Patterns in Data Management

—

Jens Dittrich

December 9, 2015

This print book and ebook were prepared with LaTeX, `tex4ht`, and Ruby scripts employing Nokogiri, written by Armando Fox (and heavily tweaked by Jens Dittrich).

Contents

Preface

When I was an undergraduate student I attended the course *Introduction to Databases*. I quickly concluded that databases are probably the single most boring and dullest topic in the world. Consequently, after a couple of weeks I dropped the course. And I was wondering: is that what computer science is about? Managing data rows and their relationship across tables? Reasoning about the "normal form" of a table? Writing a database "trigger"? Could there be anything less exciting in the universe? Maybe in some parallel universe? Why would such universe exist then? Shouldn't I better switch fields?

And even the term "Database": it reminded me of dusty file cabinets crammed with worn-out cardboards. All of that hidden in dark aisles, somewhere downstairs in a forgotten floor even way below to what people call "the basement", probably observed by some weird librarian who had never seen the light of the day.

Nevertheless, I continued studying computer science, but I focussed on other topics like software engineering. Eventually, I had to write a Diploma Thesis (a predecessor to what is now called an M.Sc. Thesis). And I had to make a decision: which of those two software engineering topics that a software engineering professor suggested to me would I want to work on for the following nine months or so? Well, great question! It seemed, I had finally found something even more boring than databases[1].

Out of a mere gut feeling I headed straight for the database professor's office to ask him about possible Diploma Thesis topics. He let me in and explained two topics to me on the white board. And, surprisingly, they did not sound at all like "managing rows", "determining normal forms of tables" or "writing triggers". Quite in contrast, a whole new world of exciting algorithmic problems opened up. The Diploma topic he suggested would be about designing new algorithms, and comparing them with existing algorithms published in related work at top international conferences. In the months that followed I learned a lot, and I understood how exciting computer science, and in particular *databases*, can be.

With this book I am trying to share my excitement for the field of data management.

Saarbrücken, October 2015

Prof. Dr. Jens Dittrich

[1] This was not so much the fault of software engineering as a field, but of the specific subtopic that software engineering professor suggested. Software engineering, to me, is actually another extremely exciting and useful area in computer science.

Acknowledgments

This book wouldn't have been possible without the support of a great team of students including my Ph.D. students: Stefan Schuh, Endre Palatinus, Stefan Richter and Felix Martin Schuhknecht. They delivered ideas for exercises and helped on the quizzes. They also peer-reviewed some of the material and discussed plans for videos with me. My secretary Angelika Scholl-Danopoulos helped typesetting this book. I am also grateful to the tutors and students of my database systems classes of summer 2014 and winter 2014/15 who were the first to play with the material presented in this book. I would also like to thank my subscribers on youtube for the encouraging positive comments.

How to use this Book

This book is not a standard textbook. This book was written extending and complementing preexisting educational videos I designed and recorded in winter 2013/14. The main goal of these videos was to use them in my flipped classroom "Database Systems" which is an intermediate-level university course designed for B.Sc. students in their third year or M.Sc. students of computer science and related disciplines. Though in general my students liked both the flipped classroom model and (most of) the videos, several students asked for an additional written script that would allow them to quickly lookup explanations for material in text that would otherwise be hard to re-find in the videos. Therefore, in spring 2015, I started working on such a course script which more and more evolved into something that I feel comfortable calling it a book. One central question I had to confront was: would I repeat all material from the videos in the textbook? In other words, would the book be designed to work without the videos? I quickly realized that writing such an old-fashioned text-oriented book, a "textbook", wouldn't be the appropriate thing to do anymore in 2015. My videos as well as the accompanying material are freely available to everyone anyways. And unless you are sitting on the local train from Saarbrücken to Neustadt, you will almost always have Internet access to watch them. In fact, downloading the videos in advance isn't terribly hard anyway. This observation changed the original purpose of what this book would be good for: not so much the primary source of the course's content, but *a different view* on that content, explaining that content where possible in other words. In addition, one goal was to be concise in the textual explanations allowing you to quickly re-find and remember things you learned from the videos without going through a large body of text.

Therefore I came up with a structure for this book where each section, or learning unit if you wish, is structured into:

1. **Material.** This is the primary content I recommend you to study to reach the learning goals of this unit. Typically this material is one (rarely more) short videos. In addition, we provide links to slides as QR-codes. Like this you may easily point your device to this text to open and study the material.

2. **Additional Material.** This is secondary material which you do not necessarily have to study. Yet it might help you in getting a different explanation in other words or an extended explanation of (more or less) the same content. The additional material is split into two parts again. In "Literature" we list material focussing on the actual content of this learning unit whereas. In contrast, in "Further Reading" we list material that is related to this learning unit but goes way beyond the learning

goals. All materials are provided either as bibliographic references or as QR-codes.

3. **Learning Goals and Content Summary.** This is a textual summary of the content of this learning unit. I decided to *not* write it as one flow text, but rather phrase all content as Q&As — we all know people's short attention span these days: what did I write at the beginning of this sentence?

   This Q&A-style serves multiple purposes:

   (a) **Precise Learning Goals.** Each question phrases *one learning goal* of this learning unit. By just reading the questions, you know what you should have learned. After having studied the material, you should be able to answer each question yourself. So after having worked with the material, e.g. a video, you may test your knowledge by hiding the answer and comparing your answer with the one written down.

   (b) **Many Entry Points.** While writing this text my goal was to make each answer concise and as much independently understandable as possible (wherever that was possible, this wasn't always possible as a lot of material builds upon each other). Like that it becomes easier for you to enter the text somewhere in the middle.

   (c) **Different Views.** In the videos I try to connect the dots wherever possible by motivating why a particular technique is important, to which other techniques it is related, and where it may be applied. While writing this text, I felt that here and there I wanted to extend my explanations from the videos a little bit or wanted to provide yet another view on the material in the videos. So in some answers you will see that I grabbed the opportunity to clarify explanations or come up with yet another view on the same content. For instance, in Section 2.3.4, I added a small formalism to define linearization which I believe helps a lot here to understand data layouts. Again, you may perceive this on first sight as adding even more material. However, actually these alternative explanations make your life easier. They are all designed to help you understand the material better and grasp the material faster.

4. **Quizzes.** This is an excerpt of the electronic quizzes we use for my inverted classroom "Database Systems". In that class, every week, I ask my students to study some material. Then they have to answer these quizzes in an electronic lecture tool — we use Moodle for that. Only after that, we meet in class to start working on the weekly exercises.

5. **Exercises.** This is a collection of exercises we use for my inverted classroom "Database Systems". These exercises are taken from the weekly assignments. We change them a bit every year. The students start working on these exercises in the "class", I call it "the LAB". During that time my tutors and me walk through the aisles to consult the students.

Chapter 0

Introduction

0.1 Course Overview and Motivation

0.1.1 The Truth about Databases

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Database Management System

Learning Goals and Content Summary

Why are database systems a vertical topic?

vertical topic

Database systems covers topics from multiple fields. These fields (or topics) are typically treated in separate textbooks and often investigated separately. However, in order to really understand and design a complex system (like a database system), we have to understand all relevant fields (and what is important in those fields for data management) and in addition understand their *interactions*.

Which are those topics?

All systems layers (aka levels): hardware, operating system, software, and even how users interact with the system. In the context of this book we are particularly interested in hardware, data layouts, algorithms, data structures (referred to as indexes in this book), and how the different components interact in a complex system.

Is this book only about database systems or software in general?

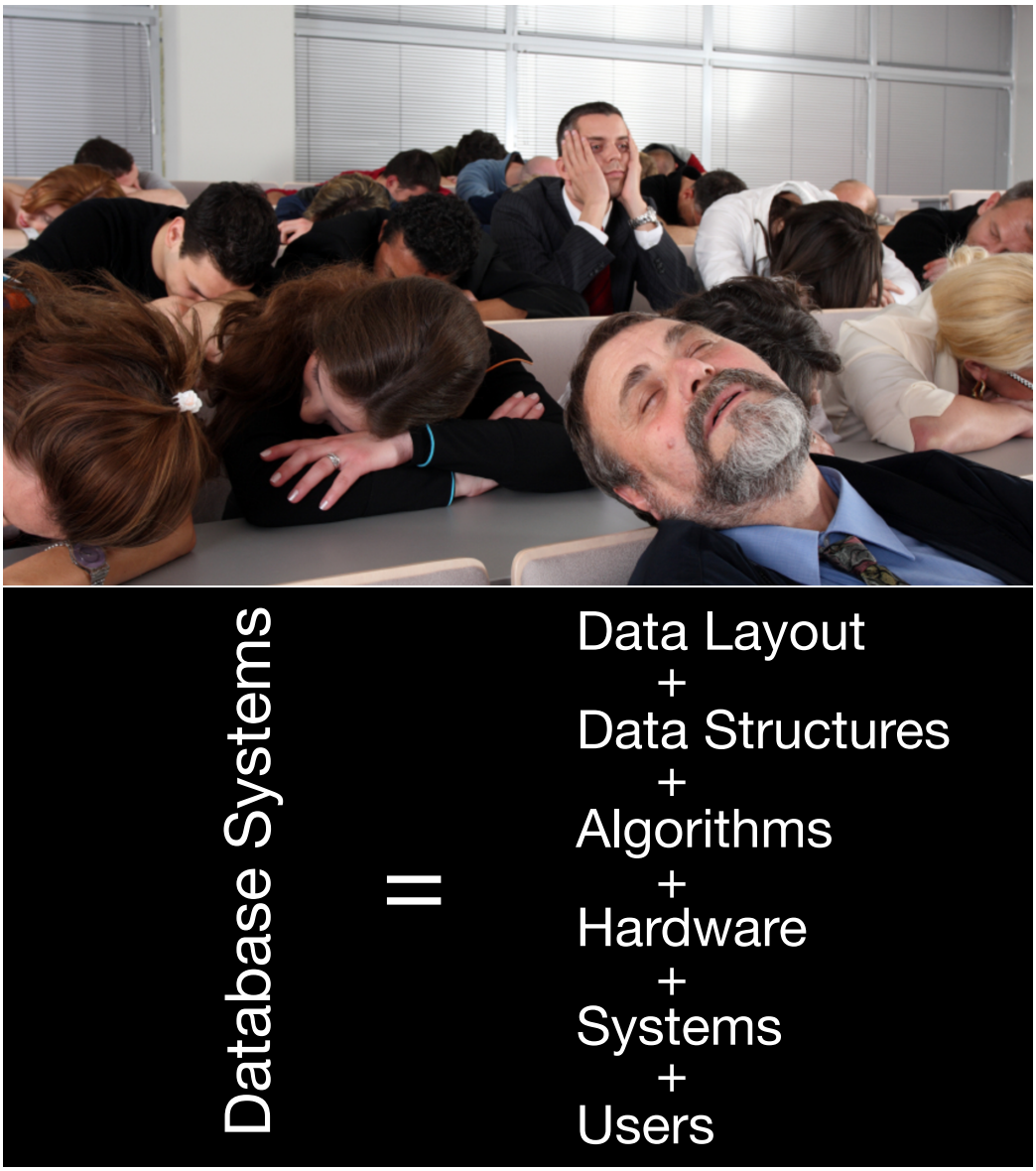


Figure 1: different aspects of database systems

Whether you are using a database management system or not: most of the software artifacts we design manage data in one way or the other. Therefore, the techniques we learn about in this book (and the videos accompanying it) are not only used inside database systems, but may be used in other more general software. The impact of these techniques may be in terms of performance, data consistency or both. This book is designed to teach best practices in data management. This means, we will not focus on special cases, but rather spend most of our time discussing general techniques that have been identified in the past four decades of database research to be extremely useful and efficient. We will also phrase some of those techniques as 'data design patterns' (similar to the standard software design patterns presented by Gamma et.al. [GHJV95]).

0.1.2 Architecture of a DBMS

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

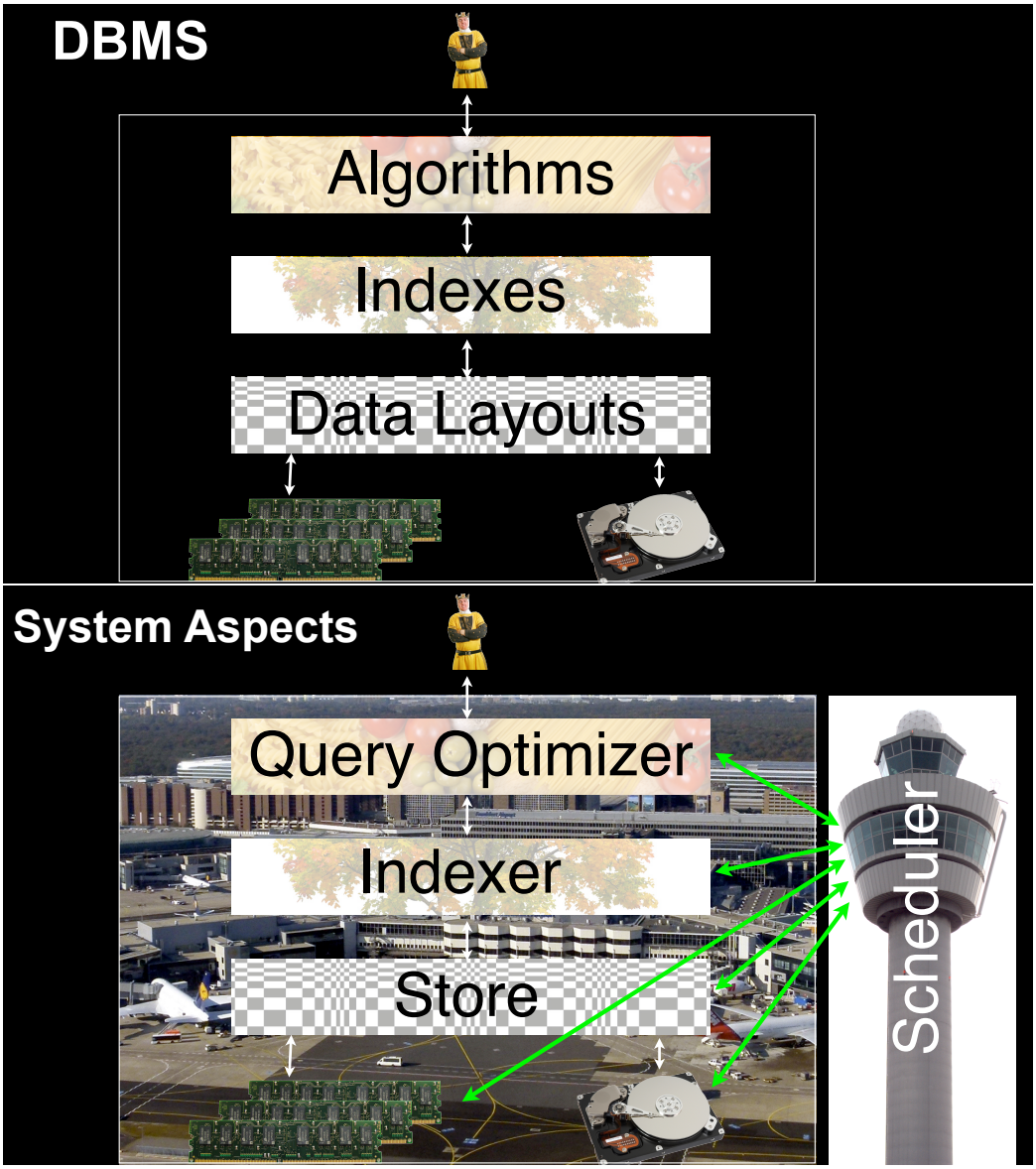


Figure 2: Layers of a database system

What are the different layers of a database system?

layers

The layers are hardware, store (aka storage), indexer, and query optimizer. Hardware may or may not be considered to be part of the database system. Usually, it makes sense to consider it to be part of the system in order to fully exploit the performance

optimizations possible on a particular hardware platform. Some database systems are even sold as a package of software and hardware (aka appliance).

vertical topic

How do they relate to the different vertical topics?

The query optimizer is mostly about algorithms, the indexer mostly about data structures (indexes), and the store mostly about data layouts. So basically each layer in a database system corresponds to one of the vertical topics.

store

indexer

query optimizer

What are the major tasks of store, indexer, and query optimizer?

The main task of the store is to manage fine-granular data items (rather than just files as done by a file system). The indexer provides data structures that allow us to quickly find data items in the store (rather than just scanning through all items available). The query optimizer takes an incoming query (typically an SQL-statement created by an application program) and translates it into an efficient program. SQL is declarative, i.e. it defines WHAT needs to be retrieved. In contrast, the query optimizer has to find out HOW to retrieve that data.

system aspects

What are system aspects of database systems?

