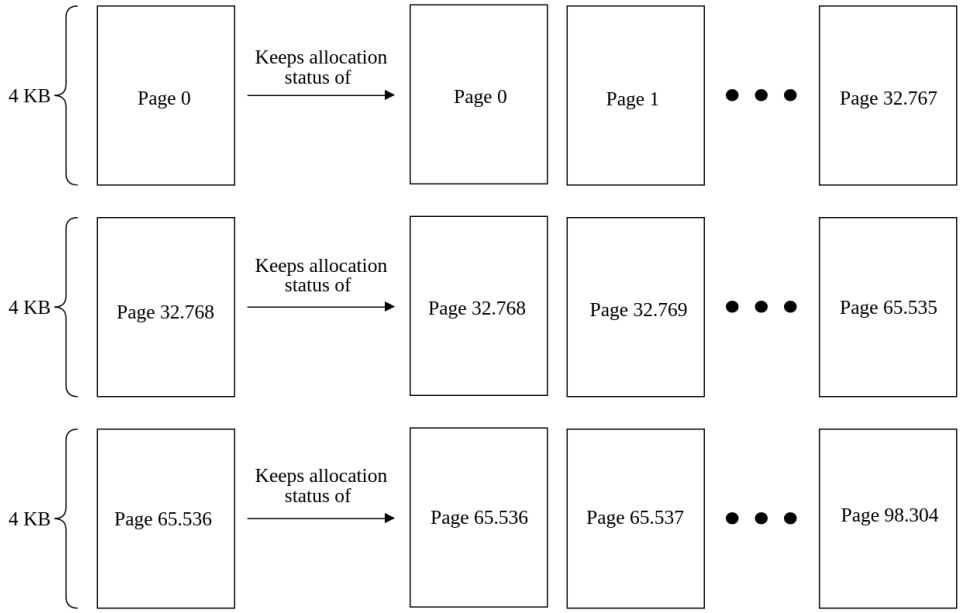

Figure 3.4: Pattern used to store the allocation table in disk.

3.3.2 Operations

The interface of the buffer manager needs to provide enough operations to allow a process to request pages, more disk space, etc. However, it also must support procedures to safely power on or off the system, test the correctness of the data or obtain general usage information. Following, we detail the available actions that can be requested to our buffer pool.

- **Create**. This operation requires specifying the configuration parameters to be used for both the buffer manager (sec. 3.3.1.1) and the storage (sec. 3.2.1) and a file path to the place where to store the data and metadata files. With that information, a new database is generated and initialized, including the buffer manager and the storage layers. In this step, all the structures presented in section 3.3.1 are created and initialized. Moreover, enough main memory is allocated into the different NUMA nodes to build the buffer slots array as depicted in 3.3.1.5.
- **Open**. In this case, the configuration parameters of the buffer manager and a file path where a database is kept are needed. The software powers on the DB found in the provided path and initializes the system similarly as done in the create operation.
- **Close**. It is the operation expected to be peformed to shutdown the system with no issues. First, all the dirty buffer slots are written back

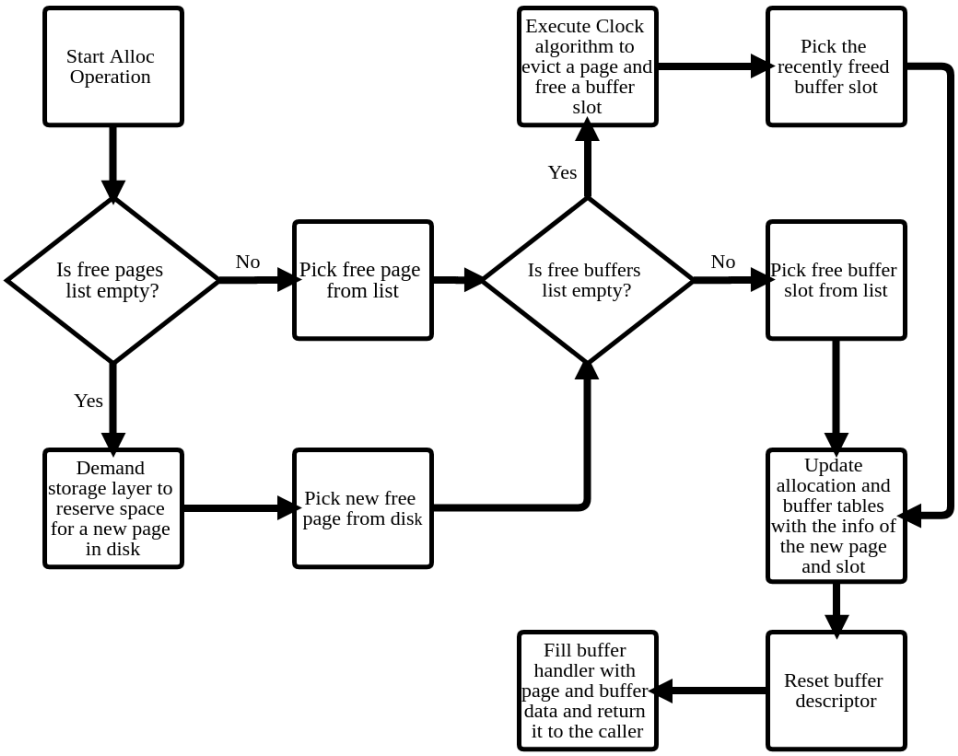

Figure 3.5: Flow diagram of the alloc operation.

to disk. Second, the allocation table is stored into the data file. Finally, all the structures are conveniently freed to avoid memory leaks and the storage is asked for closing the files to keep information safe.

- **Alloc**. This is the method that a process needs to invoke to demand more disk space. When called, a free page is searched between the free pages list of all partitions or, in case none is available, the storage layer is inquired to reserve space in disk accordingly. After so, an empty buffer pool slot is selected checking the free buffers list of the relative partition or executing the Clock algorithm to evict a page if all slots are being used. Then, the allocation and partition buffer tables are updated to reflect the new buffer-page association. Once this is done, the fields of the correponding buffer descriptor are reset and a buffer handler is populated and sent back to the client process to allow it to read or modify the newly allocated page. Figure 3.5 shows a flow diagram to help understanding the behavior of the alloc operation.
- **Release**. A page release is intended to be issued when a data page stored in disk is no longer needed and can be deleted. First, the database takes the page ID specified by the client process and checks

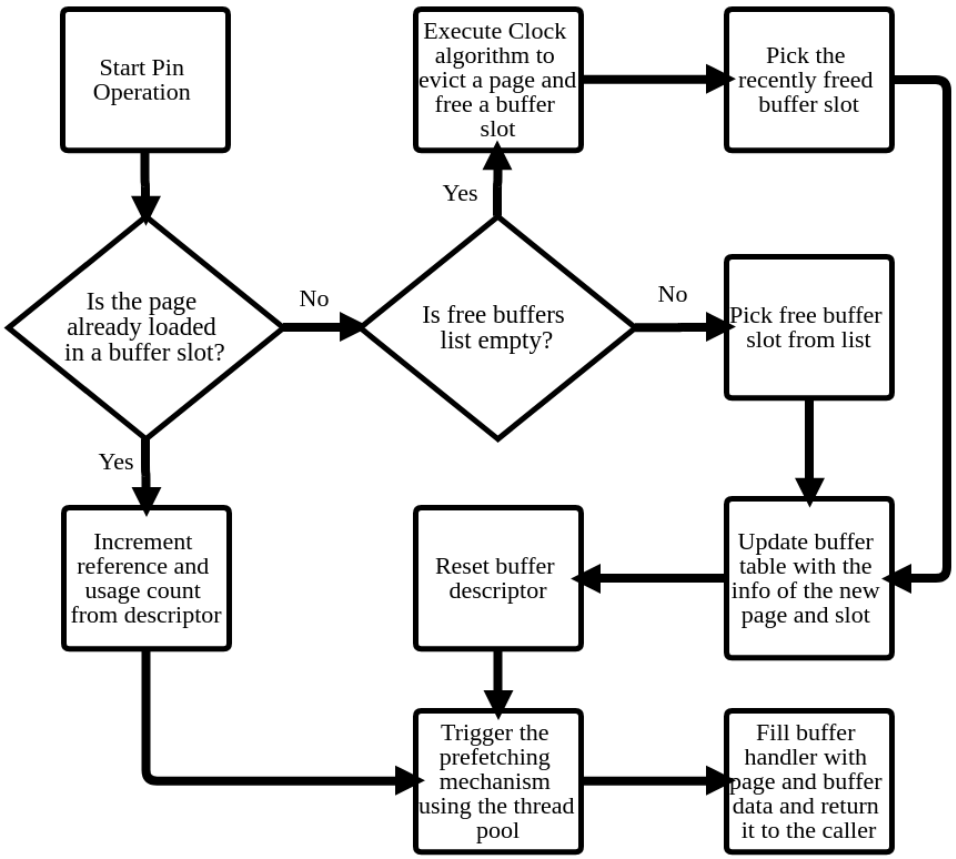

Figure 3.6: Flow diagram of the pin operation.

whether it is currently loaded in the buffer pool by checking the buffer table of the corresponding partition. In case it is not cached, the system only needs to update the allocation table specifying that the page is not allocated and add its identifier to the free pages list of the partition. In the other case, if the page is loaded in a buffer slot, apart from these actions, the database also needs to delete the page entry from the buffer table, write back the page to disk if it is dirty, reset the buffer descriptor fields and add the identifier of the released buffer to the free buffers list.