Some aspects of the database system are hard to comprehend when investigating one of the layers alone. These 'system aspects' are cross-layer aspects that often need to be treated holistically. A good example for this is scheduling, e.g. what to do if the system is allowed to handle multiple queries/transactions concurrently? The consistency and performance problems implied by this should be handled considering their impact across all layers.

computation

data access

What is the conflict of computation versus data access about? And does this conflict relate to the different layers of a database system?

Traditionally, many courses in computer science focus on the computational aspects of algorithms, e.g. the runtime complexity of an algorithm or data structure which is often modeled along the number of CPU operations. In databases, very often the actual computation time of an algorithm is *not* the most critical aspect of an algorithm or system. Rather, the time it takes to retrieve or store (read or write) data items may have a much higher impact on the overall runtime of a program than the computational effort. Therefore, depending on which layer of a database architecture we are talking about, the effects of data access may outweigh the computational effects. In general, the closer we get to the store, the higher are the data access effects, e.g. particular layouts or random vs sequential access. Vice versa, the closer we get to the query optimizer the stronger are the computational effects, e.g. computational effort of join enumeration, efficiency of query unnesting.

Q&As

1. What can be considered architectural layers of a DBMS?
   (a) Hardware
   (b) BIOS
   (c) Operating system
   (d) Store

(e) Indexer

(f) Query optimizer

(g) Application

(h) User

2. The part of a DBMS that is concerned most with data layouts is:

   (a) The query optimizer

   (b) The indexer

   (c) The scheduler

   (d) The store

3. The part of a DBMS that is responsible to determine how to compute results to queries:

   (a) The store

   (b) The indexer

   (c) The scheduler

   (d) The query optimizer

4. The part of a DBMS that provides physical organizations speeding up selective access to tuples:

   (a) The store

   (b) The indexer

   (c) The scheduler

   (d) The query optimizer

5. A physical design advisor is used for:

   (a) Designing high-performance database hardware

   (b) It is actually a job role for database administrators.

   (c) Tuning the knobs of a DBMS to achieve better performance

6. Which component is coordinating the different actions inside a DBMS?

   (a) The user

   (b) The physical design advisor

   (c) The scheduler

   (d) The query optimizer

7. The layer of a DBMS that is focusing mainly on data access and not so much on computation is:

   (a) The indexer

   (b) The query optimizer

(c) The store

(d) The scheduler

Exercise

Discuss:

(a) What additional layer could be considered part of the database system? What would be the advantage of considering that layer?

(b) What if you implement the DBMS as shown in the slides (and bypass the layer discussed in (a)? List at least one concrete advantage of doing this.

(c) Some vendors implement parts of the DBMS in hardware (be it configurable FPGAs or as real chips). Examples include Netezza (acquired by IBM). What would be the advantages/disadvantages of this approach?

0.2 History of Relational Databases

0.2.1 A Footnote about the Young History of Database Systems

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

Why would it make sense to use a database system anyway?

Almost every program needs to manage data at one point or another. So, why not bundle the data managing functionality in a separate software component which can then be developed, tested, and maintained separately. This has the advantage that software developers do not have to constantly re-invent the wheel in terms of data management.

How did the development of database systems start?

physical data independence

Since the early days of computer science people tried to come up with efficient data managing solutions. The early approaches like navigational and hierarchical database systems had the problem that they did not support real physical data independence, i.e. the developer had to know how the data was organized in the database system in order to be able to phrase queries. In other words: the WHAT (which data do I want?) and the HOW (how is that data actually retrieved by the system?) were not clearly separated. In the early 70s, this problem was solved with the relational model, invented by E. Codd.

relational model

What was a major change introduced by the relational model?

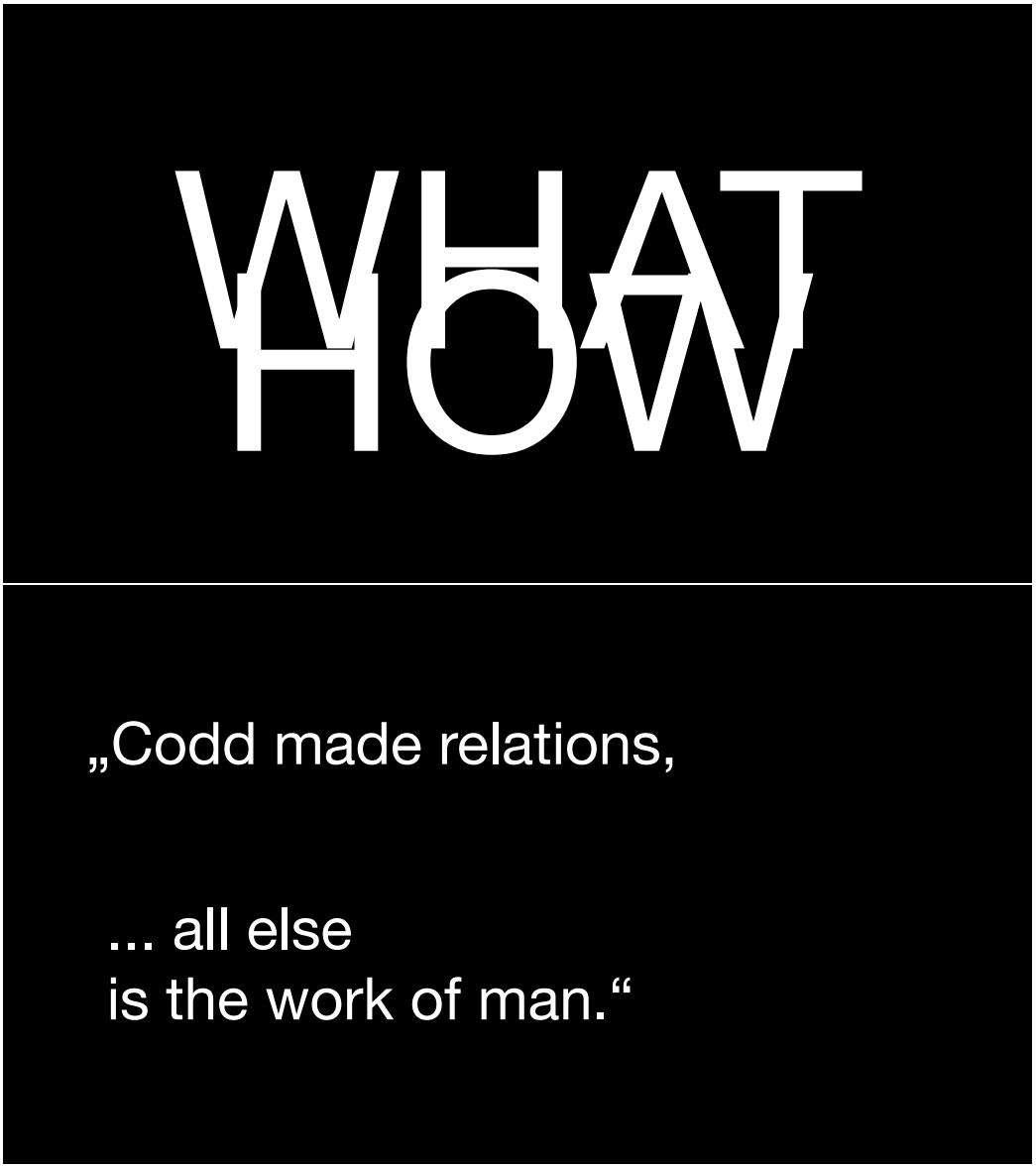


Figure 3: from non-relational to relational database systems

The major change was providing physical data independence, again: this separated the WHAT from the HOW. However note, that this separation is not fully preserved by modern database systems. For instance, still inside relational database management systems, the database administrator (DBA) has some influence on the HOW part by deciding which indexes to create, e.g. `CREATE INDEX...`, or how to assign data to physical storage, e.g. through tablespaces.

database history

What were major developments in database history?

The development of systems implementing the relational model. These systems are sometimes referred to as relational database management systems (RDBMS). Then SQL and its standardization, several extensions to SQL and database systems in general like object orientation, parallel databases, analytical databases (OLAP), support for XML and JSON, data stream management and column stores. Since 2010, many database tech-

niques have been revisited (again) in the context of 'big data' and so-called NoSQL systems. It is also worth noting that the field of database systems, even though relational systems have been developed since the 70ies, is still facing major innovations. This is also underlined by the high number of new start-ups and acquisitions of start-ups through big, established database companies every year.

Q&As

1. What is the main distinguishing feature of pre-relational and relational database management systems?
   (a) XML support
   (b) Physical data independence
   (c) Big data support
   (d) Object orientedness
2. What was the first commercial relational DBMS?
   (a) Oracle Database
   (b) IBM DB2
   (c) Informix
   (d) Knoppix
   (e) IBM System R
   (f) MySQL
3. What does physical data independence mean?
   (a) The logical organization of the data is independent from the database schema.
   (b) The physical organization of data is independent from the database schema and the data in it.
   (c) Creating multiple copies of the database to speed-up query processing.
4. Which of the following types of DBMSs are the most important ones today?
   (a) Relational
   (b) Object-oriented
   (c) Object-relational
5. Having read-mostly, complex queries is also referred to as:
   (a) Big data
   (b) OLAP
   (c) Object-oriented DBMS
   (d) OLTP
6. What is an appliance?
   (a) A bundle of software and hardware

(b) A storage server

(c) A DBMS

(d) A cloud computing software

0.2.2 Relational Database — A Practical Foundation of Productivity

Material

Literature:
[Cod82], Sections 0 to 4

Additional Material

Literature:
[Bac73]

Learning Goals and Content Summary

What was the problem with data management in the 60s?

There was no sharp distinction between the logical view on the data (the WHAT part) and its physical representation (the HOW). Hence, the programmer had to know how data was stored rather than focussing on the what part.

associative addressing

What is associative addressing in the context of a database system?

Associative addressing overcomes positional addressing. In positional addressing data items are identified by their position. In contrast, in associative addressing, data items are identifiable solely based on the triple (relation name, primary key, attribute name).

data model

What is a data model?

Codd's defines a data model to contain at least three components:

1. a *structural part*, e.g. domains, relations, attributes, tuples, candidate and primary keys,
2. a *manipulative part*, e.g. algebraic operators like select, project, and join which transform input relations into output relations, and
3. a *integrity part*, e.g. integrity constraints which impose restrictions on the data instances that may be represented in the structural part.

relational model

column

row

schema

In the relational model, does the order of the columns or rows in a schema matter?

No, the order of columns in a relation is irrelevant. Obviously, the same applies to the order of rows. This is a feature of associative addressing.

What is the primary goal of relational processing?

A goal of relational processing was loop-avoidance which was assumed to boost programmer productivity. This is in so much true that the actual SQL-statement is loop-free and

declarative, i.e. the programmer does not have to write down loops, for instance to specify a join operation. However note that as soon as the result of an SQL-statement is fetched from a database into a programming language, e.g. through JDBC, the result is typically treated in a loop again.

relational algebra

end-users

Is relational algebra intended to be used as a language for end-users?

No, it is not. It is meant as a starting point to design appropriate sublanguages supporting these set operations. In a modern database system (as of 2015), (enriched) relational algebra is typically used as an intermediate language in particular for query processing, see Chapter 4, and query optimization, see Chapter 5.

Q&As

1. In the late sixties the DBMS failed to boost productivity due to the lack of
   (a) separation of logical and physical views of the data
   (b) support for views
   (c) support for stored procedures
   (d) set processing commands
   (e) iterative processing commands
2. Codd proposed to access data
   (a) by positions
   (b) by associative addressing
   (c) with indexes
   (d) in text format
3. Which of the following are not part of the relational model:
   (a) domains, relations, attributes
   (b) algebraic operators
   (c) integrity rules
   (d) indexes

Chapter 1

Hardware and Storage

1.1 Storage Hierarchies

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Literature:	
[LÖ09], Memory Hierarchy	
[LÖ09], Main Memory	
[RG03], Section 9.1	

Further Reading:	
[LÖ09], Storage Management	
[LÖ09], Storage Security	
[LÖ09], Write Once Read Many	
W disk and SSD performance charts	
W RAM performance charts	
[PH12], Section 5	

Learning Goals and Content Summary

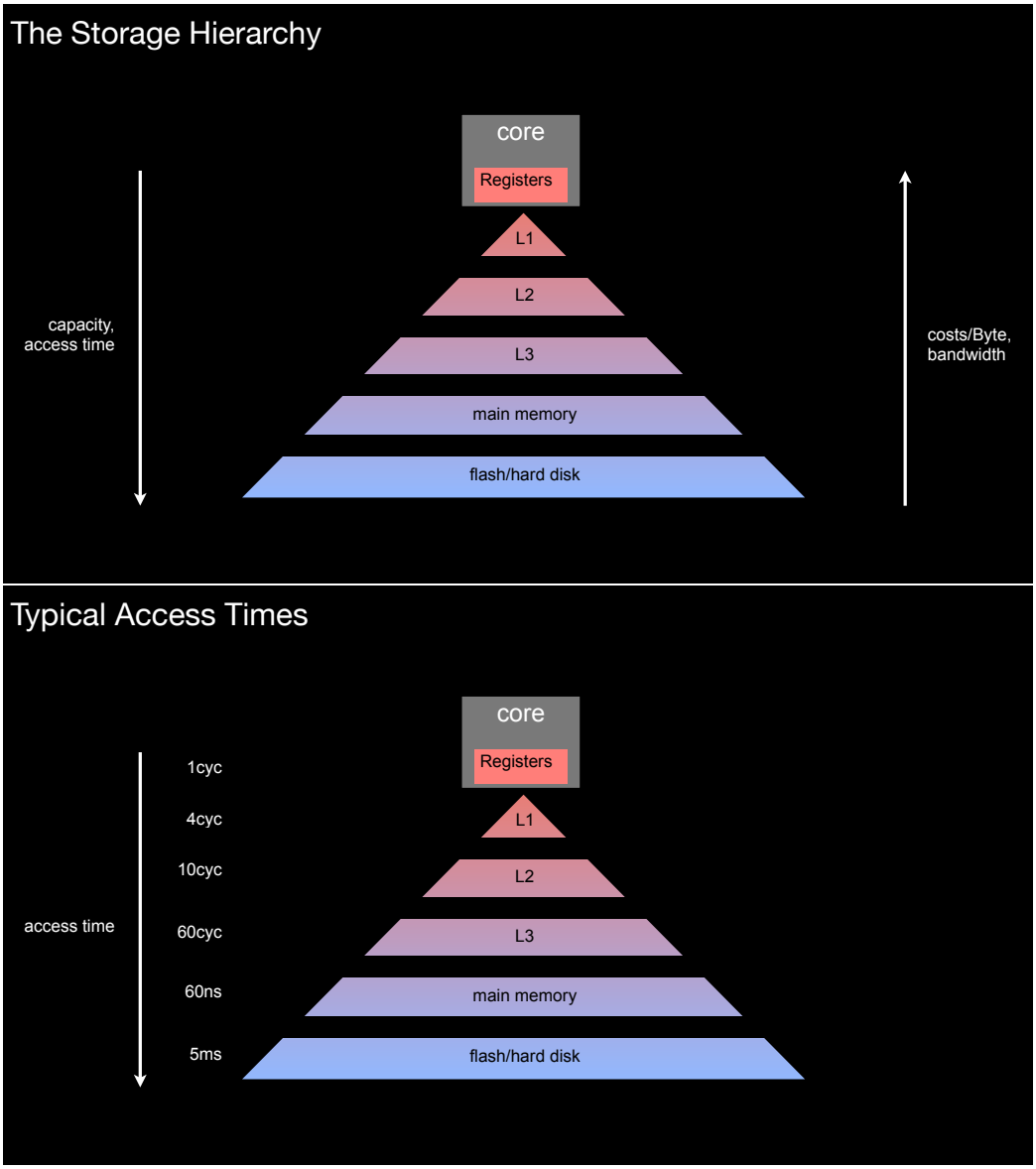


Figure 1.1: A simple storage hierarchy and typical access times

memory

What are the properties of ideal computer memory?

It would have unlimited capacity and bandwidth, zero random access times. It should be for free and persistent. In addition, it should not trigger any read errors whatever happens.

storage hierarchy

What is the core idea of the storage hierarchy?

The core idea of a storage hierarchy is to approach ideal memory in terms of performance however with dramatically reduced costs.

storage capacity

access time

computing core

What is the relationship of storage capacity and access time to the distance to the computing core?

The closer a storage layer gets to the computing core, the faster random access, at the

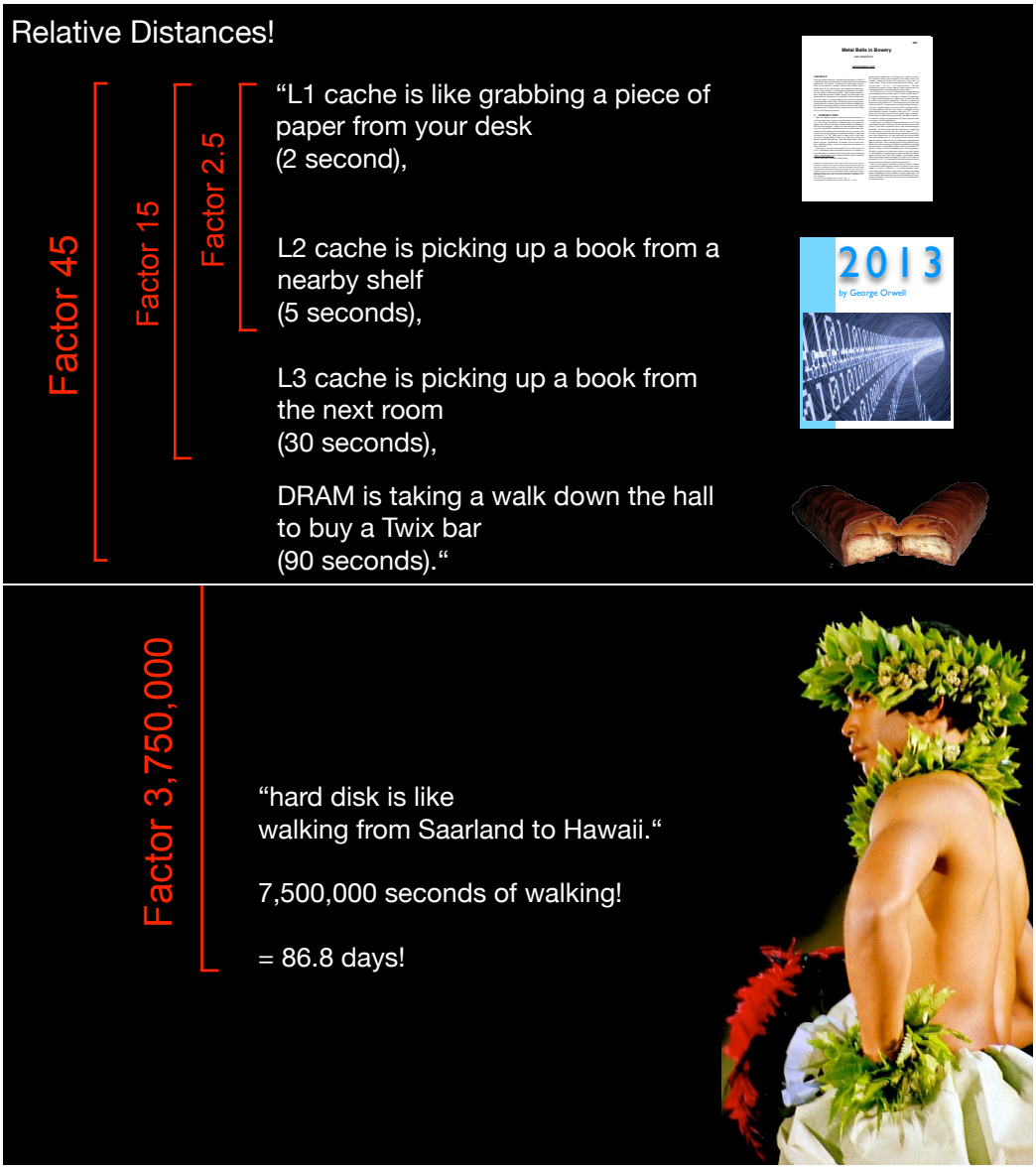


Figure 1.2: Access times translated to a real-life scenario: picking up something from your desk vs walking to Hawaii

same time the smaller the storage capacity.

What is the relationship of costs/ bandwidth to the distance to the computing core?

costs

bandwidth

The closer a storage layer gets to the computing core, the faster sequential bandwidth, the more expensive the memory (when calculated per Byte).

What are typical access times of the individual storage layers? How does this translate to relative distances?

The term “typical access time” indicates already some vagueness here. The actual access time depends a lot on the type of the device. As of 2015 it is fair to assume that accessing L1 takes ∼4 cycles, L2, ∼10 cycles, and L3 ∼60 cycles. Following memorybenchmark.net the latency for DDR3 and DDR4 RAM modules are in an interval of 15–116 ns. The video assumes slightly older RAM with 60 ns latency. It is fair to assume that modern

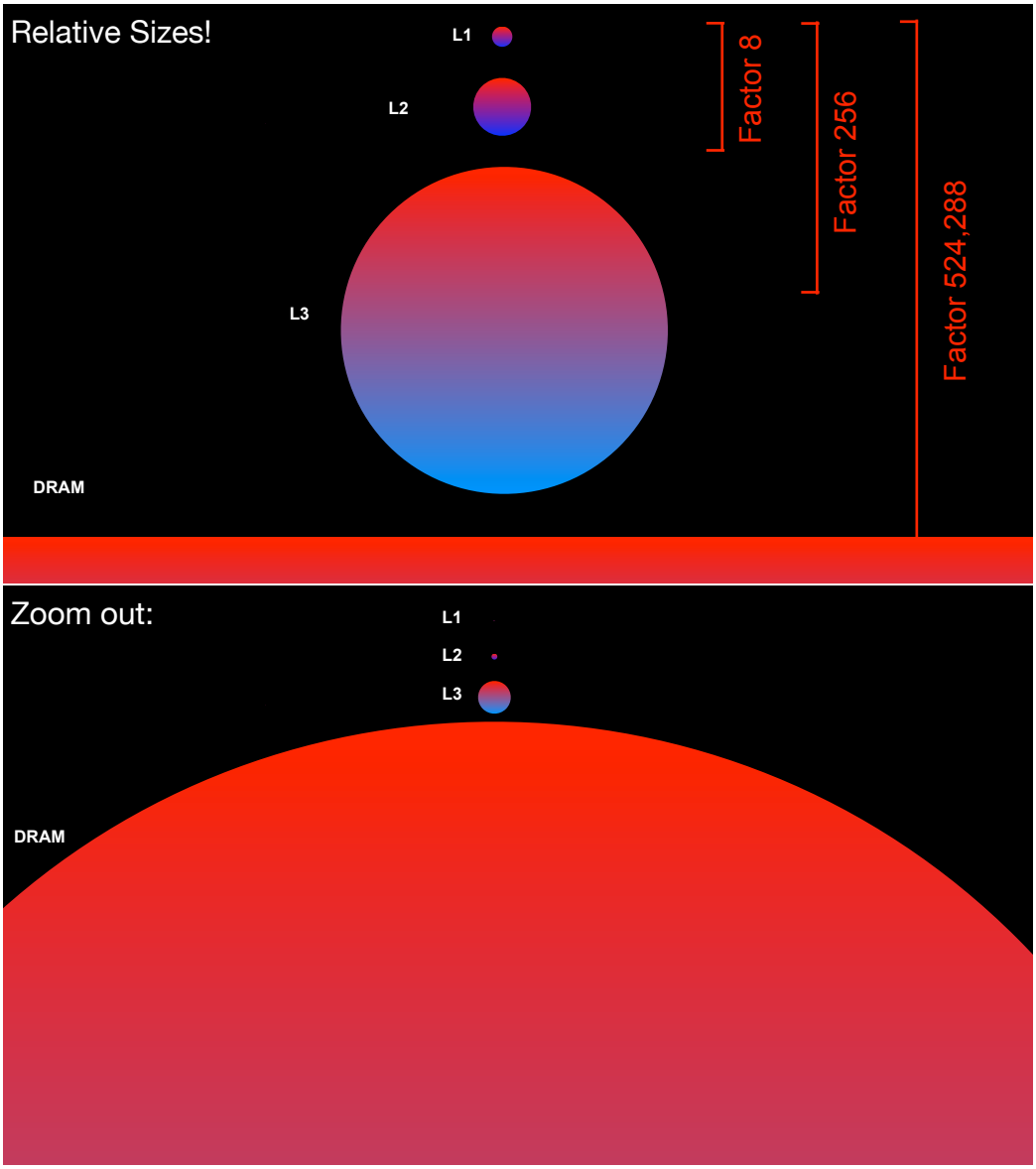


Figure 1.3: Relative Sizes of the different caches in a storage hierarchy: L1–L3 caches vs DRam

RAM has about ∼20ns latency only.

storage layer
storage level

What are typical sizes of the different storage layers (aka storage levels)? How do they translate to relative distances?

Again, this is a parameter that improves quickly every year. It depends on your CPU, the number of CPUS and the prize you pay for your server. As a rule of thumb as of 2015, as we pick a typical CPU, say an Intel Haswell L1 is 64 KB per core (32 KB for data, 32 KB for code), L2 256 KB per core, and L3 may be in-between 8–25MB (shared on the CPU). DRAM is measured in the hundreds of GBs. Having a server with 1 TB of RAM is not uncommon and affordable even for small companies. Due to these numbers most relational database systems fit comfortably into DRAM.

What are the tasks of each level of the storage hierarchy?

Localization of data objects, caching of data from lower levels (typically inclusion), im-

plementation of data replacement strategies and writing of modified data (possibly synchronization with other caches) different strategies (write through and write back).

cache

How would we use a storage hierarchy to cache only reads?

If you use the write through strategy, write operations are never cached. This means, the storage hierarchy caches only data w.r.t. read but not to writes.

How do we use a storage hierarchy to cache both reads and writes?

If you use the write back strategy, write operations are cached as well. This means, the storage hierarchy caches data w.r.t. both read and writes. Extra care has to be taken to persist changes performed by write operations.

inclusion

What is inclusion?

Inclusion means that data available on a higher level is a subset of data on a lower level. This does not have to hold strictly. For instance, main memory typically includes data structures that are not necessarily existent on disk.

data replacement strategy

What is a data replacement strategy?

Whenever a storage layer has to store some data, yet there is no free slot available for storing that data, some data has to be evicted from that storage layer. The data replacement strategy decides which data item to remove. Obviously it has quite some effect on the performance of a storage hierarchy.

Q&As

1. What level of the storage hierarchy can be accessed the fastest?
   (a) L1 Cache
   (b) Memory
   (c) Disk
   (d) Register
2. What of the following levels of the storage hierarchy can typically store the most data?
   (a) L1 Cache
   (b) Memory
   (c) Disk
   (d) Register
3. The L2 Cache includes the L1 Cache. Does the hard disk include the whole main memory?
   (a) Yes, if we strictly apply the inclusion property.
   (b) No, inclusion forbids this.
   (c) Often, but in practice inclusion is not strictly applied between these layers.
4. How does the write through strategy on any level of the storage hierarchy work?

(a) A modified item is written back to the lower level of the storage hierarchy just before it gets evicted.

(b) Every modification is immediately written back to the lower level in the storage hierarchy.

(c) Every modified item is written immediately to disk.

5. How does the write back strategy on any level of the storage hierarchy work?

(a) A modified item is written to the lower level of the storage hierarchy just before it gets evicted.

(b) Every modification is immediately written back to the lower level in the storage hierarchy.

(c) Every modified item is written back to disk after it has been evicted.

1.1.1 The All Levels are Equal Pattern

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

What is the central observation of the All Levels are Equal Pattern?

No mater what layer of the storage hierarchy we are talking about, the techniques and algorithms used a very similar. Those algorithms may need some tweaking, yet their central idea is often the same.

storage hierarchy

What does All Levels are Equal mean for practical algorithms? How can I exploit it to solve a problem on a specific layer of the storage hierarchy?

If you have an algorithm X that works for one specific layer Y of the storage hierarchy, you should consider adapting X to work for storage layer Y.

pattern

How to misunderstand the All Levels are Equal pattern?

You misunderstand it if you take it literally. The statement "All levels are equal" is incorrect in the strict sense that the exact same techniques can be used, however the statement is correct as a high-level observation: the core ideas of the algorithms are often the same, just the granule, i.e. the layer(s) an algorithm operates on is exchanged. Compare also the fractal design pattern in Section 2.3.5.

1.1.2 Multicore Storage Hierarchies, NUMA

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Further Reading:
[KSL13]

Learning Goals and Content Summary

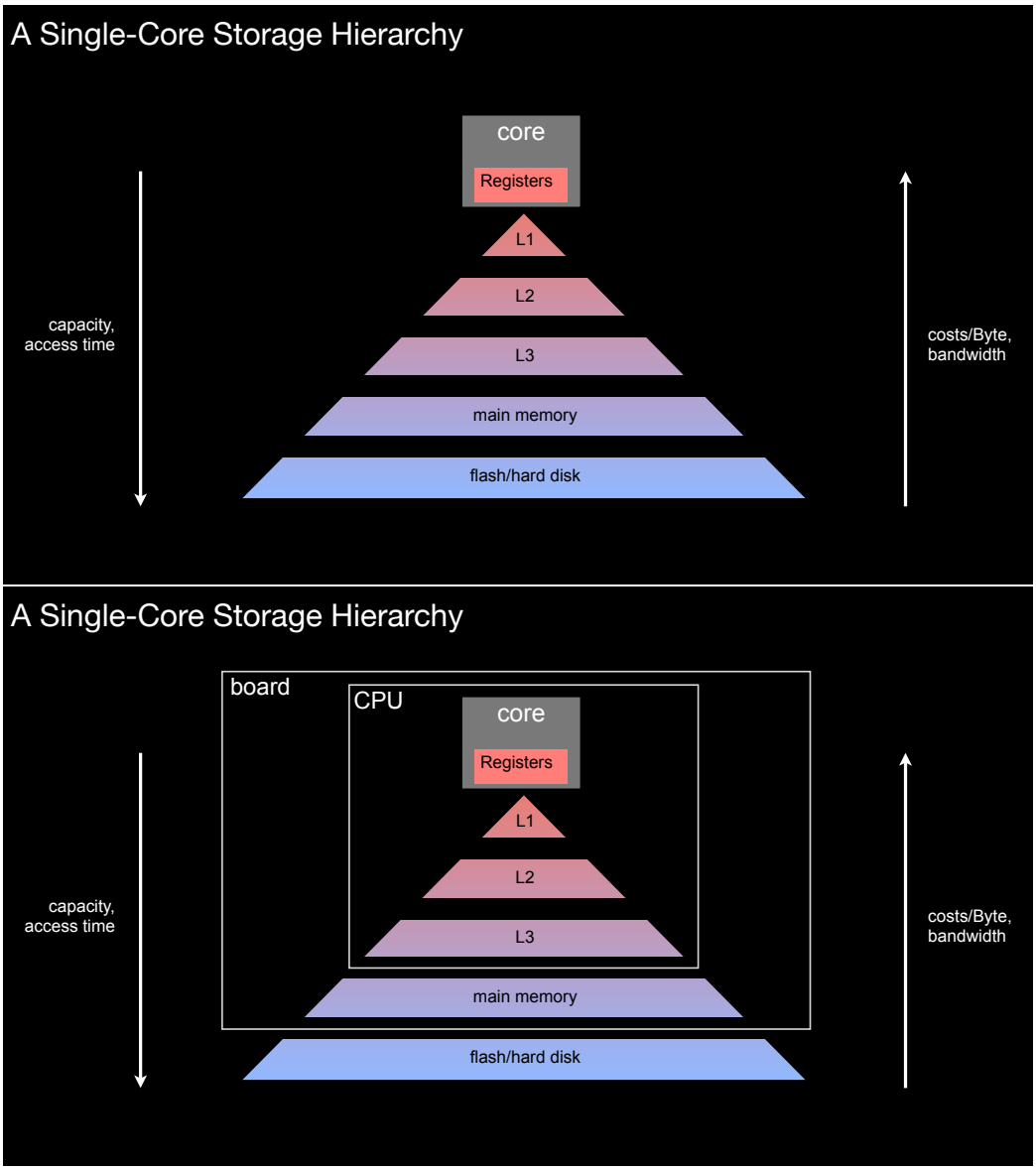


Figure 1.4: A single-core storage hierarchy and how it is mapped to a CPU and the board