- **Pin**. Pinning a page is done when a process wants to read or modify some data stored in the database. Because of this, when the caller executes the operation, it indicates the page ID of the page to access and a corresponding buffer handler is given back. However, between this two actions a lot of activity takes place. Figure 3.6 presents a flow diagram of the operation to complement the description.

  The first thing is to check in the corresponding buffer table if the page

is already in the buffer pool. If it is there, we just need to increment the reference count and usage count fields of the buffer descriptor. Else, the page must be read from disk and loaded into a buffer slot – remember that the slot ID should be retrieved from the free buffers list or, in case it is empty, from a page eviction using the Clock algorithm. Then the new buffer-page association can be indicated in the buffer table and the buffer descriptor fields initialized. Finally, the prefetching mechanism is triggered as described in section 3.3.1.6.

- **Unpin**. When a process that has previously pinned a page does not need to access it anymore (until a future task or so), it can unpin the page in order to free resources. What is done is basically to decrement the reference count of the buffer descriptor. Take into account that the page will only be considered unpinned when the reference count reaches zero. Then, although a page is unpinned by a process A, if a process B is still holding it, its reference count will be one, so it could not be evicted during a page replacement.
- **Checkpoint**. That method should be run periodically by an internal process after certain predetermined intervals of time. While blocking the system, it flushes the dirty buffers and saves the current state of the allocation table into disk. Thanks to this and a transaction log mechanism [10] still pending to be implemented, we could recover the status of all data in the database after a power failure during transactions. So, note that although the checkpoint operation is already implemented, the recovery mechanism is not yet available. Notwithstanding, we plan to deliver such feature in the near future.
- **Set dirty**. As it name implies, given a page ID, it sets the dirty bit of its buffer descriptor. This operation must be executed with any page that is modified by a process.
- **Check consistency**. This option has been conceived only with debugging purposes. It locks the system and checks the allocation and buffer tables, the free pages and buffers lists and the buffer descriptors trying to find if there is any incongruence or status corruption. In case something strange is found, an error is raised accordingly.
- **Dump allocation table**. As its name suggests, it shows a dump of the whole allocation table so it can be revised by the user. Once again, this feature is expected to be used only during development phases to check the system's correctness.
- **Get statistics**. We have implemented this call in order to retrieve some relevant information of the database at runtime. Even though its use is mainly focused in debugging, it can also be used to take metrics and analyze its behavior. The stats taken are the number of allocated

pages in storage, the total number of pages reserved in storage and the page size. The returned information can be expanded easily by adding new fields in the `std::struct` defining the set of statistics and computing them conveniently.

3.4 Error Handling, Debugging and Testing

The interaction between the aforementioned layers could possibly lead to inconsistency states or operation errors under some specific circumstances that may have not been taken into account. In those cases, the database should be prepared to abort the execution of the system to avoid greater problems. This is what is called error handling and several techniques for this exist.

In our case, we have opted for defining an error system with several codes refering to particular issues. Then when a problem is detected, the current action's execution is aborted and the corresponding error is returned to the caller to inform about it. Following, we state the different codes that we currently support alongside a definition of the issue they are related to.

- E_STORAGE_INVALID_PATH:
  The provided path where to build the database does not exist.
- E_STORAGE_PATH_ALREADY_EXISTS:
  There is already a database build in the provided path. The error is raised to avoid overwriting it and lose data.
- E_STORAGE_CRITICAL_ERROR:
  Error raised when the storage tries to seek a page in disk and it goes out of bounds.
- E_STORAGE_UNEXPECTED_READ_ERROR:
  There has been some error when calling the low-level `istream::read` method for reading a page.
- E_STORAGE_UNEXPECTED_WRITE_ERROR:
  There has been some error when calling the low-level `ostream::write` method for writing a page.
- E_BUFPOOL_OUT_OF_MEMORY:
  A page has tried to be loaded into the buffer pool but there is no unpinned buffer slot.
- E_BUFPOOL_SIZE_NOT_MULTIPLE_OF_PAGE_SIZE:
  The configuration parameter corresponding to the buffer pool size is

not multiple from the page size parameter. The error is raised to make sure that that all pages have the same exact size.

- E_BUFPOOL_ALLOCATED_PAGE_IN_FREELIST:
  A consistency error indicating that an already allocated page is in the free pages list.
- E_BUFPOOL_PROTECTED_PAGE_IN_FREELIST:
  A consistency error indicating that a page used for storing the allocation table is in the free pages list.
- E_BUFPOOL_FREE_PAGE_NOT_IN_FREELIST:
  A consistency error indicating that a page that is not yet allocated is not in the free pages list.
- E_BUFPOOL_BUFFER_DESCRIPTOR_INCORRECT_DATA:
  A consistency error indicating that a buffer descriptor corresponding to a cached page presents incorrect data (e.g. the in use bit is not correctly set).
- E_BUFPOOL_FREE_PAGE_MAPPED_TO_BUFFER:
  A consistency error indicating that a page that is not yet allocated is loaded in the buffer pool.

Finally, as in every complex software piece, bugs may arise due to minor unchecked changes, the introduction of new functionality, the management of data accessed by multiple threads simultaneously, design errors, etc. In that sense, our database is not exempt of these setbacks. To cope with them, we have build a battery of tests that stress different parts of the system and allow us to guarantee that no existing functionality is broken during development. These have been created making use of the well-kown **Google Test** or **GTest** unit testing library. Although writing tests takes time and the definition of a significant test can be complex, we consider that it is an important part of the development phase that helps stabilization as long as the database grows – obviously, new GTests will be needed to cover future updates.

Chapter 4

Experiments

In this chapter, we evaluate the current implementation of SMILE, as defined in chapter 3. In section 4.1, we introduce the experimental environment used to test both performance and behavior of our database. In section 4.2, we review the test that has been implemented in order to stress the system while simulating the execution of a realistic database operator. Finally, in section 4.3, we analyze how the database behaves under several factors such as the underlying machine, the page size, the number of threads, etc.

4.1 Experimental Setup

In this section, we first describe the setup where our tests have been run and later we expose and describe the main tools used.

4.1.1 Test Environment

The previously presented database has been analyzed under two quite different kinds of system. On the one hand, we have made use of a regular server with great specs to measure how efficient is our design when used by a powerful machine, that is where DB are traditionally deployed. On the other hand, we have also tested it in way simpler device that aims for representing a fog machine, like the ones we are developing the DB for. Hence, we are able to compare the resulting behavior in both ecosystems, so we can have a better understanding of how well it performs.

Table 4.1 shows an overview of the two computers which are correspondingly identified as **server** and **ODROID-C2**. The second one is in fact a specific model in a series of single-board computers created by *Hardkernel Ltd.*, an

	Server	ODROID-C2
Processor	Intel® Xeon® CPU E5-2630 v3 @ 2.40 GHz	Amlogic S905 ARM® Cortex®-A53 @ 1.5Ghz
Number of CPUs	32	4
L1d cache	32 KB	32 KB
L1i cache	32 KB	32 KB
LL cache	20 MB	512 KB
Main memory	256 GB	2 GB
OS page size	4 KB	4 KB
NUMA nodes	2	1

Table 4.1: Technical characteristics of the machines used for evaluation.

open-source hardware company from South Korea. The biggests contrasts between the devices are in terms of computational power – the server is equipped with a x86 processor with lots of threads and the board just has a modest ARM CPU – and in main memory capacity.

4.1.2 Relevant Tools

Several tools have been used to profile the application's execution, test its correctness and gather relevant information. Here, we just focus on the two most relevant ones.

- One is **Page Cache Management for Linux** [8]. This is a software developed by *Google LLC* that allows the user to limit the amount of pagecache used by applications under Linux. This has been useful during development to help us taking measures of our database without having to contemplate partial file caching performed by the underlying operating system. The use of this tool adds an overhead of about 2% in the worst case.
- The other is **Hotspot** [13], developed by *KDAB* and used for general profiling. It is a replacement for perf report, providing a graphical user interface (GUI) that takes a perf.data file, parses and evaluates its contents and then displays relevant information in a graphical way. Apart from the interactive GUI that makes it easier to analyze the data, its killer feature is the built-in flame graph [9]. Flame graphs are a visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately and, unlike other code-path visualizations, it can handle large-scale profiles, while usually remaining readable on one screen. The program have been used during the experiments to profile the database behavior and find possible bottlenecks.

Algorithm 2 Pseudocode of the Sequential Scan test.

```
1:  Counter ← 0
2:  for all p ∈ DatabasePageSet do
3:      Pin page p
4:      n ← 0
5:      while n < NumBytes(p) do
6:          value ← integer(n)
7:          if value > Threshold then
8:              Counter ← Counter + 1
9:          n ← n + 4
10:     Unpin page p
```

4.2 Sequential Scan

In this section, we describe the test that has been used to evaluate SMILE. We name it Sequential Scan and is, in fact, a quite simple test which is briefly described in algorithm 2. It basically loads all pages from the database, one by one, and compares each integer of data inside it with a threshold value manually set. If the value of such integer is greater than that of the threshold, a counter is increased.

This simplicity makes that the amount of work between pin/unpin pairs is small, which implies that the database will be receiveing requests constantly. In order to optimize the depicted process and increment the pressure on the multi-thread support of the database – which allow us to observe the system's response to heavy workloads –, we make use of the OpenMP library (`omp.h`), which is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in *C/C++* and *Fortran* [5].

The strategy followed to parallelize the code has been to divide the set of iterations – one per each DB page – and evenly assign them to each thread, which will be updating its partial counter separatedly. More specifically, each thread takes care of pages satisfying: $pID \mod N = tID$, where *pID* stands for the page ID, N for the total amount of threads and *tID* for the thread ID. Thence, by carefully configuring both the test and the buffer pool, it is easy to make that each CPU works with pages from its same NUMA node (see section 3.3.1.5). Then, the more threads are used, the smaller are the chunks to be processed and, since all run in parallel, the execution time is accordingly decreased. Notice that our tests are prepared to be configured and run using any amount of threads as long as there are enough available resources in the system.

Finally, a single CPU is in charge of computing the aggregate of the partial

counters when all threads have finished their work. In order to avoid false sharing between threads when accessing to the array of partial counters, extra padding have been added so that no pair of counters share the same cache line.

Scan operations are, indeed, one of the most elementary and important operations in data processing. Our scan does some little computations to make it a bit more realistic and simulate, for instance, a table scan trying to find the amount of entries having a field that fulfills a specific condition. Moreover, due to its nature and simplicity, it is very useful operation to show up to which point of saturation of request can our database work efficiently. This evaluation is depicted in section 4.3

4.3 Evaluation

In this passage, we present the results of our evaluation of SMILE in the two aforementioned environments. Thereby, section 4.3.1 will focus on the server machine, whilst section 4.3.2 will cover the ODROID's experiments. During this work, we have focused on optimizing the buffer pool specially for sequential accesses, which are one of the most common patterns in database operators. Because of this, all the experiments have been performed with different configurations of the Sequential Scan. The case of SMILE's performance with random access workloads is not covered in this research and is left as future work.

4.3.1 Results in the Server

First, we will focus on the server machine to be able to see where are we in terms of performance when using our database in regular equipment for this kind of software. The system is initially tested with in-memory tests and later some out-of-core test results are provided.

4.3.1.1 In-Memory Server Results

In figure 4.1 we show the timings obtained when runnning several iterations of the Sequential Scan test one after the other and starting from a state in which no part of the data is either loaded in the buffer pool nor cached by the system. On the one hand, the system has been configured to use a 1 GB buffer pool, 64 KB pages, 1 partition – this is, without partitioning – and the NUMA-aware mode and the prefetcher disabled. On the other hand the test has been configured to use just one thread and cover 1 GB of pages, thus all data fits in memory.

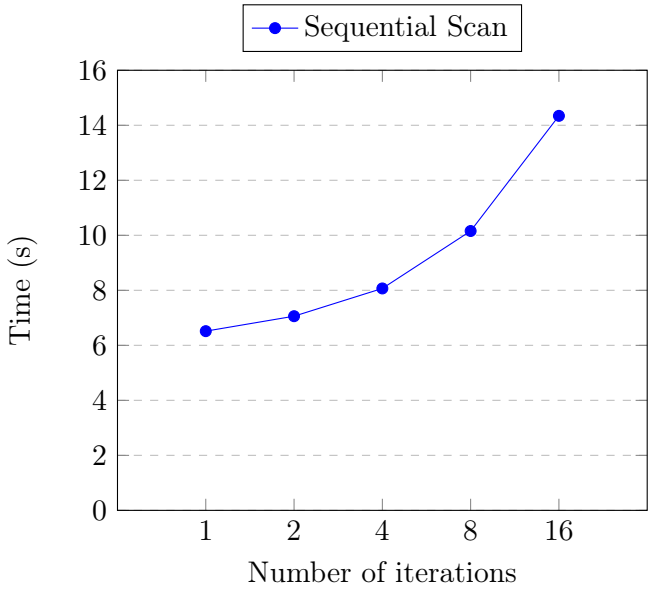

Figure 4.1: Sequential Scan timings in the server depending on the number of iterations over the pages.

When doing a single iteration, it takes about ∼6.5 seconds to load the whole gigabyte of information and performing the corresponding computations with the threshold and counter updates. Nevertheless, in the two iterations case, the whole process lasts about 7 seconds, only half a second more approximately. In that case, during the second run the data is already present in the buffer pool, so the data access time lowers a lot and the thread almost only needs to take care of the processing part. In the following cases of 4, 8 and 16 iterations it can be observed how this behavior remains quite constant, with a base cost of $\sim 6.5s$ for loading the data into the DB in the first iteration and an extra cost of $\sim 0.5s$ per each repetition. This simple experiment evidences that the buffer pool does its job of facilitating disk accesses to the client processes.

Prior to trying out other configuration parameters, we show in figure 4.2 how variations in the page size used affect the DB's efficiency. Here, the experiment has been done again with a 1 GB buffer pool and no partitions, NUMA-awareness nor prefetching; the test also is run over 1 GB of data and with one thread. As can be seen, differences are, in fact, not very big between samples ($\pm 0.028s$). Nonetheless, it is clear that as long as we keep increasing the page size we obtain better results until reaching the frontier of 64 KB, point from which on, performance starts to degrade again. The suitability of a given page size is intrinsic to the underlying machine characteristics, so different computers may perform better with distinct ones. Since it is proved that 64 KB is the most adequate configuration for the server machine, we

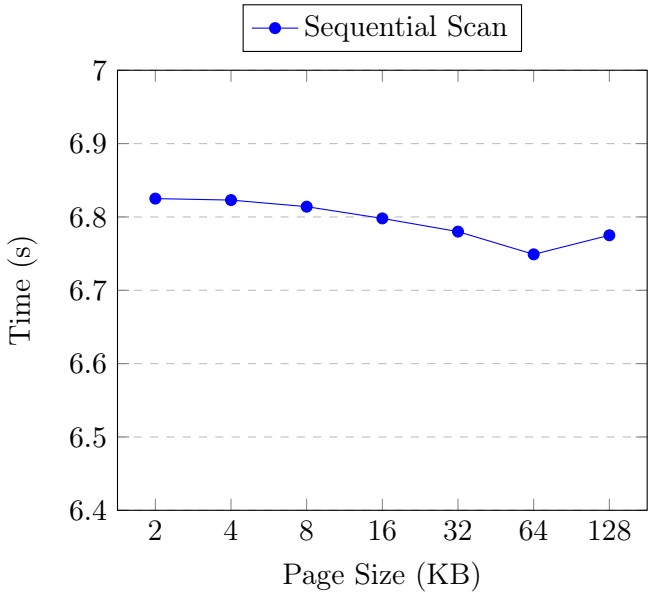

Figure 4.2: Sequential Scan timings in the server depending on the page size.

will keep it for the next experiments.

Now its time to evaluate the scalability of the buffer pool and, to do so, we compare the results of the execution of the Sequential Scan test with different threads. Moreover we want to see if partitioning the buffer pool really helps in that sense, so we have different series of samples depending on the amount of partitions used. For the rest, the buffer pool has been configured with a 1 GB of buffer slots, 64 KB pages and the NUMA-aware mode and prefetcher disabled. The test still covers 1 GB of pages but with a little variation to avoid the variability of the results: a first iteration is done for warming up the buffer pool and, what is measured, is the total time of executing ten subsequent iterations of it. This information is presented in figure 4.3.

What can observed is that, overall, as long as threads are doubled, the execution time is almost halved, meaning that all series scale very good, at least up to 8 threads. Nevertheless, overhead is appreciated in the cases with 16 and 32 threads since they no longer scale linearly. The 32 CPUs of the server are arranged into two NUMA nodes with 16 CPUs each; hence, as we approach or surpass to the amount of threads available in a NUMA node, it is understandable that such overhead arises as a result of the communication of buffer pool data between both nodes. In that scale is difficult to observe the resulting speedups of using partitioning. Because of this, we also show figure 4.4 with the relative speedup of each case when compared with the non-partitioned version.