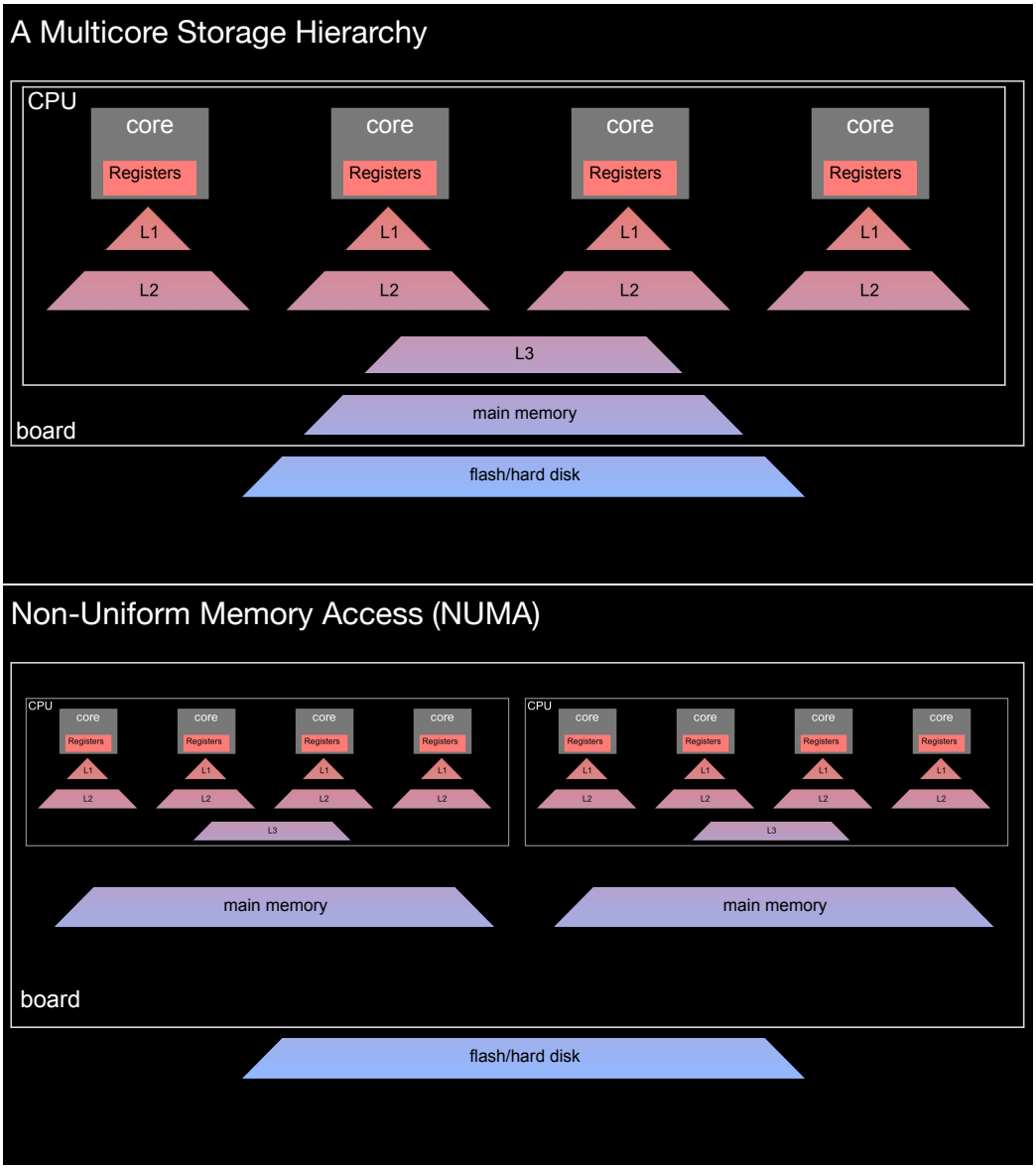


Figure 1.5: A multicore storage hierarchy vs NUMA

core

How are the two terms core and CPU related?

CPU

A CPU may contain several cores (separate arithmetic logical units).

multicore storage hierarchy

What does a typical multicore storage hierarchy look like? What is different compared to the storage hierarchy?

It looks similar to a standard storage hierarchy except that each core has separate L1 and L2 caches.

L1

Why isn't L1 shared as well among multiple cores?

For two major reasons: the access times would go down (for physical reasons) and too many locking conflicts would occur.

Non-Uniform Memory Access

What is a Non-Uniform Memory Access (NUMA) architecture?

NUMA

In NUMA, main memory is not considered as one uniform unit but rather split into

regions. Each region is typically attached to one CPU. This implies that a CPU has NUMA-local memory as well as NUMA-remote memory. Access to NUMA-local memory is slightly faster than accessing NUMA-remote memory. Though the term NUMA is typically used w.r.t. memory, similar effects may happen on any layer of the storage hierarchy.

multicore architecture

What is the difference of NUMA compared to the multicore architecture?

In a multicore architecture we use local L1 and L2 caches in NUMA we add local main memories. We could also say: in a multicore architecture we have non-uniform L1 and L2 access as access to non-local L1 and L2 is slightly more expensive.

DRAM

What does NUMA imply for accesses to DRAM?

Memory accesses to non-local memory (be it main memory or caches or whatever) is slightly more expensive. This should be factored in into algorithm design. However, be aware that the overheads of remote-access may be small (depending on your concrete application).

How again does this relate to The All Levels are Equal?

It is fair to say that a main memory system with different RAM banks as well as a shared-nothing system are both NUMA, the former w.r.t. RAM, the latter w.r.t. disks. So again, if a phenomenon exists for one layer of the storage hierarchy, it very likely also exists for other layers.

Q&As

1. Which components are replicated in a multicore architecture?
   (a) L1 Cache
   (b) L2 Cache
   (c) L3 Cache
   (d) Registers
   (e) Hard Disk
2. Which components are replicated in a multi socket (NUMA) architecture?
   (a) L1 Cache
   (b) L2 Cache
   (c) L3 Cache
   (d) Registers
   (e) Hard Disk
   (f) Motherboard

1.2 Storage Media

1.2.1 Tape

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Literature:	
[LÖ09], Storage Devices	

Learning Goals and Content Summary

tape

What are the major properties of tape?

Tape has (very) slow random access due to winding (approximately 100 sec) and high bandwidth (about 100MB/sec). It is good for archival storage. Tape storage is typically available as separate tape cartridges which are accessed by a tape drive.

Why would tape still be important these days?

It is still heavily used for archives and backups, but it will get more and more replaced by hard disks.

tape jukebox

access time

What is a tape jukebox? How does it work? What does this imply for access times?

A jukebox is an automated library of tapes, a tape drive and a robot. The robot fetches tapes and places them in the tape drive. This adds some additional delay to the random access time. Therefore, random access on a tape cartridge in a jukebox is the sum of the times for fetching the tape cartridge, placing it in the tape reader, *and* winding the tape to the right position.

Q&As

1. Why are there still tape drives out there?
   (a) Fast random access
   (b) Cheap space for archival purposes
   (c) Super high bandwidth
2. Tape Jukeboxes allow for faster random access compared to a single tape reader.
   (a) False
   (b) True
3. Tape Jukeboxes allow for near-line access to an archive. Why is that the case?

(a) The robot works 24/7.

(b) The robot replaces the librarian who is expected to be slower.

1.2.2 Hard Disks: Sectors, Zone Bit Recording, Sectors vs Blocks, CHS, LBA, Sparing

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	
[LÖ09], Disk	
Video, a Hard Disk in action	

Further Reading:	
[LÖ09], Active Storage	

Learning Goals and Content Summary

What are the major components and properties of hard disks?

Disks are at their core mechanical devices: they are build around a stack of magnetic platters rotating on a spindle, platters can be read from/written to on both sides.

Why would hard disks still be important these days? **hard disk**

in Industry is slow-moving, major db products still disk-based. Disks are also required for datasets exceeding main memory. In addition, many legacy database systems are still in use. Even for main-memory centric systems, it is also important to persist changes for durability and recovery purposes (like in ARIES recovery).

What is a platter? **platter**

A platter is a magnetic disk used for storage. Platters are stacked on a spindle which rotates with constant speed, e.g. several thousand rotations per minute.

What is a diskhead and a disk arm? **diskhead**

disk arm

A diskhead reads from and writes to exactly one side of one platter. A disk arm contains one disk head for each side of each platter. The disk arm may be moved and positioned on different cylinders. Only one of those disk heads may be active at any as the position of the arm must be fine-tuned for each side of a platter separately. This means, within a particular cylinder there is still some fine-tuning involved to position the currently active disk head.

How do tracks and cylinders relate? **track**

cylinder

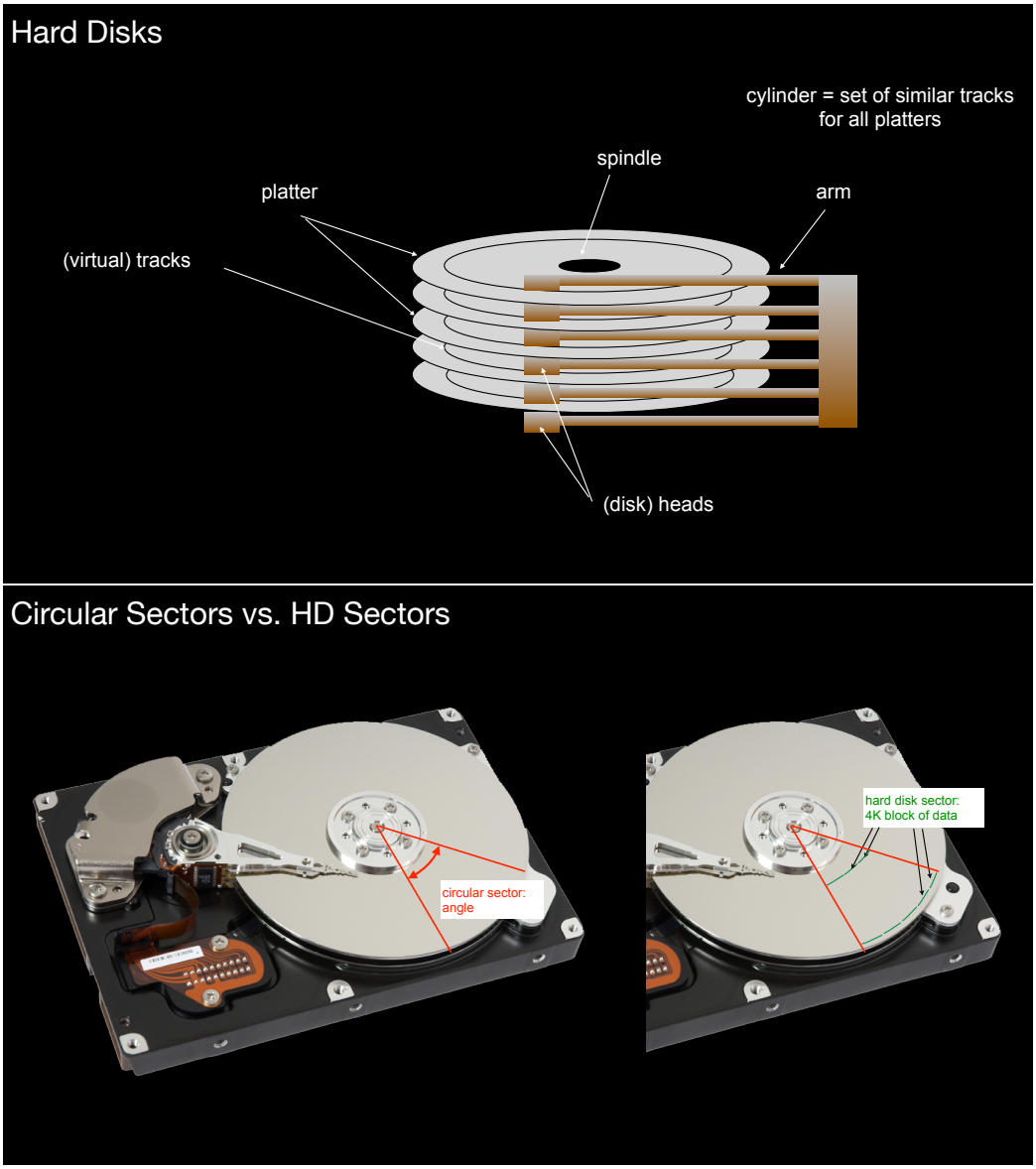


Figure 1.6: The principal architecture of a hard disk; circular vs HD sectors

track on a platter contains all points having the same distance to the center of the platter, i.e. it corresponds to a circle with that radius around the center. The tracks of all platters having the same distance to the center are grouped into a cylinder.

circular sector

HD sector

What is the difference of a circular sector from a HD sector?

A circular sector denotes the surface of the disk enclosed by two radii and an arc. The size of the sector is defined by the angle among the two radii. In contrast, a hard disk sector is a consecutive sequence of bits on any track. Typically, a hard disk sector has a fixed size of 512 Bytes or bigger.

zone bit recording

What is zone bit recording?

The circumference of a track grows with its radius. Therefore the larger the radius the more HD sectors fit on a track. Tracks containing the same number of HD sectors are grouped into a zone. In other words: a zone is a set of adjacent tracks having the same

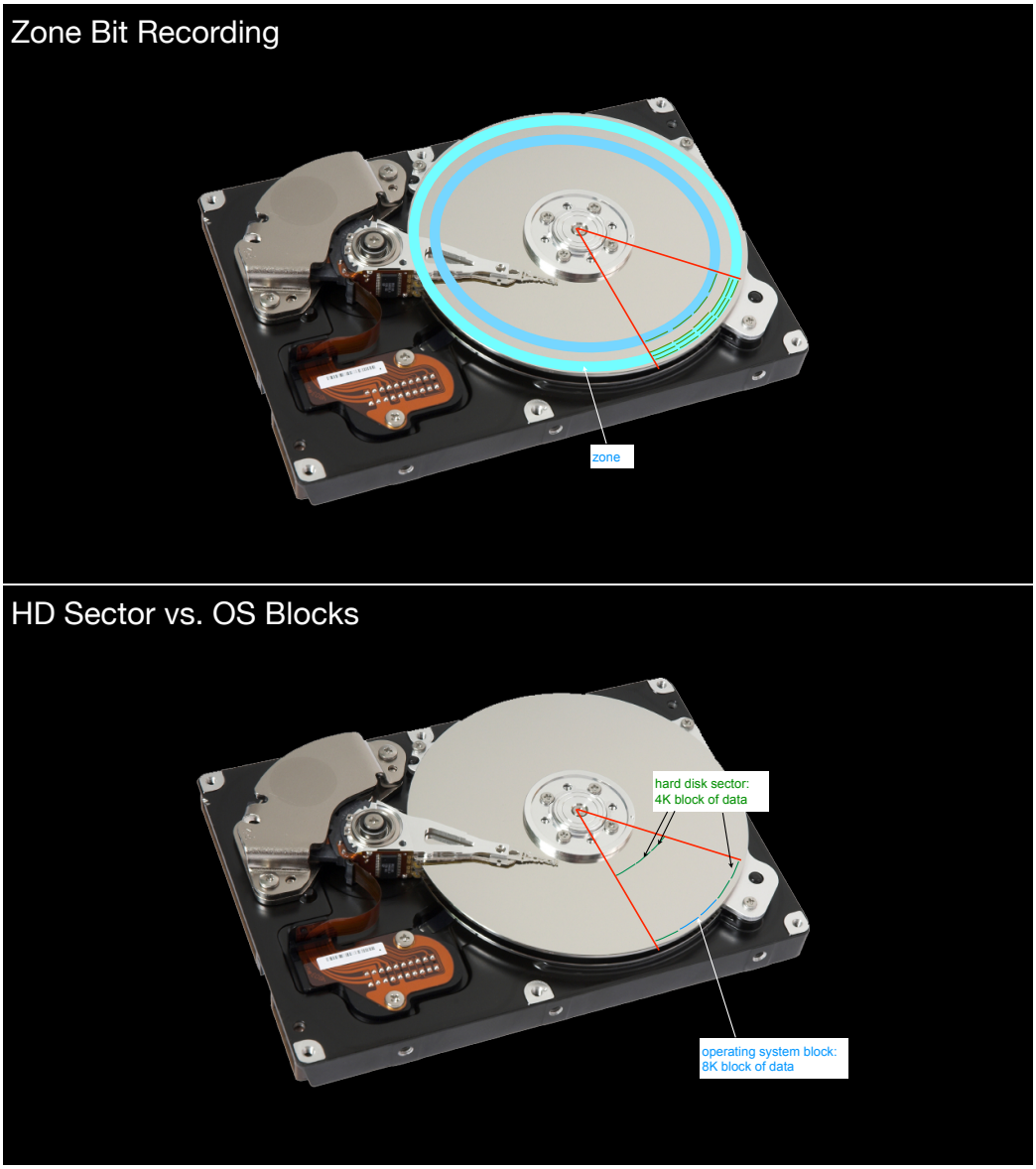


Figure 1.7: Zone bit recording; HD sectors vs operating system blocks

number of HD sectors.