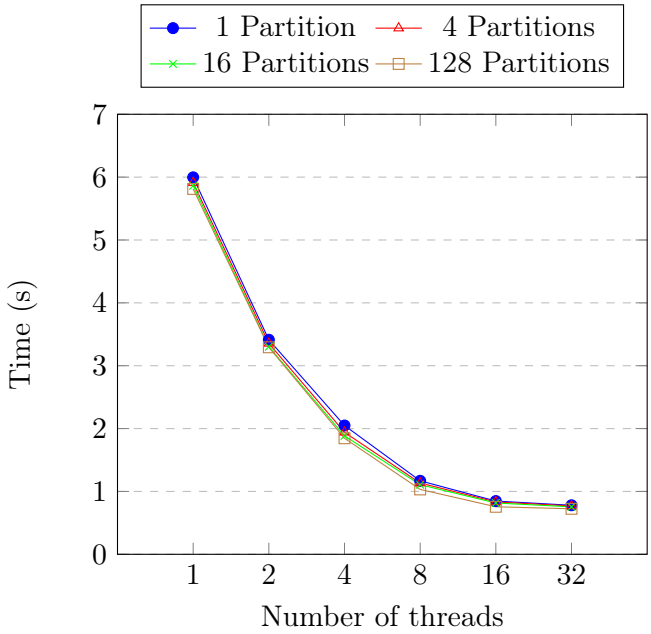

Figure 4.3: Sequential Scan timings in the server depending on the number of threads and partitions.

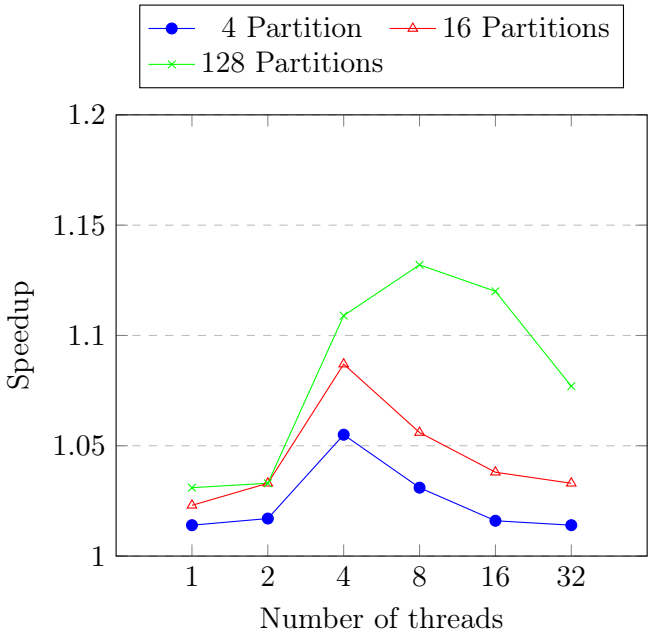

Figure 4.4: Sequential Scan speedups in the server depending on the number of threads and partitions.

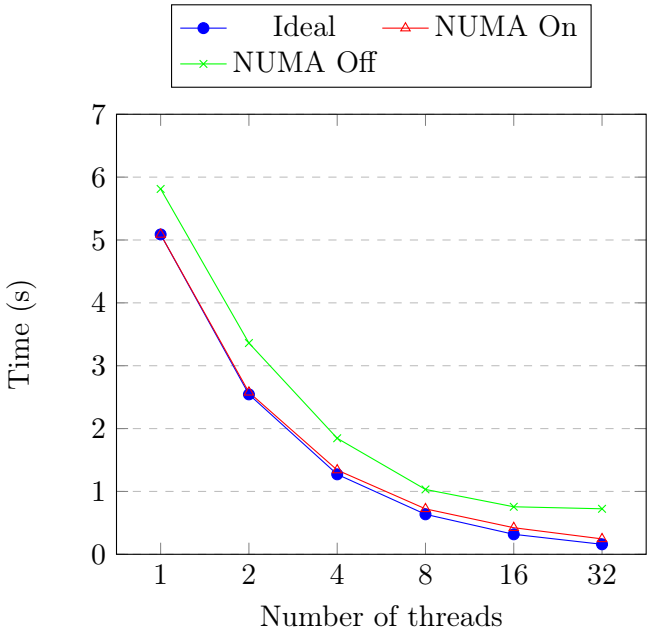

Figure 4.5: Sequential Scan timings in the server depending on whether the NUMA-aware mode is activated or not.

Our experiment shows that the more partitions exist, the better performance is when multiple clients make requests to the buffer pool. The 4-partitions version has a ∼5% speedup when using 4 threads, however, it tends to go down as more threads are used. This is because using more than 4 threads increases a lot the probability of two processes colliding in a same partition, causing lock contention. The 16-partitions case has a similar plot shape but with better results, as expected, due to being able to lower the collision probability. As it is obvious, the best results are obtained using 128 partitions, which runs up to a ∼13% faster than the original version. Nonetheless, as before, there is a point where lock contention starts increasing again although the system is highly partitioned, reducing the possible speedup.

The final step is to improve the execution to avoid the overhead from NUMA communications so that we can still scale linearly with 16 and 32 threads. In this point is where our NUMA-aware mode joins in scene. In figure 4.5 we compare three series: one making use of this feature, other one that does not and the ideal performance scale. Both real cases have been measured using the same buffer pool and test configuration as in the previous experiment, with the exception of the number of partitions, which we have opted to fix at 128 since is the one that proportioned better results.

Thanks to the NUMA-aware mode, the 16 and 32 threads cases are able to continue scaling well, allowing the Sequential Scan to reach its fastest timing,

which is found at $\sim 0.025s$ per iteration. This means that, with the already presented optimizations, our buffer pool is able to obtain a $20X$ speedup over the baseline access time in an intensive test that tries to saturate it. It also must be noticed that this mode does not only helps to scale in architectures with multiple NUMA nodes, but that also provides way better results although using a limited number of CPUs. Thence, we can conclude that using NUMA-awareness should be the preferred choice in all situations. Furthermore, as can be seen in the figure, the NUMA results are indeed really close to the ideal ones, which stands for how well our optimizations work.

4.3.1.2 Out-of-Core Server Results

We also need to give some measurings regarding SMILE's performance for out-of-core workloads and the results of the prefetcher, which tries to speedup these cases. To do so, we have taken our top configuration – which, as reasoned through this section, is using 64 KB pages, 128 partitions and with the NUMA-aware mode on – and have tested, with and without prefetching, the Sequential Scan doing a single iteration loading 1 GB of data into a buffer pool completely empty at start. In more detail, the prefetcher has been configured to fetch only the next page for each pin. Moreover, the thread pool has been set with 16 threads, implying that we have only tested scaling up to 16 threads since the rest are exclusive for prefetching. The outcomes are given in figure 4.6.

On this occasion, the feature does not meet the expectations and is completely unable to improve the execution. But it is not just that, it also exists some payoff which is causing the program to work even slower. It must be remembered that our prefetching mechanism is very simple and limited to a few next pages. One could expect that, due to how we split iterations among CPUs, an execution with more than one thread makes prefetches to be issued too late (consecutive pages are managed by different threads *simultaneously*). However, an execution with just one thread should be able to obtain some benefit (consecutive pages would be managed by the same CPU). That is neither the case, suggesting that prefetches are done too slow to be useful, the root cause of which can be that the thread pool is not well optimized and represents a loss of time or that there is not enough work between pins to overlap prefetches with other computations. Then, this has to be one of the next points where future development must be focused.

In any case, we observe that the more threads are used, the slower the program is executed when it requests non-buffered pages. This is because disk accesses are only sequential when using a single thread, and, as long as more processes are used, such sequentiality gets more and more diluted. This

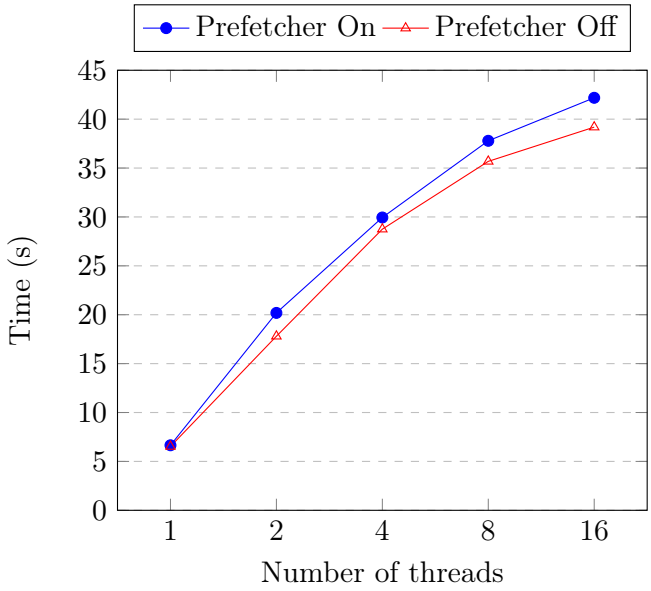

Figure 4.6: Sequential Scan timings in the server depending on whether the prefetcher is activated or not.

behavior causes disk reads to take more time when using multi-threading. In order to cope with this issue and, overall, improve accesses, we propose disk access coalescing as a future work technique to be implemented in SMILE.