What does this imply for sequential access?

sequential access

Assuming a constant rotation speed of the platters, sequential read and write performance is higher the closer the track is positioned to the edge of the platter.

Where are self-correcting blocks used?

self-correcting block

Some drives use internal error-correction codes to be able to detect erroneous blocks and faulty reads, i.e. the data read is different from what was written before.

What is the difference of HD sectors and operating systems blocks?

operating systems block

Blocks used by the operating system are typically larger, i.e. a multiple, of a HD block.

What is physical cylinder/head/sector-addressing?

physical cylinder/head/sector-addressing

This is an old addressing scheme for hard disks exposing the physical properties of the disk. Blocks are addressed using a compound key of cylinder, head, and sector.

logical cylinder/head/sector-addressing

What is logical cylinder/head/sector-addressing? How does it relate to the former?

This addressing scheme has a similar interface as in physical cylinder/head/sector-addressing. However, block addresses are mapped to different physical positions by the hard disk controller (a small computational device which is part of the disk, see also Section 1.2.4). This implies the device mimics a certain architecture, i.e. X cylinders, Y heads, and Z sectors. In reality, that drive may have completely different physical properties.

logical block addressing

What is logical block addressing?

In LBA a simple integer key domain is used for addressing blocks. As these addresses are mapped to physical positions by the hard disk controller (just like in logical cylinder/head/sector-addressing), there is no need to expose an artificial cylinder/head-/sector to the interface anyway.

sparing

What is sparing?

Erroneous blocks may be remapped to different physical locations by the hard disk controller.

sequential access

What does this imply for a sequential access?

It increases the likelihood of random accesses (yet this remapping shouldn't happen too often).

Q&As

1. Why are hard disk systems in use today?
   (a) Durability (from ACID)
   (b) To provide enough storage space for very large datasets
   (c) Many legacy systems are still disk-based.
2. If a disk rotates with 15,000 RPM (rotations per minute), how long does it take to rotate exactly once?
   (a) $t_{rot} = 15,000/3600sec$
   (b) $t_{rot} = 1/(15,000/60)/3600sec$
   (c) $t_{rot} = 1/(15,000/60)/60sec$
   (d) $t_{rot} = 1/(15,000/60)sec$
3. What zone has the highest transfer rate?
   (a) They are all equal.
   (b) The inner zone has the highest transfer rate.
   (c) The outer zone has the highest transfer rate.
4. Which of the following addressing methods are supported by hard disk controllers?
   (a) physical addresses

(b) Logical Block Addressing

(c) Logical CHS

5. Which of the following are true?

   (a) A track consists of spindles.

   (b) A platter has several tracks.

   (c) A spindle is connected to several platters.

   (d) A track has several sectors.

   (e) There is a separate arm for each cylinder.

6. Which of the following are true?

   (a) OS blocks are always the same as HD sectors.

   (b) OS blocks can be composed of several HD sectors.

   (c) HD sectors always contain several OS blocks.

7. How many HD sectors are contained in a circular sector?

   (a) That depends on the angle and the track.

   (b) An HD sector is a synonym for circular sector.

Exercise

Consider a disk with 2,000 cylinders where the cylinders are numbered from 0 to 1,999 (numbering start from the outermost track). For simplicity, let's assume that a track on cylinder i has $50 - \lfloor i/100 \rfloor$ blocks per track, 5 double-sided platters, and a block size of 4096 bytes. Suppose that a file contains 1,000,000 records with 200 bytes each where no record is allowed to span more than one block.

1. How many zones does this disk have?

2. How many records fit into one block?

3. How many blocks are required to store the entire file? If the file is arranged sequentially on disk (filling up one cylinder after the other), how many cylinders are needed **(when starting storing the file in the outermost zone)**?

4. How many records of 200 bytes can be stored using this disk?

5. Assume that the time to move the arm between 2 tracks is at least 2 ms whereas the head switch time is 1 ms, the average seek time is 6 ms, the disk platters rotate at 10,000 rpm: how much time is required to read the entire file sequentially? Further assume that the data is laid out on disk with head skew and track skew in mind.

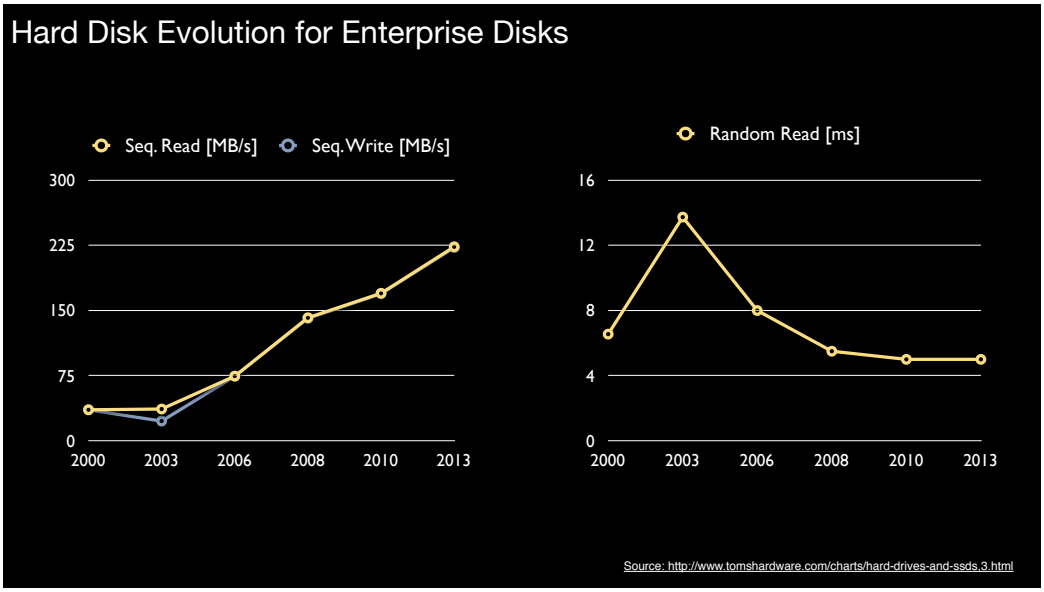


Figure 1.8: Evolution of sequential read and writ bandwidth and random access over time

1.2.3 Hard Disks: Sequential Versus Random Access

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

random access

What exactly is a random access and what are its major components?

If we read from or write to a sector that requires the disk drive to reposition its disk arm, we call this type of access a random access. A random access on hard disk has two major components: 1. the actual arm movement to position a particular disk head on the right track, and 2. rotational delay, i.e. we have to wait that the HD sector are interested in becomes available under the disk head. Though we should also count in the costs for transferring the actual block, the random access time is dominated by these two components.

sequential access

What is a sequential access? and how do we estimate its costs?

If we read from or write to a sector that does not require the disk drive to reposition its disk arm, we call this type of access a sequential access. A sequential access initially has the same costs as a random access (if the disk arm is not already in the correct position). After that it the costs are dominated by the transfer time. Ideally, a sequential scan will first read/write all HD sectors on the same track. Then it will switch to another platter on the same cylinder. It will switch through all platters until all platters for this cylinder have been considered. Only after that it will switch to an adjacent cylinder. Switching disk heads takes some time (about 1ms), as well as switching to an adjacent cylinder

(about 1ms).

What is track skewing?

track skewing

The idea of track skewing is to position HD sectors such that rotational delay fully overlaps with switch times. This means, HD sectors on tracks on the same cylinder (as well as on adjacent cylinders) are slightly shifted in their position on the platter. The shift is done such that the head switch time (or the cylinder switch time) corresponds to the rotational delay. Like that no additional delay occurs once the disk head is in place.

How did hard disks evolve over the last decades?

hard disk

In general random access times evolve slowly. For end-user disks they may even get worse. In contrast, sequential access times increase considerably every year.

What do we learn from that?

Accessing data sequentially gets cheaper every year. In contrast, accessing data randomly does not improve much.

Bonus Question: What does this imply for index structures?

index

This has major impact on indexing decisions (discussed later on in this course). As scans become cheaper, for certain types of queries indexing does not pay off anymore. We will get back to this in Chapter 3.

Q&As

1. What type of accesses are supported by hard disks?
   (a) Temporal Access
   (b) Random Access
   (c) Sequential Access
   (d) Microsoft Access
2. What are the major components of random access time?
   (a) t_r: time needed to rotate to the correct angle (half a rotation on average)
   (b) t_s: time to move the arm to the right track
   (c) t_{tr}: time to transfer the data
   (d) t_{op}: time the operating system needs to send the read command to the disk controller
3. How do you compute the costs to randomly read all blocks of a whole file from disk? Assume the file consists of x blocks. Given the rotational delay t_r, the seek time t_s and the time needed to transfer a single block t_{tr}.
   (a) $t_{ran} = x \cdot (t_r + t_s + t_{tr})$
   (b) $t_{ran} = t_r + t_s + x \cdot t_t r$
4. When sequential reading and switching from one head or track to another you always have to wait on average half a rotation to start reading.
   (a) False

(b) True

5. The random access times have improved tremendously over the last decades.

   (a) False

   (b) True

6. In 1970, what percentage of a file could be read randomly in the time needed to read the file fully, when reading sequentially?

   (a) 25 percent

   (b) 50 percent

   (c) 1 percent

7. With modern hard drives, what percentage of a file can be read randomly in the same time needed to read the file fully, when reading sequentially?

   (a) 25 percent

   (b) 50 percent

   (c) 1 percent

Exercise

Suppose you have 2^{20} blocks of size 8 KB each sequentially laid out on the device:

S-ATA disk:
Average Seek Time = 2 ms
Rotational Speed = 15,000 RPM
Sustained Transfer Rate = 150 MB/s
Maximum Transfer Rate = 600 MB/s

The *maximum transfer rate* describes the bandwidth that can be observed on the outer zone if no head switch or movement needs to be done. For simplicity, you can assume, that the whole file fits in the outer zone. In contrast, the *sustained transfer rate* is achieved when large consecutive portions of the disk are read. This is a simplification and includes all the needed head switch and arm movement times involved in a sequential scan.

Give the percentage of blocks that should be accessed in a full table scan to outperform random I/O.

1.2.4 Hard Disk Controller Caching

Material

Video:	Original Slides:	Inverted Slides:
		

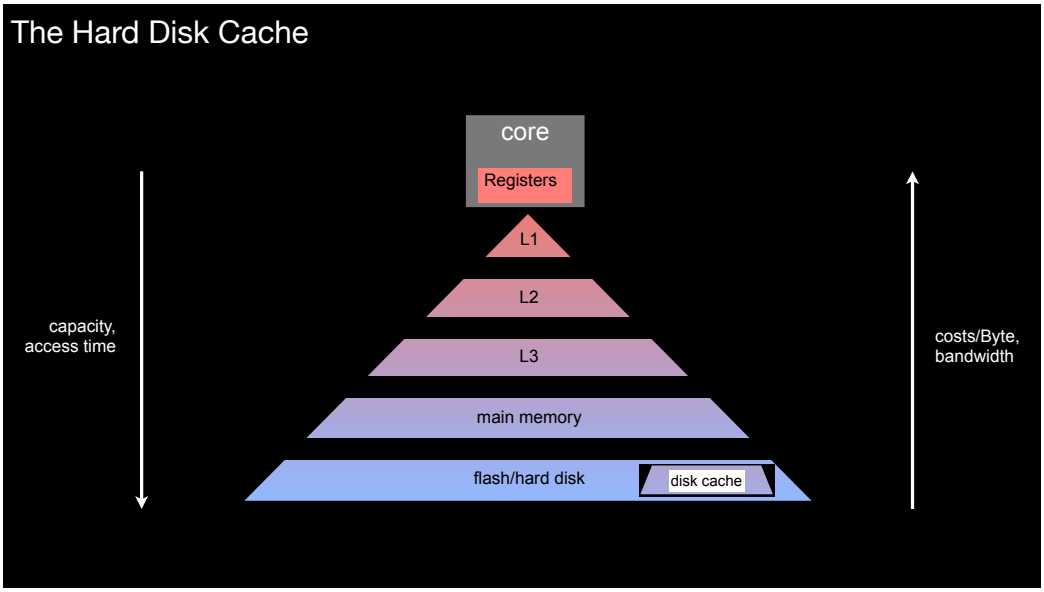


Figure 1.9: The relative position of the hard disk cache within the storage hierarchy

Learning Goals and Content Summary

What exactly is the hard disk cache?

hard disk cache

The hard disk cache is a small amount of volatile memory (typically about 128MB) managed by the hard disk controller.

How is the hard disk cache related to the storage hierarchy?

storage hierarchy

It is like adding a mini storage hierarchy inside the level 'flash/hard disk'.

What do we gain by using the hard disk cache? What do we lose?

We see the same effects as for every situation when a faster layer in a storage hierarchy caches data from a layer above: read requests already available in that cache (in this case the hard disk cache) do not have to be fetched from underneath (in this case the actual platters). For write requests: once we write data to the cache (in this case the hard disk cache) from a storage layer above (say main-memory) we do not necessarily write it through immediately to the layers underneath (in this case the actual platters). The latter optimization obviously also has some risks: if data was only written to the volatile cache but not to the persistent disk, we may lose that data.

What does this imply when storing data on disk?

We must make sure to understand when data written to disk is actually forced to the platters. This can typically be enforced by calling a separate `flush`-command.

What is the elevator optimization?

elevator optimization

If the hard disk controller receives requests for multiple tracks, it may reorder the execution order of those requests. So rather than executing requests in the same order as they arrive, the controller may improve the overall throughput of the device by handling requests which can be visited *on the way*. This is similar to an elevator which will stop on every floor where the button got pressed.

Q&As

1. What is typically cached in the disk cache?
   (a) Just the requested block.
   (b) The whole track that was read to serve the read request.
2. For what types of requests can the disk cache be used?
   (a) Only read requests.
   (b) Both read and write requests.
3. Why is it important to flush after writing to disks?
   (a) The data might still be in the disk cache and not yet written to disk.
   (b) The data is only kept in main memory till a call to flush occurs.
4. Accesses to the hard disk is always served in the order of the request arrivals.
   (a) False
   (b) True
5. Accesses to the hard disk can be reordered by the disk controller to increase the throughput of the hard disk.
   (a) False
   (b) True

Exercise

Assume we extend the HD-interface to allow for the retrieval not only of a block having a particular logical block-ID, but a conditional retrieval where the block is only returned if it contains a particular byte-sequence (of any size smaller than the block size).

For instance, rather than sending something like:

$$getBlock(42) \rightarrow block\_contents,$$

which returns the contents of block 42, we want to have something like this:

$$getBlock(42, 0x4304F04F04C) \rightarrow block\_contents.$$

Only if block 42 contains that byte-sequence, the contents of that block are returned (just like before), otherwise an error code NOT_FOUND is returned. Notice that this may be extended to something like:

$$getBlocks(0, 420000, 0x4304F04F04C) \rightarrow \{block\_contents\}.$$

"get me all blocks in range 0 to 420000 including having that byte-sequence". Notice that this call returns a **set of blocks** containing the qualifying blocks only.

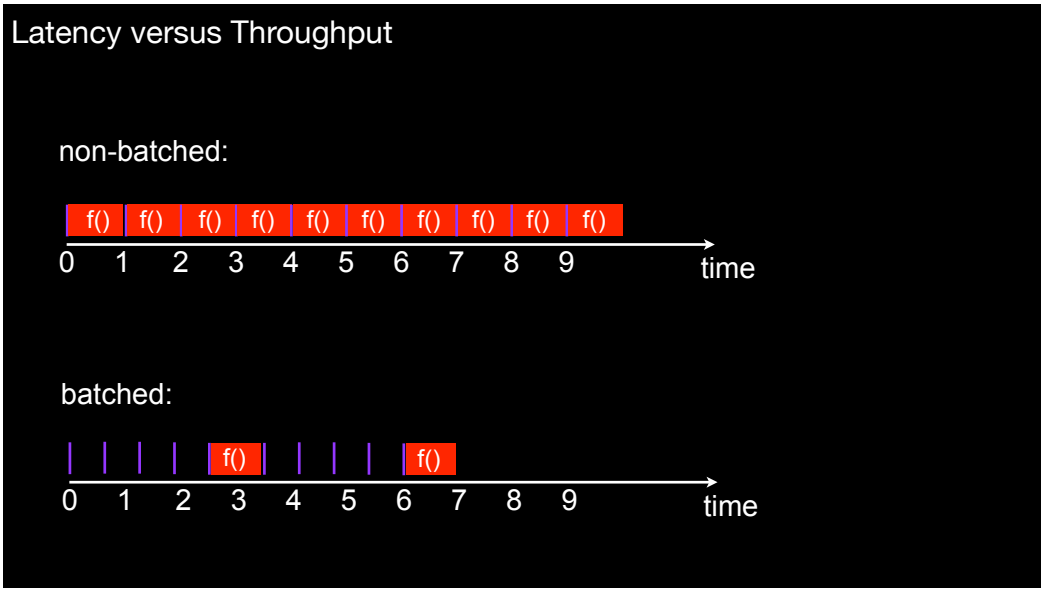


Figure 1.10: Balancing latency and throughput when batching data

(a) Discuss why such an idea does **not** make sense. Find arguments why this **increases** costs in terms of the overall system costs and makes the implementation of filter functionality on top of this device more difficult and **less efficient**.

(b) Discuss why such an idea **does** make sense. Find arguments why this **saves** costs in terms of the overall system costs and makes the implementation of filter functionality on top of this device easier and **more efficient**.

1.2.5 The Batch Pattern

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

batch pattern

What is the central observation here of the batch pattern?

If you are in a situation where the same function f(item) is applied to a set of items individually, it often makes sense, to collect k items in a batch and then apply a modified function f(batch) on each batch of items.

What are the advantages?

The costs for processing k items in a batch may be cheaper than processing k times f(item). The throughput is likely to increase (which is typically good).

What are the disadvantages?

The latency for an individual request is likely to increase (which is typically bad).

What does this mean for practical algorithms?

We have to find the speed spot: collect as many items as to keep the latency low enough. What this means exactly is typically defined by the application in terms of service-level agreements, aka SLAs.

What are possible applications?

Possible applications are queries in a database system ;-), requests to HD sectors, and requests to data items.

Q&As

1. Why should you use the batch pattern?
   (a) To increase throughput.
   (b) To decrease latency.
2. In the batch pattern several function calls on single items are combined to a single function call on multiple items.
   (a) False
   (b) True
3. Please mark all the following examples that use the Batch Pattern.
   (a) Single Instruction Multiple Data (SIMD)
   (b) Elevator Optimization
   (c) Batch files in Windows (.bat)
4. Algorithm A processes 10 items in 5 seconds and B processes 25 items in 10 seconds. What can be said about the performance of the algorithms?
   (a) A has a higher throughput than B
   (b) B has a higher throughput than A
   (c) both have the same throughput

1.2.6 Hard Disk Failures and RAID 0; 1; 4; 5; 6

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Literature:	
[LÖ09],	RAID 
R RAID: High-Performance, Reliable Secondary Storage	
[PH12], Section 6.9	
[RG03], Section 9.2	

Further Reading:
[SG07]
[JFJT11]

Learning Goals and Content Summary

Why should I worry about hard disk failures? **hard disk failures**

Hard case failures are common and may have several reasons. As a hard disk failure may result in loosing data you should design your system in a way that it can survive one or multiple hard disk failures.

What is the core idea of RAID? **RAID**

The core idea of RAID is to combine multiple hard disks in a Redundant Array of Inexpensive Disks. Like that RAID is able to survive single disk failures (depending on the RAID-level used). RAID may be implemented in software and in hardware.

What is the impact on performance of RAID 0?

RAID 0 simply stripes data across the disks without introducing redundancy. Therefore, we do not gain anything w.r.t. reliability. However, we gain in terms of I/O performance: read and write requests may be executed in parallel by the different drives.

Is my data safer by using RAID 0?

Not at all (see above).

What is RAID 1?

RAID 1 replicates blocks across all drives. If you have n disk drives in RAID 1, you have n copies of the same data. Hence, even if you lose any $n-1$ drives, you still have one drive left to read the data from. In terms of read I/O you may gain for requests that may

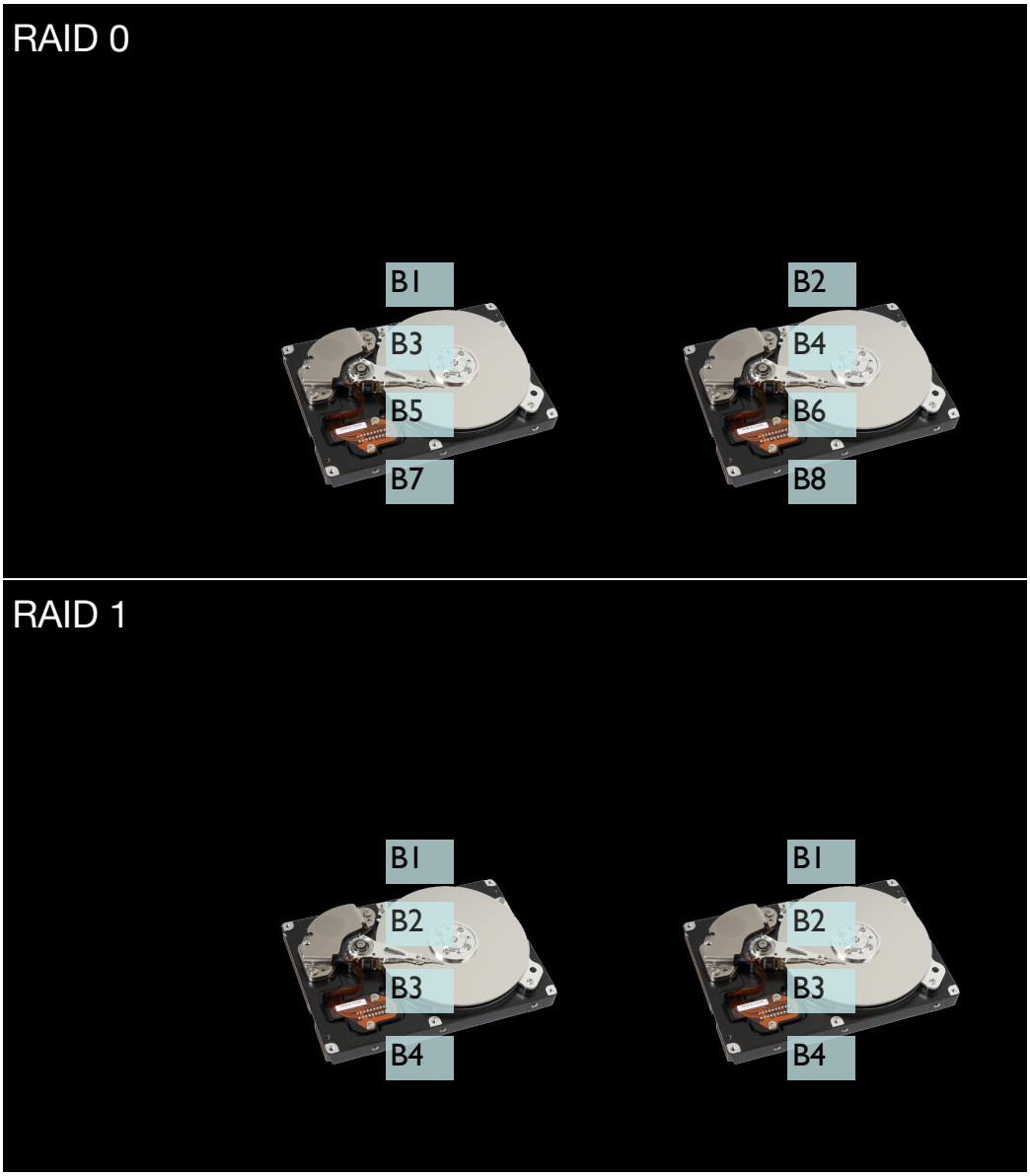


Figure 1.11: RAID 0 vs RAID 1

be split up such that one portion of the data is read from one drive and another portion from another drive. Those read requests may be executed in parallel. In terms of write I/O notice that every block written to a RAID 1 system has to be written on each of the n drives. Yet these requests may be executed in parallel.

What is the difference of RAID 4 and RAID 5?

RAID 4 is like RAID 0 with $n-1$ disks plus one additional disk for parity. The parity on that last disk is computed by XORing the striped blocks from the first $n-1$ disks. Now, if any disks fails, the contents of that disk may be reconstructed by XORing the remaining disks. RAID 4 may survive the failure of one disk (no matter which one).

A problem with RAID 4 is that the parity disk may become a bottleneck, i.e. any write operation also effects the parity. Hence whatever you write you also have to touch the

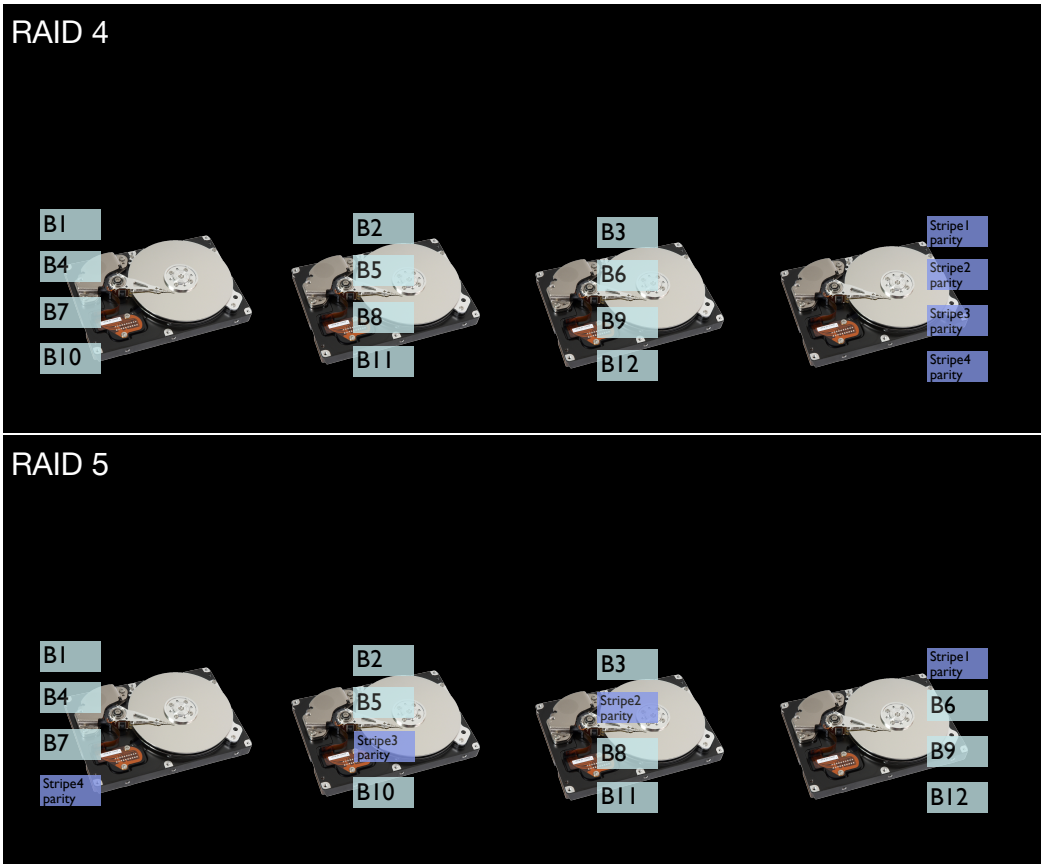


Figure 1.12: RAID 4 vs RAID 5

parity disk. Therefore a better solution is RAID 5 which distributes the parity blocks in a round robin fashion.

How many disks do I need for a RAID 4 or 5 system?

At least three disks are required in both cases.

Which sequential read performance can I expect in a system with n disks?

For RAID 4 and 5 you can expect a speed-up of factor $n - 1$.

And which write performance?

For single block writes there is no speed-up (in parallel you have to write the parity block as well!)

For sequential write operations you may have similar gains as for reads (if the RAID controller is able to group write operations affecting the same parity into a single write operation to that parity).

How many disk failures may a RAID 4 or 5 system survive?

They may both survive failure of a single disk (any of them).

What is the difference between RAID 6 and RAID 5?

RAID 5 uses a single parity whereas RAID 6 uses double parity. RAID 6 requires at least 4 disks, but may survive two disk failures.

Q&As

1. Why should you use multiple disks instead of a singe disk?
   (a) To decrease the likelihood of a failure of a single hard disk device.
   (b) To increase the likelihood of a failure of a single hard disk device.
   (c) To increase the mean time to failure of the I/O subsystem as a whole.
   (d) To decrease the mean time to failure of the I/O subsystem as a whole.
2. Please mark all properties of RAID 0.
   (a) The system survives a single disk failure.
   (b) The system can read from all disks in parallel (if the blocks you are interested in are uniformly distributed over all disks).
   (c) The system can split the write effort to all disks in parallel (if the blocks you are interested in are uniformly distributed over all disks).
3. Please mark all properties of a RAID 1 system with n disks.
   (a) The system survives n-1 disk failures.
   (b) The system can read from all disks in parallel (if enough continuous blocks are requested).
   (c) The system can split the write effort to all disks in parallel (if enough continuous blocks are written).
4. How many disk failures can a RAID 4 or RAID 5 system with n disks survive?
   (a) 1
   (b) $n - 1$
   (c) $n - 2$
   (d) $n/4$
5. RAID 5 improves over RAID 4 by distributing the parity information. Therefore the parity disk is no longer the write bottleneck
   (a) False
   (b) True
6. How many disks are needed for the minimal RAID 6 system?
   (a) 4
   (b) 3
   (c) 2

1.2.7 Nested RAID Levels 1+0; 10; 0+1; 01

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

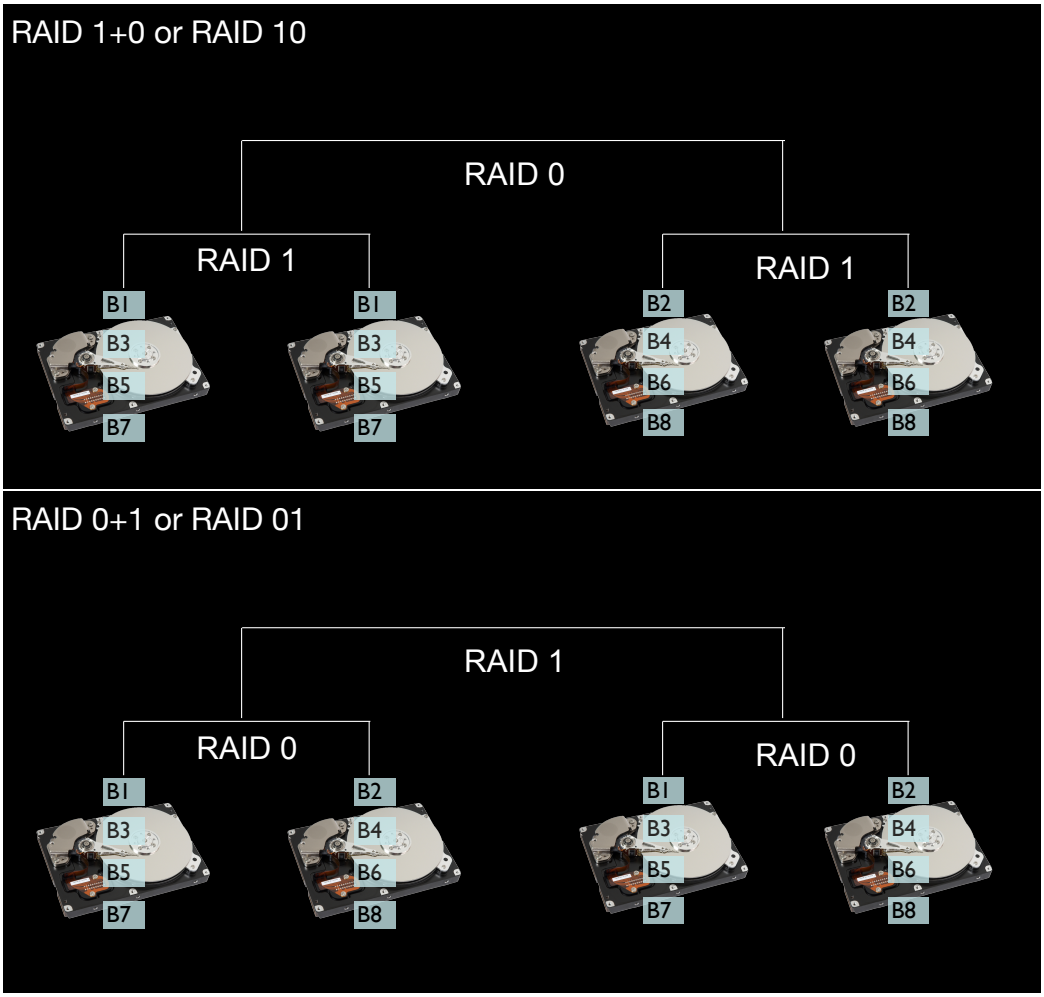


Figure 1.13: RAID 10 (=1+0) vs RAID 01 (=0+1), Note: nested RAID-levels are read from bottom to top

What is RAID 10? And why would we call RAID 10 a nested RAID?

nested RAID

RAID 10 (aka RAID 1+0) combines RAID-levels 1 and 0. It hierarchically combines (applies) both RAID-levels. On the leaf-level it combines multiple disks into a RAID 1 system (the leaf). Multiple leafs are then combined into a RAID 0 system. See Figure 1.13.