4.3.2 Results in the ODROID-C2

After observing the server results, we want to evaluate the buffer pool's behavior in more modest hardware, so, now we jump to the ODROID-C2 machine to run the tests.

4.3.2.1 In-Memory ODROID-C2 Results

Similarly as before, we start by looking at the execution times resulting from running several iterations of the Sequential Scan test from a state in which the buffer pool has no data. However, given the differences in processing capability between the two machines, we have adapted the configuration of the test to avoid it to take much time. Then, the buffer pool has been initialized with 256 MB buffer pool, no partitions, 64 KB pages and the NUMA mode and the prefetcher disabled. Also, the Sequential Scan has been adjusted to run one thread and cover 256 MB of pages. The results of experiments are found in figure 4.7

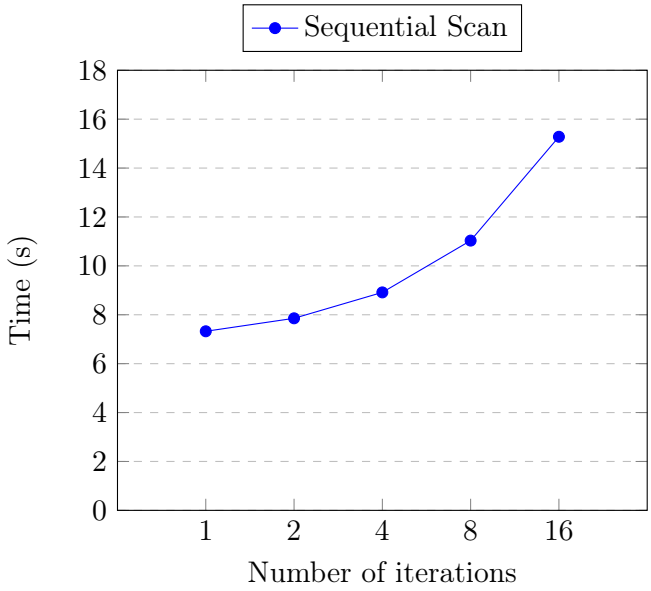

Figure 4.7: Sequential Scan timings in the ODROID-C2 depending on the number of iterations over the pages.

Once again, the aim of this experiment is to evidence that the buffer pool works correctly and lowers disk access time to client processes. The 1-iteration sample shows that ODROID-C2 takes $\sim 7.3s$ to access disk and load 256 MB into the buffer pool. If a second iteration is done, its timing lowers a lot as seen in the second sample, which also applies if more and more iterations are done. Overall, the time for computing an iteration with the data already loaded in the system lowers up to $\sim 0.5s$, which means a speedup of $14.6X$.

As before, the next step is to check the effect of the page size on performance. To do so, we have experimented keeping the previous configuration but launching a single iteration and assigning different page sizes. Figure 4.8 depicts so, where we can find a quite distinct conduct than the one seen in the server (figure 4.2). If we remember, performance in the server tend to be better the more we increased the page size; being 64 KB an inflexion point from where larger sizes started to misperform. Nevertheless, here it is difficult to identify a clear tendency, for instance, 4 and 32 KB page sizes work better than the 8, 16 and 64 ones. In any case, differences are about ($\pm 0.011s$), which is not much and is even less – both proportionally and in absolute terms – than the ones of the server. Since the better result is given by the 4 KB pages ($\sim 7.29s$), we will stick with it for the remaining experiments.

Now its time to evaluate the scalability, so we want to test the use of mul-

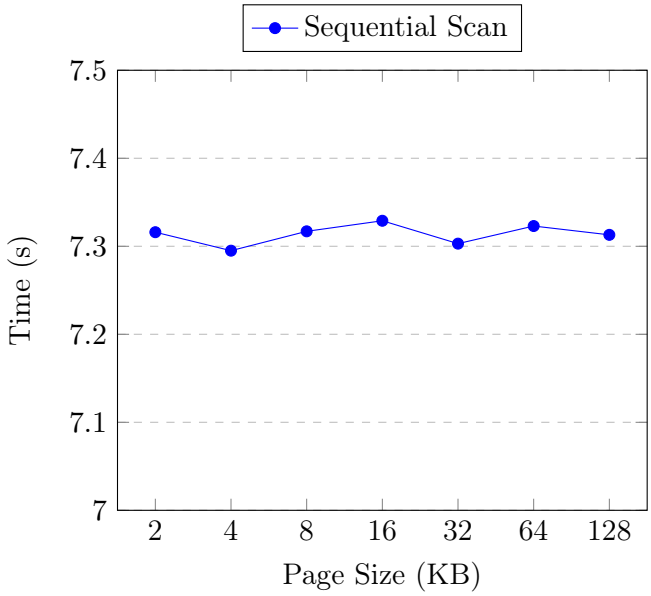

Figure 4.8: Sequential Scan timings in the ODROID-C2 depending on the page size.

tiple threads and different partitioninig. We must remark, again, that the ODROID-C2 is a way less powerful machine and that the amount of threads it has is just 4. Because of this, we are just evaluating the system with 1, 2 and 4 threads and 1, 2, 4 and 16 partitions – it makes no sense try out greater partitionings if there are not enough CPUs to make it profit. The rest of parameters are kept the same as in previous configurations.

In figure 4.9, we observe the execution times of the different partitioning configurations compared with the ideal outcomes for each amount of threads used. In that sense, the regular version does not scale perfectly but is able to reach a $2.46X$ speedup using 4 threads. If we compare it with the partitioned versions, we see how the rest scale better as we increase the amount of workers. That is because partitioning allows for a better scaling with fewer lock contention as shown in the figure and previously observed in the server. However, the results are still quite far from the ideal ones, which can be related to the relatively low processing power of the machine.

The granted speedup of each version respect to the original is shown in figure 4.10. As expected, the best results come from using 16 partitions, which achieves a $\sim$10% faster execution when using 4 threads, compared with the baseline. Nonetheless, it impossible to benefit more from partitioning because ODROID-C2 has a very limited quantity of threads and so the system can not be stressed more. If we could continue scaling with more threads, we would probably observe greater differences between configurations.

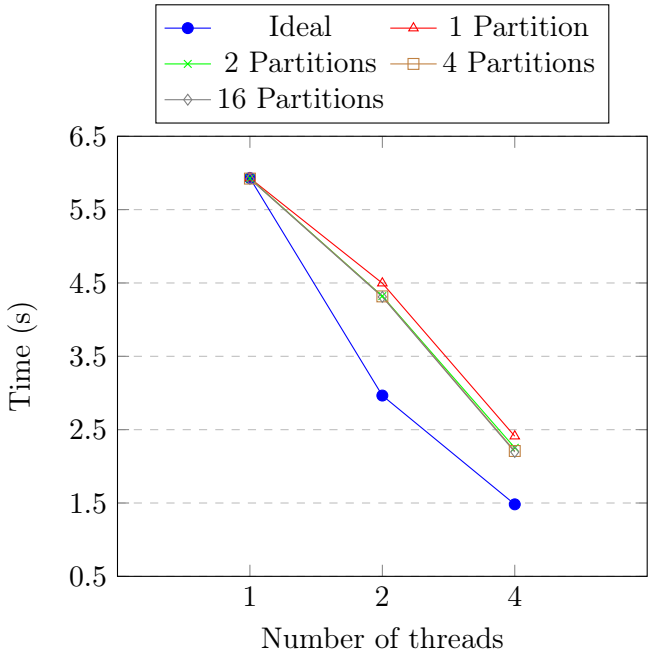

Figure 4.9: Sequential Scan timings in the ODROID-C2 depending on the number of threads and partitions.

In the server, we have tested and presented the differences observed when the NUMA mode is activated or not, but we will not be doing so with the ODROID-C2. The reason behind so is that the case that concerns us now has jut a single NUMA node, which means that the behavior of the buffer pool is the same having activated such option or not. Thence, to avoid redundant tests and explanaitions we omit such part from the evaluation. Notwithstanding, we expect in the future to have low power devices with more threads and NUMA domains, so that the already applied and tested NUMA-aware technique can improve, even more, SMILE's execution in this context.