And what is RAID 01 then?

RAID 01 (aka RAID 0+1) combines RAID-levels 0 and 1. It hierarchically combines (applies) both RAID-levels. On the leaf-level it combines multiple disks into a RAID 0

system (the leaf). Multiple leafs are then combined into a RAID 1 system. See Figure 1.13.

Is it possible, in principal, to combine any kinds of RAID-levels?

Yes.

What are the properties of different nested RAID-levels?

This depends on the concrete nesting. In general, nested RAID-levels inherit properties from the non-nested RAID-levels. However, in which level we apply the non-nested RAID-levels makes a differences. For instance, let's compare RAID 10 and RAID 01 both four four disks. See Figure 1.1. They have the same properties w.r.t. performance and failover. This changes if we use more disks per group, see Figure 1.2.

	RAID 10 **2×2** ((.,.),(.,.))	**RAID 01** **2×2** ((.,.),(.,.))
max seq. read speed	×4	×4
max seq. write speed	×2	×2
max disk failures	2	2
redundancy	2	2

Table 1.1: Comparison of RAID 10 and RAID 01 using four disks each.

	RAID 10 **2L×3R** ((.,.),(.,.),(.,.))	**RAID 01** **2L×3R** ((.,.),(.,.),(.,.))	**RAID 10** **3L×2R** ((.,.,.),(.,.,.))	**RAID 01** **3L×2R** ((.,.,.),(.,.,.))
max seq. read speed	×6	×6	×6	×6
max seq. write speed	×3	×2	×2	×3
max disk failures	3	4	4	3
redundancy	2	3	3	2

Table 1.2: Comparison of RAID 10 and RAID 01 using six disks each. L means leaf-level, R means root node.

Q&As

1. How does a minimal RAID 10 setup look like?

   (a) You need four disks. Two disks form a mirror (RAID 1). On top of the two RAID 1 systems data is striped (RAID 0).

   (b) You need four disks. Two disks use striping (RAID 0). On top of the two RAID 0 systems data is mirrored (RAID 1).

2. How many disk failures can a minimal nested RAID system survive?

   (a) At least one.

   (b) Two depending on where the second failure occurs.

   (c) Up to three.

   (d) At least two.

3. How many disks are needed at least to create a RAID 50 system?

   (a) 6

   (b) 4

   (c) 3

4. In a RAID 50 configuration with 6 disks, how much data can be stored on the system?

   (a) The capacity of the smallest disk times four.

   (b) The capacity of the smallest disk times five.

   (c) The capacity of the smallest four disks (even if the disks have different capacities).

Exercise

Discuss each of the RAID configurations below in terms of (a) read performance, (b) write performance, and (c) reliability (minimal and maximal number of disks that may fail).
Assume twelve disks are used for both configurations and you are reading or writing sequentially from or to large files.

1. RAID 5 + 5 using three RAID 5 subsystems consisting of four disks each

2. RAID 6

1.2.8 The Data Redundancy Pattern

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Literature:	
[LÖ09], Replication for High Availability	
[LÖ09], Replication	

Further Reading:	
[LÖ09], Partial Replication	

Learning Goals and Content Summary

redundancy

Why is data redundancy good?

If we keep multiple physical copies of the data, we may afford loosing all but one of them. Then, we are still able to recover the data. Similarly, if we keep error correction codes/checksums that allow us to reconstruct faulty data, we may be able to recover some of the data.

Why is data redundancy bad?

Data redundancy creates storage overhead. How much depends on the type of redundancy, i.e. the RAID-level used or for RAID 1 the number of replicas used.

Can you list three examples where data redundancy is used for good?

hard disks (RAID), datacenter redundancy, backups

Q&As

1. What are the benefits of keeping redundant copies of your data on different devices?
   (a) It decreases the likelihood of losing all devices with all copies of the data.
   (b) You can write in parallel and get higher throughput.
   (c) When reading, it is just fine to read from one of the copies.
2. What are the drawbacks of keeping redundant copies of your data on different devices?
   (a) additional write effort

(b) additional resource consumption

(c) additional parity computation effort

(d) additional read effort to retrieve the data

3. Which of the following are examples of the Data Redundancy Pattern?

(a) RAID

(b) Hot Standby Server

(c) Database Partitioning (Sharding)

4. On what levels can you apply the Data Redundancy Pattern?

(a) to combine multiple devices

(b) to combine multiple systems

(c) to combine multiple data centers

5. What kind of hazzard can the Data Redundancy Pattern on disk level solve?

(a) Fire or Flood.

(b) Defect in the disc.

(c) Programming errors.

1.2.9 Flash Memory and Solid State Drives (SSDs)

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

What are the major properties of flash memory? **flash**

Flash memory is non-volatile. In contrast to hard disks it is also robust as it does not have any moving parts. In addition, many devices allow you to access data in parallel, i.e. in contrast to a hard disk which can only read one sector at a time, some types of flash memory can serve multiple requests concurrently.

What is an SSD? **SSD**

A solid state disk is flash memory in the form factor of a hard disk. It is connected to the computer via a hard disk interfaces such as S-ATA or SAS. Like that it may simply replace a hard disk in a computer system.

What is the major differentiator over hard disks w.r.t. performance? **hard disk**

A major feature of flash memory is that random access is by a factor of 100 faster — or even more: this depends on the specific device.

In an SSD, what are blocks and superblocks? **block** **superblock**

Figure 1.14: Flash memory comes in different form factors, when it comes in the shape of a hard disk we call it solid state disk (SSD).

Superblocks contain a set of blocks.

What does this mean for write operations?

An SSD cannot simply overwrite a block. Therefore, if you want to overwrite the same physical block, you first have to erase its superblock. However, in practice, with every write to a logical block you may map that logical block to a different (already erased) physical block; this is similar to what you exploit in copy-on-write and like that you do not have to wait for the erase.

write amplification

What is write amplification?

Write amplification is a measure for the write overhead triggered by superblock erasure, garbage collection, and internal RAID.

storage layer

How would this affect the storage layer of a database system?

As hard disks pages can simply be overwritten, a hard disk does not have to tell the hard disk if it considers a particular page to be unused. This is a problem for SSDs as it cannot immediately erase that page. Only when the next write to that page occurs, the SSD will COW that page and eventually erase the old page. Therefore, in terms of performance it helps, if the storage layer of the DBMS informs the SSD about pages that are unused or will soon be overwritten. This can be done by calling trim() for particular blocks of the SSD.

RAID

How does it relate to RAID?

Internally, many SSDs use a RAID-like configuration of their memory banks to increase both performance and reliability, e.g. RAID 5. This is another example of Fractal Design, see Section 2.3.5.

volatile memory

How does flash memory relate to volatile memory?

SSDs typically contain a volatile cache just like hard disks. The same problems as with hard disk caches occur, in particular lost writes.

sequential bandwidth

Which sequential bandwidth can we expect these days?

This depends a lot on the quality of the flash memory and the interface used to connect it to the computer. For an SSD a sequential bandwidth of 500 MB/sec is realistic.

random access time

And which random access time?

About 0.05 ms

Q&As

1. Please mark all properties of SSDs.
   (a) SSDs are non-volatile
   (b) SSDs need some power source to keep the data persisted
   (c) SSDs are more robust than hard drives
   (d) SSDs can be accessed faster than hard drives, especially when accessed randomly.
   (e) SSDs provide parallel accesses
2. A superblock consists of several blocks of typically 8KB
   (a) False
   (b) True
3. How are blocks written on an SSD?
   (a) One can only write into empty, freshly erased blocks.
   (b) If no block is empty, the system simply erases a block and writes into that block.
   (c) Erase can only happen at the superblock level. If no block is empty, the system has to erase a whole superblock and write into one of the erased blocks.
4. Why does an SSD sometimes physically write more data than logically required by the system?

(a) super block erasure

(b) garbage collection

(c) blocks are always stored redundantly

5. To improve the performance of database systems that use SSDs the system should send trim()-commands to the SSD for a block, as soon as the block has been written.

(a) False

(b) True

Exercise

Assume a block storage interface allowing you to store logical blocks numbered from 0 to N. Each block can be identified by its logical ID termed $LBID \in 0, \ldots, N$. Internally, the device maps these blocks to k internal physical devices. This means, the device implements a mapping

LBID $\mapsto$ {(DID, IBID, isPARITY)}.

This means logical block IDs are mapped to a **set** of internal locations. Each internal location is a triple consisting of the ID of the internal device, termed DID $\in \{0, \ldots, k-1\}$; the block ID on that device, termed IBID; and a flag isPARITY indicating whether this internal location reflects parity information about this logical block.

For instance, for a RAID 0 using k disks, assign_RAID_0(LBID, k) is implemented as:

```
assign_RAID_0(LBID, k){
      DID = LBID MOD k;
      IBID = LBID DIV k;
      isPARITY = false;
      return {(DID, IBID, isPARITY)};
}
```

Recall, that if the set contains more than one pair-wise different entry where isPARITY==false, that block is stored twice.

So, when calling `assign_RAID_0(LBID, k)` with an ascending sequence of LBIDs, and assuming three disks ($k = 3$), we obtain:

(0,3) $\mapsto$ {(0, 0, false)},

(1,3) $\mapsto$ {(1, 0, false)},

(2,3) $\mapsto$ {(2, 0, false)},

(3,3) $\mapsto$ {(0, 1, false)},

(4,3) $\mapsto$ {(1, 1, false)},

(5,3) $\mapsto$ {(2, 1, false)}, ...

(a) Implement assign functions for RAID 1, 4, and 5.

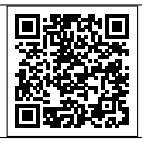
(b) Implement assign functions for RAID 10, 01, and 51 (first number is the type of the nested subsystem, i.e. for RAID 10, RAID 1 is used on the leaf-level and RAID 0 on the root)

(c) What is the relationship of assign() to LBA and disk sparing in a single hard disk?

(d) What is the relationship of assign() to wear-leveling and trim() on an SSD?

1.2.10 Example Hard Disks, SSDs and PCI-connected Flash Memory

Material

Video:	Original Slides:	Inverted Slides:
		

Learning Goals and Content Summary

What are typical performance characteristics of hard disks and SSDs?

hard disk

SSD

They have similar throughput, random access times, however are roughly by a factor 100 better for SSDs.

Would you buy an SAS (Serial attached SCSI) disk for your PC?

SAS

Probably not, as it is relatively expensive and loud.

What is a PCI flash drive?

PCI flash drive

This is flash memory connected to the computer by PCI.

Why would using a PCI flash drive be a good idea?

Like that much higher bandwidth becomes possible.

Why do I get considerably more random read operations per second than one divided by the random access time?

random access time

Many flash devices can handle multiple requests in parallel.

Q&As

1. How can SSDs fix your performance problems?
   (a) They provide much faster random reads and writes in comparison to HDDs.
   (b) They have higher storage capacities.
   (c) They allow for better cache-locality.
2. Why is the number of I/O-operations per second higher than 1 divided by the random access time for SSDs?
   (a) Several I/O operations can be served in parallel by SSDs.
   (b) This result is obtained when connecting several SSDs to the system.
   (c) The SSDs use more data buses simultaneosly.

1.3 Fundamentals of Reading and Writing in a Storage Hierarchy

1.3.1 Pulling Up and Pushing Down Data, Database Buffer, Blocks, Spatial vs Temporal Locality

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Literature:	
[RG03], Section 9.4.1 and 9.4.2	
[LÖ09], Buffer Manager	
[LÖ09], Buffer Pool	
[LÖ09], Memory Locality	

Further Reading:	
[LÖ09], Buffer Management	

Learning Goals and Content Summary

pulling up data

What does pulling up data mean?

This means that data is transferred from any layer of the storage hierarchy to a layer closer to the CPU(s).

pushing down data

What does pushing down data mean?

This means that data is transferred from any layer of the storage hierarchy to a layer further away from the CPU(s). This is only necessary if that data was modified or newly created.

database buffer

Did you see the database buffer anywhere?

The database buffer sits in-between main memory and hard disk. It controls which pages from hard disk are cached in main-memory and which pages are replaced and written back.

temporal locality

What is temporal locality?

Given a set of address references $A_1, \ldots, A_n$ and some i and j where $1 \leq i < j \leq n$. If $A_i = A_j$ and the distance of i and j is "small", we coin this temporal locality. In other words: the **same** memory address is referenced twice within a "short" period of time.

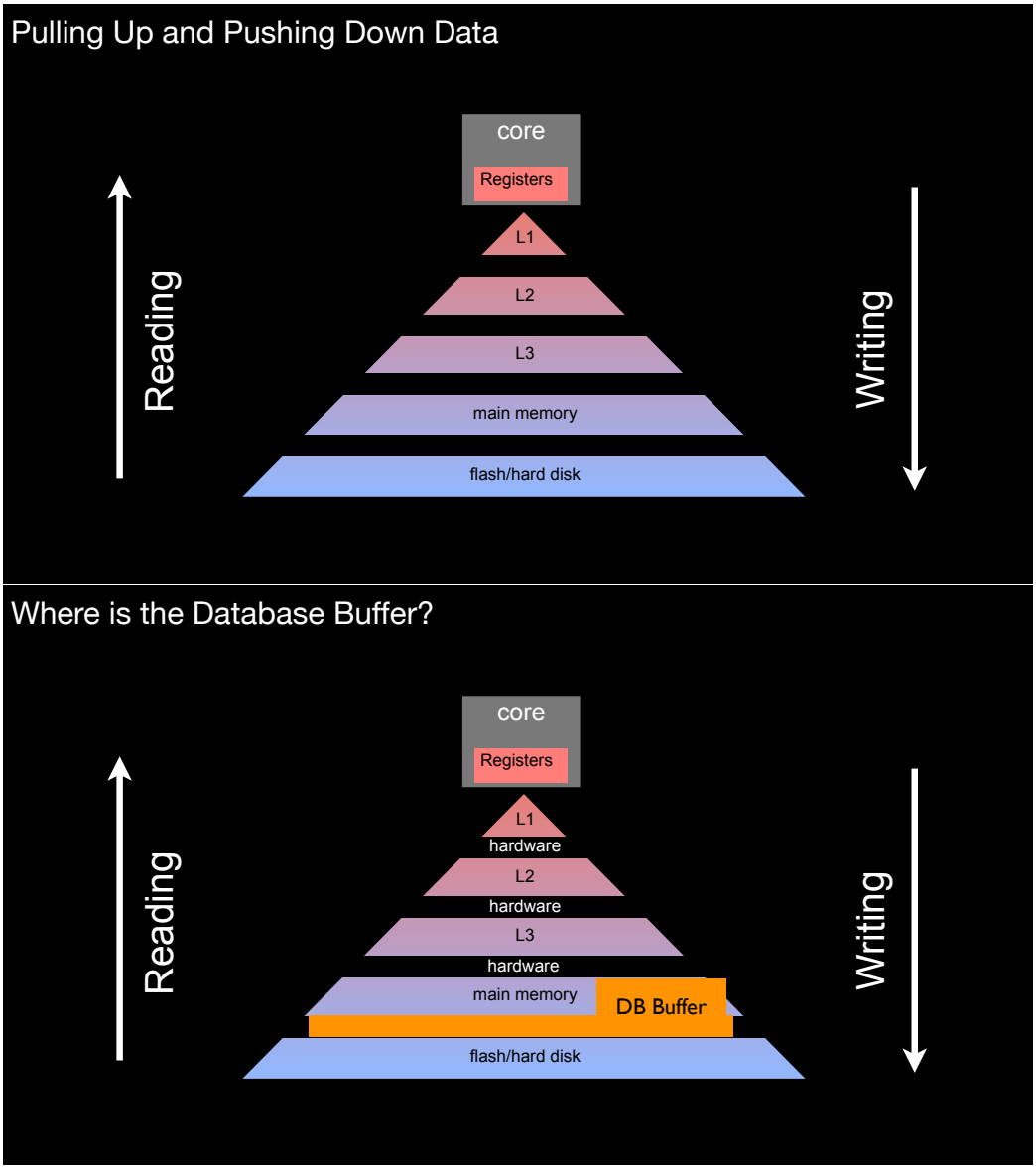


Figure 1.15: Reading data is equivalent to pulling up data in the storage hierarchy. Writing data is equivalent to pushing down data in the storage hierarchy. What is the relative position of the DB-buffer?