4.3.2.2 Out-of-Core ODROID-C2 Results

Finally, we only lack a prefetcher demonstration on the ODROID-C2, which is found in figure 4.11. In this simple experiment, we have used the buffer pool setup of the last test and have run the Sequential Scan with and without the prefetcher enabled to see its timings when doing a single iteration with no data previously loaded into the buffer slots. As with the server, the thread pool has been configured to have a worker thread per each thread used to run the Sequential Scan. This gives us only two combinations: one OpenMP and prefetching threads, or two OpenMP and prefetcher threads.

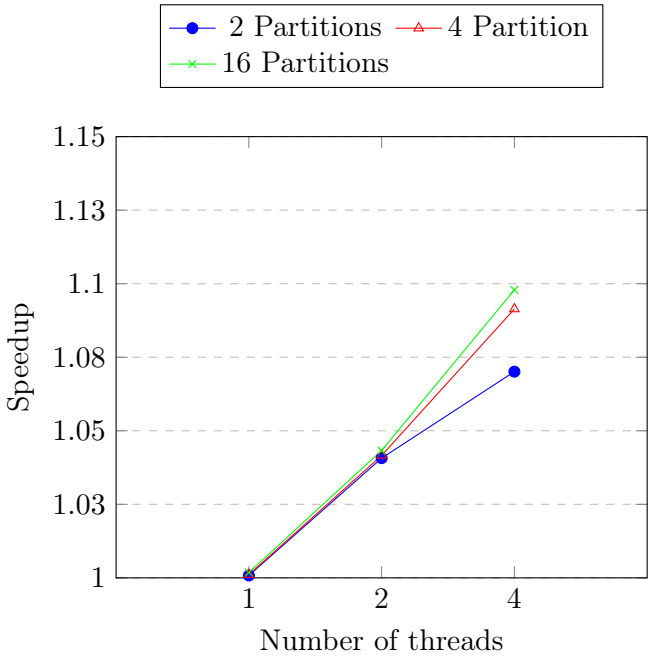

Figure 4.10: Sequential Scan speedups in the ODROID-C2 depending on the number of threads and partitions.

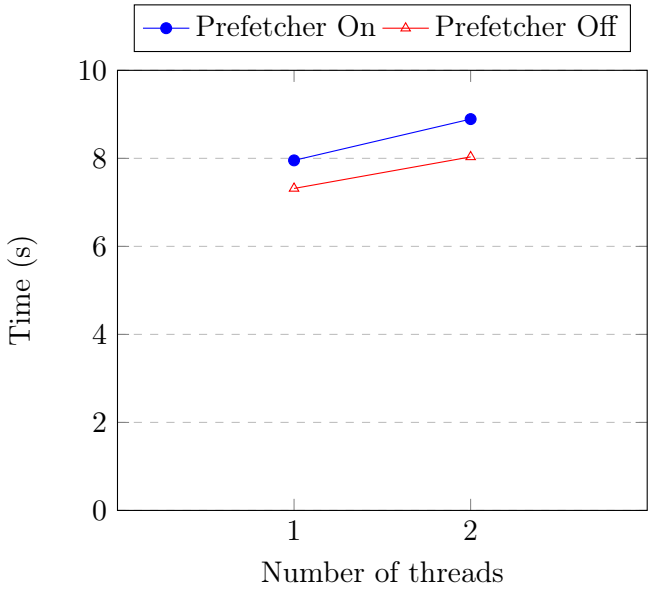

Figure 4.11: Sequential Scan timings in the ODROID-C2 depending on whether the prefetcher is activated or not.

Before, in the server, we proved that our thread pool was not well optimized,

making the code to run slower than when it is enabled. The results in the ODROID-C2 serve to nothing else but to strengthen this belief. However, the architectural differencess between the two machines make such penalty to be slightly different in both systems. These outcomes must be taken into account and part of the next future of this database project should be considered to be spent into improving the current thread pool mechanisms.

Chapter 5

Future Work

The near future of our research will be oriented to maximizing the efficiency of the buffer pool in any kind of environment by coping with some current issues and leveraging upcoming optimizations. In this chapter, we take a closer look to the main research lines in which we plan to focus our research in the short term.

5.1 Additional Tests and Operators

In section 4.3, we analyzed the performance of our database by using the Sequential Scan. Nevertheless, we have developed other more complex operators, aiming to provide us with tests representing real workloads and with different access patterns so that we could see the particularities of each kind of interaction with the buffer pool and be able to know in which areas we can improve the system. Unfortunately, the majority of them still not represent a stressful enough task for our system to be useful to demonstrate its performance depending on how we configure it.

Because of this, testing has been only done with the sequential scan, the one with greater proportion of buffer pool requests / computation. In any case, these other tests are available to anyone who wants to try them with SMILE. More specifically, their code is also found in the GitHub webpage of the project and are prepared to be run as regular GTests.

As part of the future work, we plan to revisit and improve these extra operations in order to have more available test suites with which stress SMILE and gather data. Furthermore, this would help to evaluate SMILE's performance when running workloads that perform random accesses, which – as explained in section 4.3 – has been left as future work.

In this section, we summarize the current implementation of the rest of tests.

5.1.1 Allocator

This test starts by creating a whole new database configured with a page size that can be chosen by the user. Then, it begins to allocate pages until the total size of the database reaches a particular value specified manually. Nevertheless, these pages are not only allocated but also filled with information. More specifically, each byte on every page is assigned a randomly generated 8-bit value and, finally, all pages are flushed to disk, so the data is kept safe. This is the most simple operator, yet it is important because it allow us to create new databases, initialize them and populate them with data. In this way, the subsequent operators can work directly with an already existing DB, which simplifies the process since tests do not need to generate its own data before doing its calculus.

5.1.2 Group By

The Group By statement is a database operator often used with aggregate functions (e.g. count, max, min, sum, avg, etc.) to group the result-set by one or more columns. The test that occupies us tries to simulate such behavior over a database previously created with the allocator. Hence, it requests all pages present in the database by pinning them and performs the operations showed in algorithm 3 before unpinning them. These operations basically consist in counting the total amount of appearances of each 8-bit values thanks to a `std::map`, which is similar to the procedure used to group real query results by a given parameter.

The process of parallelizing this test is quite similar to that of the Scan Filter. Pages are distributed among threads in the same fashion and instead of counters, partial tables (implemented with `std::unordered_map`) are used and converged in a final aggregation.

5.1.3 Hash Join

The hash join is an specific type of join algorithm which is used by relational database management systems. The task of such join algorithm is to search, for each different existing value of the join attribute, the set of tuples in each relation which have that value. As with Group By, we have designed a test trying to emulate its behavior in our database, a pseudocode of which is presented in algorithm 4. Here, we divide the DB pages into two groups, the

Algorithm 3 Pseudocode of the Group By test.

```
1: OccurrencesTable ← ∅
2: for all p ∈ DatabasePageSet do
3:     Pin page p
4:     n ← 0
5:     while n < NumBytes(p) do
6:         value ← byte(n)
7:         if value ∉ OccurrencesTable then
8:             OccurrencesTable(value) ← 1
9:         else
10:            OccurrencesTable(value) ← OccurrencesTable(value) + 1
11:        n ← n + 1
12:    Unpin page p
```

build and the probe sets. The build one is formed by the first sixteenth part of the pages, which are used to assemble a hash table by taking groups of two bytes considering them as key-value pairs and filtering them to discard the odd keys. Then, the algorithm uses the probe set of pages – composed by the rest – to test the hash table and perform a reduction of the values associated to a same key. This procedure would simulate the execution of a Hash Join requesting the values of the tuples whose join attribute is even.

This operator has also been optimized by means of OpenMP directives. In that sense, the construction of the hash table is done in parallel using multiple threads. More specifically, partial tables are generated per each thread and are put together by a single one. After that, the probe phase is launched dividing pages in consecutive chunks in the same fashion used in Scan Filter and Group By.

5.1.4 Load Graph

As its name may suggest, this test is intended to load a graph into a full new database. Alongside the allocator test, they both represent the only auxiliar operations that we provide. The first one is expected to be used to initialize a DB with the needed information to be able to run the Group By or the Hash Join algorithms. Complementary, the load graph test should be used to achieve loading a graph into the DB, so that we can run the BFS test (see section 5.1.5) on it.

Contrary as happens in the allocator case, here data is not randomly generated since creating aleatory realistic graphs could be a complex process. Instead, we have opted for taking profit of some already existing schemas that can be freely accessed and downloaded from the Stanford Large Network

Algorithm 4 Pseudocode of the Hash Join test.

```
1: BuildSet ← {p | p ∈ DatabasePageSet ∧ Index(p) < NumPages/16}
2: ProbeSet ← {p | p ∈ DatabasePageSet ∧ Index(p) > NumPages/16}
3: HashTable ← ∅
4: for all p ∈ BuildSet do
5:     Pin page p
6:     n ← 0
7:     while n < NumBytes(p) do
8:         key ← byte(n)
9:         value ← byte(n + 1)
10:        if key mod 2 = 0 then
11:            HashTable(key) ← value
12:        n ← n + 2
13:    Unpin page p
14: for all p ∈ ProbeSet do
15:    Pin page p
16:    n ← 0
17:    while n < NumBytes(p) do
18:        key ← byte(n)
19:        value ← byte(n + 1)
20:        if key ∈ HashTable then
21:            HashTable(key) ← HashTable(key) + value
22:        n ← n + 2
23:    Unpin page p
```

Dataset Collection (`https://snap.stanford.edu/data/`). Such compilation provides graph data extracted from real applications (Facebook, Twitter, YouTube, Wikipedia, etc.) alongside a brief description of each dataset in terms of the number of nodes and edges and the type of the graph, that is: whether it is directed or undirected, bipartite, weighted and so on. In every case, data is presented in a plain text file indicating first the number of nodes, then the number of edges and finally a list of ordered pairs whose first element represents a specific node and the second one a neighbor of it. Notice that there are as many pairs as edges exist in the graph and that there are no repetitions. However, before processing the files, we always run Gorder on them to improve the graph data locality.

The test takes profit of the fact that such format is respected in all files and implements a simple way to load any graph dataset into a database. More in detail, a first DB page is reserved to store graph metadata useful when trying to manage that information once stored. This metadata includes: the number of nodes and edges, the starting pages for node and neighbors

information, the number of elements per page and the IDs of the first and the last nodes having neighbors. Finally, the rest of the pages are used to store two vector structures that hold the graph data – nodes and neighbors – in a state-of-the-art format known as Compressed Sparse Row (CSR) that supports efficient access and is proven to obtain overall better performance over other implentations on Breadth-First-Search, Depth-First-Search and Dijkstra algorithms [27]. We name this two structures as the *FirstNeighbor* and the *Neighbors* vectors.

On the one hand, *Neighbors* is an ordered list with the neighbors of all nodes. On the other hand, *FirstNeighbor* has as many positions as nodes exist in the graph, each one holding an index to the *Neighbors* vector. Hence, *First-Neighbor* can tell us the location in the *Neighbors* list where the first neighbor of any given node is located. If a node has no neighbors, its *FirstNeighbor* index will be zero, whilst, if it has multiple neighbors, the rest will be in consecutive positions of *Neighbors*.

For running this test, we encourage to make use of an application which is named **Gorder** [28]. Given a graph, Gorder is intended to generate the node ordering that preserves the graph data locality. The node ordering is used to renumber the node IDs in the original dataset and a new dataset with the new node IDs is the output of the software. The new graph keeps the same topological structure of the original dataset. Gorder can be used to adapt the graph datasets to use state-of-the-art data locality optimizations.

5.1.5 Breadth-First Search

The last test designed has been an implementation of a traditional BFS algorithm, which consists in traversing graph data structures by starting at some arbitrary node and exploring all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. As explained in 5.1.4, BFS can only be run in a database containing a graph, which is achieved by using the load graph test. This BFS test is intended to force our database to serve random, thus non-sequential, page requests which should be more expensive – in terms of time – that serving them in a consecutive fashion, as demanded in the Group By or Hash Join cases.

The behavior of the program is the one expected by such algorithm but adapted to the use of our DB design, minimizing the number of page pins and consulting only the needed data instead of reading the full structure into memory before start. Algorithm 5 defines the operation of the BFS and its necessary pins. At the beginning, some graph information that is needed to control the flow of the program is retrieved from the metadata page. The rest of the program consists in traversing the nodes (starting at

a random one) by first computing the number of neighbors consulting the *FirstNeighbor* vector and, second, use the obtained index and its offset range – the number of neighbors – to iterate over the *Neighbors* array in search of them. Notice that the whole process is repeated a specified number of iterations, which is done to be able to stress the system with several BFS without limiting us with a single one each time.

5.2 Thread Pool

A thread pool could be a possible optimization for SMILE and is, in fact, under development. The idea of disposing of a thread pool is to make – in the long run – that all the operators run on top of such threads, thus allowing us to deliver asynchronous I/O. However, the current implementation is quite basic and is only used to issue prefetches, as explained in section 3.3.1.6. In this section, we try to give a design overview of our thread pool and describe its methods.

5.2.1 Internal Design

Our thread pool has been implemented using the concept of fibers. Like threads, fibers share address space, but instead of using preemptive multitasking, they use cooperative multitasking. Threads depend on the kernel's scheduler to preempt a busy thread and resume another one; fibers yield themselves to run another fiber while executing. Since fibers are managed in the user space, expensive context switches and CPU state changes need not to be made. This makes changing from one fiber to the other extremely efficient, which is the main reason why we have opted for them to run internal database operations.

Furthermore, the thread pool allows to be configured with different number of fibers depending on our needs. For the sake of explaining its behavior, we present a simple two-threaded example scheme in figure 5.1. Tasks can be created at any given time and are queued into the **to start task pool**, waiting until the thread picks them. Notice that there is not just one queue, but one specific for each thread. Hence, each fiber only retrieves tasks from its own queue, which makes possible to focus each thread in a different type of task.

Fibers run the jobs until they are done or until they yield. In the first case, the task is considered completed and its resources are freed. In the second, the task context is saved and it is requeued into a different place, the **running task pool**, for which an instance exist for each thread too.

Algorithm 5 Pseudocode of the Breadth-First Search test.

```
1:  NumberOfNodes ← total number of nodes from metadata
2:  NumberOfEdges ← total number of edges from metadata
3:  FirstNodeWithNeighbors ← first node with neighbors from metadata
4:  LastNodeWithNeighbors ← last node with neighbors from metadata
5:  i ← 0
6:  while i < TotalIterations do
7:      ∀n ∈ nodes, visited(n) ← 0
8:      queue ← ∅
9:      current ← random mod NumberOfNodes
10:     visited(current) ← 1
11:     queue.push(current)
12:     while queue is not empty do
13:         current ← queue.pop_front
14:         Pin page where FirstNeighbor index of current is located
15:         fnc ← FirstNeighbor(current)
16:         NumberOfNeighbors ← ∅
17:         if fnc = 0 ∧ current ≠ FirstNodeWithNeighbors then
18:             NumberOfNeighbors ← 0
19:         else if current = LastNodeWithNeighbors then
20:             NumberOfNeighbors ← (NumberOfEdges − 1) − fnc
21:         else
22:             found ← 0
23:             next ← current + 1
24:             while found = 0 do
25:                 Pin page where FirstNeighbor index of next is located
26:                 fnn ← FirstNeighbor(next)
27:                 if fnn ≠ 0 then
28:                     NumberOfNeighbors ← fnn − fnc
29:                     found ← 1
30:                 Unpin page where FirstNeighbor index of next is located
31:                 next ← next + 1
32:         Unpin page where FirstNeighbor index of current is located
33:         n ← 0
34:         while n < NumberOfNeighbors do
35:             Pin page where Neighbors position fnc is located
36:             neighbor ← Neighbors(fnc)
37:             if visited(neighbor) = 0 then
38:                 visited(neighbor) ← 1
39:                 queue.push(neighbor)
40:             Unpin page where Neighbors position fnc is located
41:             fnc ← fnc + 1
42:             n ← n + 1
43:     i ← i + 1
```

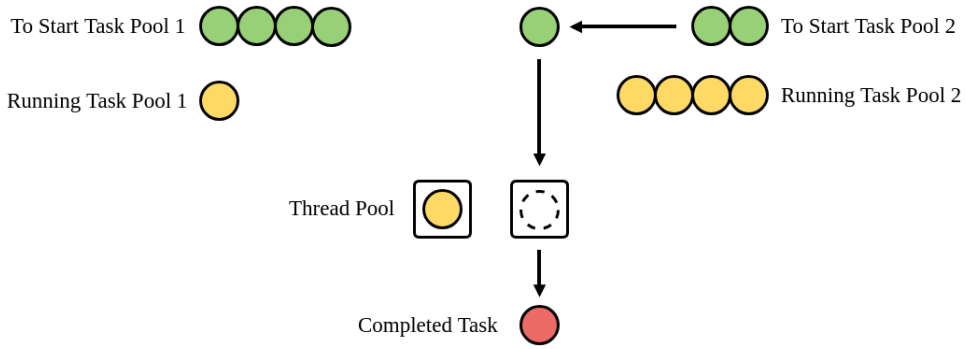

Figure 5.1: Schema of the thread pool.

At a low level, tasks are defined as a struct containing two pointers, one to the function to be executed as part of the task and another one to argument data that will be passed. Moreover, tasks are encapsulated into a bigger structure named task-context, which mainly covers the task itself, its execution context – to be able to save and resume execution – and an atomic counter that is used to synchronize threads, if needed. Thence, each fiber runs a basic thread code which takes care of retrieving or storing the task-contexts from their own *to start* or *running task pools* and execute the tasks.

Part of the future work for the thread pool is to enable work stealing. Such feature would allow a fiber with no pending tasks to *steal* one from other fiber's saturated queue. Notwithstanding, we expect to revise and adjust the whole mechanism to optimize it as much as we can.

5.2.2 Operations

Like in the storage and the buffer manager layers, the thread pool provides an interface with several methods to interact with it. Following, we review them.

- **Start thread pool**. It is the first thread pool call that must be performed if the database plans to use fibers. It takes an integer representing the number of threads to use and initializes the whole pool, its structures and instantiates the necessary threads.
- **Stop thread pool**. When issued, it tells threads to stop executing and waits until each one joins. After, all memory structures and resources are freed.
- **Execute task asynchronously**. This method is used to queue a task into an indicated to start thread pool and leave it to execute asynchronously respect to the caller. This method can be used by the

Buffer Pool to issue an asynchronous task to a specific queue.

- **Execute task synchronously**. Similarly as before, this operation queues a task into a to start thread pool but it also enforces synchronous execution respect to the caller. This method can be used by the Buffer Pool to issue a synchronous task to a specific queue.
- **Yield**. That is the action that enables cooperative multitasking by telling a fiber to yield, this is, to leave CPU to be able to run another task instead.

5.3 Other Research Lines

Apart from the already presented research lines, we have in mind a couple more that have not been that much developed. Following, we describe them.

- **Disk access coalescing**. This was a topic planned in the early stages of the project but that had not enough time to be developed and fitted in this master thesis. It aims to design a system so that allows the buffer pool to detect close-in-time disk access requests and merge them into a single transaction, needing less time to serve the same amount of data. This is a non-trivial feature that would consist in providing the buffer manager layer with a buffer to queue disk access requests. Then, a File Storage thread could easily check the buffered requests, order them in the most convenient way and execute them one after the other. Once finished, it could check the new requests in the buffer to start again the process and so on. Thence, we consider that this possibility must be conveniently evaluated to be included in the future since it should enhance the results obtained in our evaluation.
- **Prefetching mechanism**. Although our buffer pool presents a sort of next page prefetcher, it is evident that it still lacks some optimization efforts to be beneficial for non-sequential access patterns. Specifically, investigating about state-of-the-art thread pool implementations could help us to improve its performance and allow us to obtain substantial speedups with the already running prefetcher.

  Regarding the same topic, another improvement could be done by replacing the prefetcher algorithm by another more efficient. It must be taken into account that we have opted for a simple next page prefetcher, which ideally could make run faster the sequential accesses to pages. Nevertheless, not all programs behave in that way, so processes performing strided accesses or following more random patterns are unable

to see any benefit in our database. Hence, future research could focus on implementing a better mechanism such as a strided, stream or correlation prefetcher, which are concepts already developed in other fields such as hardware data prefetchers [26].

Obviously, apart from the mentioned research lines there is a lot of work to be done until our database project is finished. We expect to deliver higher levels of abstraction to allow multiple data types representation, the definition and implementation of data operators and, overall, ease the usage of the database to the end user.

Chapter 6

Conclusions

The work in this master thesis was on purpose of developing a lightweight database suited both for traditional high-end computers – such as servers – and small mobile/IoT devices. This last set of equipment could benefit from adapted on-disk databases and be able to work more autonomously, with lesser latencies and better bandwidth and making possible new applications. Since this supposes a large task that can lead to several deep ramifications, we have focused on starting by delivering what should be a first version of the core of the database: its buffer and storage managers.

In that sense, we have researched on current state-of-the-art implementations of buffer management layers and some possible improvements. Taking all this knowledge into account, we have designed and implemented a relatively simple database that, although presenting some obvious limitations compared with commercial software, is fully operational. The resulting system is able to manage data ranging from sizes of about a few MB to several GB of space, or even TB. Furthermore, it presents a handful of parameters that can be manually configured to adjust it to the underlying system and be able to obtain the best possible performance. On top of that, we have tried to provide partitioning capabilities alongside a NUMA-aware mode to the buffer pool to allow for better scaling when multiple clients make simultaneous requests. Also, a simple prefetcher has been added which runs using a thread pool structure that is expected to be expanded in the future to cover with other management tasks.

A part from the buffer pool itself, we have written a bunch of validation tests using the *Google Test* library and some more complex ones which try to represent real database operators – as can be a *group by* or a *hash join* – or typical algorithms – a *Breadth-First Search* over a graph stored in the DB – and can be used to stress and evaluate the system with more realistic tasks.

In order to validate our design, we have experimented with it into two different environments: a machine with high computing power and an ODROID-C2 board; which try to be a representation of the types of devices we plan to give support. Our buffer pool has proved to speedup the access to data of a process at a degree of $13X$ in the server and $14X$ in the ODROID-C2. Moreover, we have tested several page size configurations to see which is the real impact of this parameter to the program's execution and have found that is quite low, representing a 0.01% and 0.005% correspondingly.

We have also demonstrated that the buffer pool is prepared to cope with constant multiple requests from different processes; and that is able to manage all that correctly, and in parallel, far faster than if the queries were serialized. For instance, our experiments show that the Sequential Scan workload is able to obtain a $7.68X$ speedup in the server and $2.46X$ in the board just by processing the database requests at the same time from multiple threads. These executions can be further optimized if we enable partitioning and/or NUMA-awareness. In that case, we can achieve, instead, levels of $24.5X$ and $2.7X$ respectively.

However, our prefetching mechanism does not present such good results, incurring into overhead due to the current implementation of the thread pool. Comparatively, a same workload run with a single thread can take up to a 1.8% more time in the server and 8.7% in the ODROID's case if prefetching is enabled. This converts prefetches and, specially, the thread pool into the main existing areas that must be addressed.

All the software described in this work can be freely consulted at our GitHub repository found at `https://github.com/DAMA-UPC/smile`. We encourage anyone willing to try it out to check and test it. Any feedback or possible advice would be appreciated.

Chapter 7

Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, oct 2010.

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*, page 13, 2012.

[3] Boost C++ Libraries. NUMA - 1.65.0.

[4] Richard W. Carr and John L. Hennessy. WSCLOCK - A simple and effective algorithm for virtual memory management. *Proceedings of the eighth symposium on Operating systems principles - SOSP '81*, pages 87–95, 1981.

[5] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[6] Fabio Tudone. What are fibers and why should you care?, 2015.

[7] Hector. Garcia-Molina, Jeffrey D. Ullman, and Jennifer. Widom. Database Systems: The Complete Book. *Interface*, page 1203, 2009.

[8] Google LLC. Page Cache Management for Linux, 2008.

[9] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, may 2016.

[10] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[11] Rishab Jain. Fog computing. *Business & Information Systems Engineering*, (2017):13–14, apr 2017.

[12] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. page 35, 2005.

[13] KDAB. Hotspot - A GUI for the Linux perf profiler, 2017.

[14] Tarig Khalil. Key Success Factors and Dependencies for IoT Ecosystem of SSC. Technical report, ITU-SUDACAD Regional Forum on Internet of Things for Development of Smart and Sustainable Cities Khartoum, Sudan, 2017.

[15] Daniel Kroening. The Impact of Write Back on Cache Performance. *Simulation*, 2000.

[16] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 1075, 2008.

[17] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, pages 743–754, 2014.

[18] Norbert Martínez-Bazan, M. Ángel Águila-Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium on - IDEAS '12*, pages 110–119, New York, New York, USA, 2012. ACM Press.

[19] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6462 LNCS, pages 242–259. Springer, Berlin, Heidelberg, 2010.

[20] B Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2001.

[21] Henrik Mühe. Concurrency in database systems. *Universit, Technische Iii, Informatik Lehrstuhl*, 2014.

[22] Marcus Oppitz and Peter Tomsu. Fog Computing. In *Inventing the Cloud Century*, pages 471–486. Springer International Publishing, Cham, 2018.

[23] Per Olov Ostberg, James Byrne, Paolo Casari, Philip Eardley, Antonio Fernandez Anta, Johan Forsman, John Kennedy, Thang Le Duc, Manuel Noya Marino, Radhika Loomba, Miguel Angel Lopez Pena,

Jose Lopez Veiga, Theo Lynn, Vincenzo Mancuso, Sergej Svorobej, Anders Torneus, Stefan Wesner, Peter Willis, and Jorg Domaschka. Reliable capacity provisioning for distributed cloud/edge/fog computing applications. In *EuCNC 2017 - European Conference on Networks and Communications*, pages 1–6. IEEE, jun 2017.

[24] Raspberry Pi Foundation. Raspberry Pi - Teach, Learn, and Make with Raspberry Pi, 2012.

[25] Hironobu. Suzuki. The Internals of PostgreSQL, 2015.

[26] Martí Torrents, Raúl Martínez, and Pedro López. Comparative study of prefetching mechanisms. 2012.

[27] Mahammad Valiyev. Graph Storage: How good is CSR really? Technical report, 2017.

[28] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, pages 1813–1828, New York, New York, USA, 2016. ACM Press.