What "small" and "short" means, depends on the context.

And what is spatial locality then?

spatial locality

Given a set of address references $A_1, \ldots, A_n$ and some i and j where $1 \leq i < j \leq n$. If distance$(A_i, A_j) < \delta$ and the distance of i and j is "small", we coin this spatial locality. In other words: a **similar** memory address is referenced twice within a "short" period of time. What "small" and "short" means, depends on the context.

How are the two related?

Spatial locality is a generalization of temporal locality. Temporal locality only considers memory addresses that are equal and reasons about their distance in time. In contrast, spatial locality considers similar memory addresses (including equality as a special case) and reasons about their distance in time.

Q&As

1. In disk-based database systems the database buffer
   (a) is involved in pulling up data
   (b) is involved in pushing down data
   (c) is a hardware component
   (d) resides on the hard disk
   (e) is a dedicated special memory chip on DRAM
2. Which of the following are correct in the context of database buffers?
   (a) Blocks are loaded with higher speed than pages due to spatial locality.
   (b) Multiple pages form a block.
   (c) Blocks reside on hard disk.
   (d) Pages reside in main memory.
   (e) Blocks are pulled up, and pages are pushed down.
3. Temporal locality
   (a) is exploited by the database buffer
   (b) means accessing the same pages more than once in a given short amount of time
   (c) means accessing similar pages more than once in a given short amount of time
   (d) means accessing pages that where last modified nearly at the same time point
   (e) means actually pulling up the same blocks again and again from disk in a given short amount of time
   (f) a palindrome has temporal locality in a character stream
4. Spatial locality
   (a) is exploited by the database buffer
   (b) means accessing the same pages more than once in a given short amount of time
   (c) means accessing similar pages more than once in a given short amount of time means
   (d) accessing pages that where last modified nearly at the same time point
   (e) means actually pulling up the same blocks again and again from disk in a given short amount of time
   (f) a palindrome has the highest possible spatial locality in a character stream
5. Prefetching, e.g. read ahead, done by hard disk controllers exploits:
   (a) spatial locality
   (b) temporal locality
   (c) database buffers
   (d) non-monotonous distance functions

Implementation of GET

```
Get(Px):
1.  If (not PAGE_IN_BUFFER(Px)):                  // check whether already exists
2.          if (no empty slot available in buffer):   // is there space to load a page?
3.                  S = Pi = CHOOSE_PAGE();           // choose a page to kick out
4.                  if (Pi is dirty):                 // did anyone change this page?
5.                          flush Pi to external memory;  // oops, got to write it out first
6.          else:                                     // we have space left anyway...
7.                  S = getFreeSlot();                // pick a free page
8.          read(Px, S);                              // read Px into free slot
9.  fix(Px);                                          // fix page Px
10. return Px;                                        // return a reference to Px
```

Figure 1.16: The implementation of the get method in the database buffer

1.3.2 Methods of the Database Buffer, Costs, Implementation of GET

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

What are the most important methods of the database buffer? — **database buffer**

get(), fix(), unfix(), page_in_buffer(), and choose_page().

What does get(P_x) do? — **get(P_x)**

I returns a reference/pointer to Page P_x.

What may happen when you request a page that is not in the buffer? — **page**

If all slots in the buffer are full, we first have to choose a page for eviction (choose_page()). If the page to be evicted is dirty, we first have to write it back to the storage layer underneath. Only after that we may load the page actually requested.

What are the costs involved? — **costs**

In the worst case, two random I/O-operations may be triggered: one for writing back the dirty page to be evicted, the second for reading the page actually requested. If the page to be evicted is not dirty, we can obviously simply discard it. Then no write back is necessary.

Who would evict a page? — **evict**

Page eviction is triggered by the DB-buffer every time no empty slot is available anymore. Other than that there is no reason to evict a page. Still, a DB-buffer should run an

additional background thread to regularly write back dirty pages in the background. This avoids the extra costs for writing back dirty pages at page eviction time, and it also improves recovery time. see also Section 6.

Q&As

1. The get-method of the database buffer may perform the following operations:
   (a) Check if the page is already in the buffer.
   (b) Read the page from disk.
   (c) Evict a page from the buffer.
   (d) Flush dirty pages to disk.
2. The operations of the database buffer trigger the following costs:
   (a) 2 disk seeks when it has to flush a dirty page to disk before reading the new page.
   (b) Evicting pages from the buffer does not always necessitate a disk seek.
   (c) 2 disk seeks when it has to flush a clean page to disk before reading the new page.
3. Why do we have to be careful when flushing dirty pages to disk?
   (a) It is a significant computational effort to choose which page to flush to disk, because of the high costs of generating random numbers.
   (b) It involves 1 random I/O operation.
   (c) It involves 2 random I/O operations.
   (d) To avoid reading a dirty page again into the free slot created by flushing the page.
4. What is the role of the fix and unfix methods?
   (a) to prevent a page from being evicted while a process is still working on that page
   (b) to prevent writing modified content back to disk
   (c) to prevent concurrent processes to read from the fixed page

1.3.3 Pushing Down Data in the Storage Hierarchy (aka Writing), update in-place, deferred update

Material

Video:	Original Slides:	Inverted Slides:
		

Learning Goals and Content Summary

What is a direct write?

direct write

Assume a block A is modified in main memory and becomes A'. Further assume that A' is written back to the storage layer underneath, say the hard disk, overwriting (and replacing) the old version A. Then we call this a direct write. In other words, the new version of the block is written **over** the old version of that block.

What is an indirect write?

indirect write

Assume a block A is modified in main memory and becomes A'. Further assume that A' is written back to the storage layer underneath, say the hard disk, **but not** overwriting (or physically replacing) the old version A. In contrast, the old version of the block A is kept. The new version A' is written to a different place. Then we call this an indirect write. In other words, the new version of the block is **not** written **over** the old version of that block, but rather kept in a different place.

What are their pros?

In direct write, if anything goes wrong while writing back the new version A' (e.g. a hard disk problem), you may end up in an inconsistent state, e.g. the first half of the block represents the new version A', the second half the old version A. In other words, you do not have a fully consistent version of that block anymore. In direct write this may not happen, as if anything goes wrong while writing the new version A', you still have the consistent version A.

What are their cons?

Indirect write introduces a level of indirection. This typically implies fragmentation which deteriorates the sequential layout of data on the storage medium. Direct writes do not suffer from this problem. The storage layout is not affected by direct writes.

Q&As

1. Direct write means that
   (a) we apply updates directly on disk, bypassing the database buffer.
   (b) we apply updates individually and flush the dirty pages to disk immediately.
   (c) we simply overwrite the old block on hard-disk if the page is dirty.
   (d) we use special hardware components to accomplish the update operations, thus the name update in-place.
2. Direct write
   (a) alone cannot ensure a consistent state of the database.
   (b) can ensure a consistent state of the database if we keep old blocks.
   (c) overwrites log records for the given page, i.e. updates them in-place.
   (d) ensures a consistent state of the database.
3. Direct write without logging violates the following ACID properties:
   (a) durability, since a power failure could result in dirty pages not being completely written to disk.

(b) isolation, since the updates of different transactions cannot be identified anymore on the blocks that have been overwritten.

(c) consistency, since the blocks are overwritten on disk before the transaction commits.

(d) atomicity, since a power failure could result in dirty pages not being completely written to disk.

(e) atomicity, since a power failure could result in some dirty pages completely written to disk, but other dirty pages not written to the disk at all.

4. Indirect write

(a) creates a copy of the old block before applying each update to the page.

(b) keeps a copy of the old block until the transaction has committed.

(c) keeps a backup copy of all blocks as a hot stand-by in case the DBMS crashes.

5. Deferred updates means that

(a) updates are collected and applied to a page at once.

(b) updates are not applied immediately on the consistent version visible to other transactions, but only when the transaction has committed.

(c) updates are only visible to other transactions if the transaction has committed.

Exercise

Assume a DB-buffer having slots for 4 pages only (sic! to allow you to draw the solution more easily). The DB buffer implements LRU (least recently used), i.e. the page that was referenced the longest time ago among all entries will be evicted. Pages numbered from $0, \ldots, N$. The database currently runs two types of queries ("type" means: queries of the same type trigger the same page reference sequences):

Q1: references pages 0, 1, and 2

Q2: references pages 4, 5, 6, and 7

Notice that each query references the pages in exactly the order specified. This also means that the DB-buffer does not see all page references done by a query at the same time at the beginning of the query (in fact the concept of a query is not understood by the DB-buffer), i.e. the DB-buffer sees individual page requests one by one and has to make sure that the page requested is or becomes available in main memory.

Whenever a query accesses a page that is **not** available in the DB-buffer, we count this as **one cost unit**, otherwise we assume no costs for accessing that page.

(a) Let's assume we start with an empty DB-buffer. Q1 and then Q2 is executed. What is the state of the DB-buffer after these two queries? Notice that the state of the DB-buffer is the **set** of pages currently kept in the DB-buffer. What were the costs?

(b) Let's assume we start with an empty DB-buffer. Assume that Q1 is executed frequently in this system whereas Q2 is executed rarely. For instance, Q1-type queries

are executed x times in a row. Then a Q2-type query is executed exactly once. This pattern is then repeated N times. What is the state of the DB-buffer after these $(x+1) \cdot N$ queries? What were the costs?

(c) What other methods (at least two algorithmically different methods, possibly having the same effect on the pages evicted from the DB-buffer) could you think of to lower the costs computed in (b)? What is the state of the DB-buffer after these $(x+1) \cdot N$ queries? What were the costs?

1.3.4 Twin Block, Fragmentation

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

twin block

What is the core idea of twin block?

The core idea of twin block is to keep two versions of each block. Like that for each block we have an a-version and a b-version. One of these versions is considered consistent and read-only, the other version is considered possibly inconsistent and may be modified by an ongoing transaction. The roles of the a and b versions change over time. A global switch indicates which of the two versions is currently considered the consistent version.

What are its pros?

We do not need extra helper data structures, undo of changes (e.g. aborting a transaction) is easy, and there is no fragmentation introduced by the method.

What are its cons?

The storage requirements are doubled.

Q&As

1. In case of twin block
   (a) each block is stored twice physically on disk.
   (b) each block has two backup copies.
   (c) backup copies of blocks are only created for dirty pages.
   (d) backup copies are only created for blocks read into the database buffer.
2. In case of twin block without concurrent transactions
   (a) we direct all read requests to the consistent version of the block.
   (b) we direct all read requests to the new version of the block.
   (c) we are dealing with an in-place update method.
   (d) we are dealing with an indirect write method.

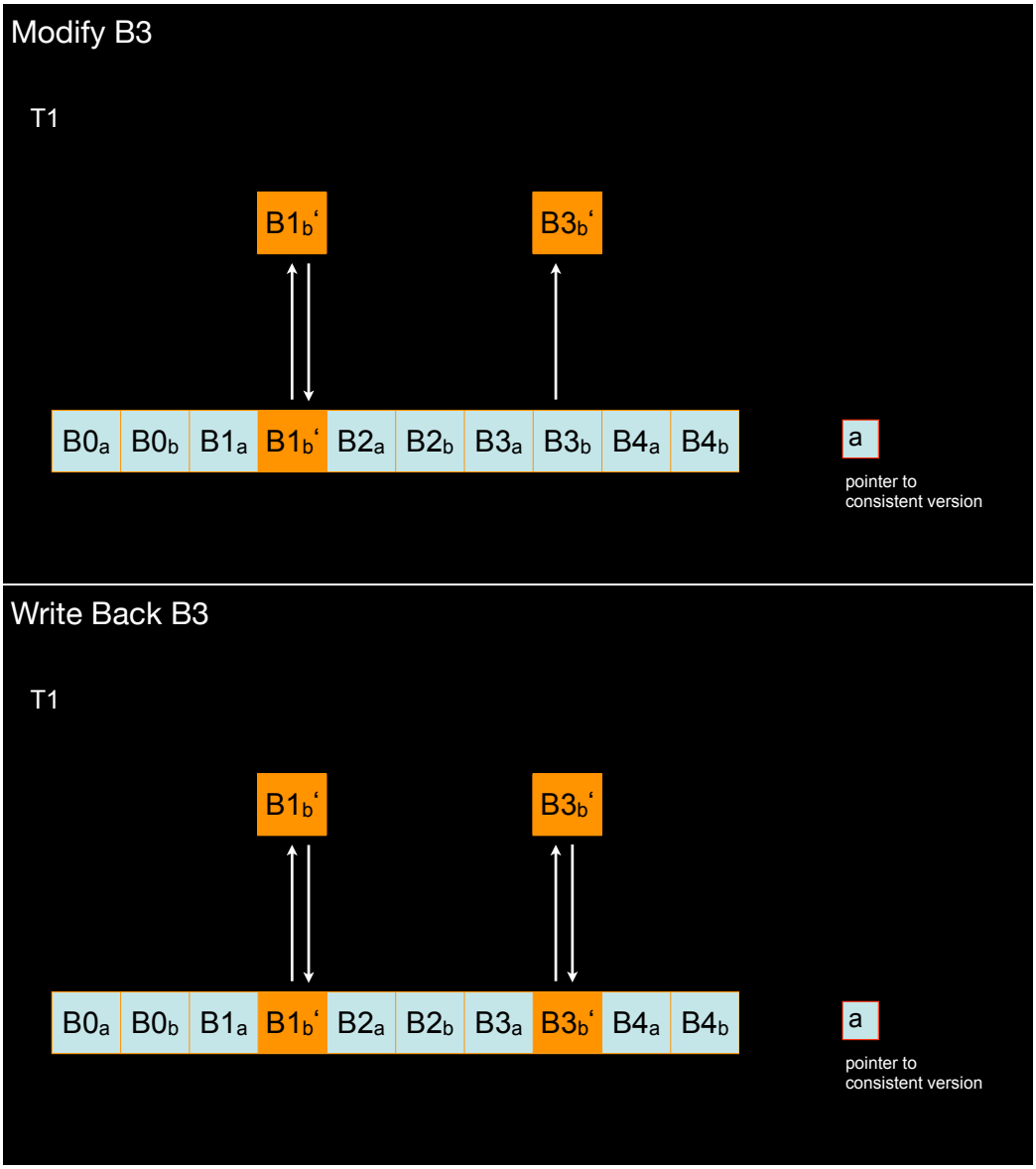


Figure 1.17: Twin block: modifying version b of block B3 to B3' and writing it back to storage.

3. The method switching between the consistent and (possibly) inconsistent versions of the blocks has to be:
   (a) in-place
   (b) atomic
   (c) asynchronous
   (d) write-through
4. When flushing out dirty pages to disk at any time before committing transactions using twin block:
   (a) the fragmentation of the disk increases due to the free space created by deleting the twin block.
   (b) we actually need to write the page twice when flushing it to disk.

(c) we are only allowed to overwrite the inconsistent version of the block.

(d) after switching versions, all blocks changed by the transaction have to be copied to the inconsistent version.

5. The benefits of twin block are:

(a) undoing changes is easy

(b) storage requirements increase only moderately

(c) needs only an A-B version toggle per page

(d) does not require complex helper data structures

1.3.5 Shadow Storage

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

shadow storage

What is the core idea of shadow storage?

We only keep two versions of those blocks currently being modified, i.e. those blocks that are part of an uncommitted transaction. Indirection from logical blocks to physical blocks is organized using two mapping tables. One of these mapping tables is considered consistent and read-only, the other version is considered possibly inconsistent and may be modified by an ongoing transaction. The roles of the two mapping tables may change over time. A global switch indicates which of the two mapping tables is currently considered the consistent version. Shadow storage is an example of an indirect write method.

What are its pros?

We do not double the storage overhead (as in twin block), but rather only double the storage requirements for the modified blocks. Undoing changes is easy.

What are its cons?

The indirection introduced by the mapping table may lead to fragmentation. The helper data structures may become big.

Where is this used outside databases?

It is used in file systems like ZFS. In addition, virtual memory is basically an implementation of shadow storage. See also Section 1.4.

Q&As

1. In shadow storage we keep

(a) two versions of each modified block

(b) two versions of each block

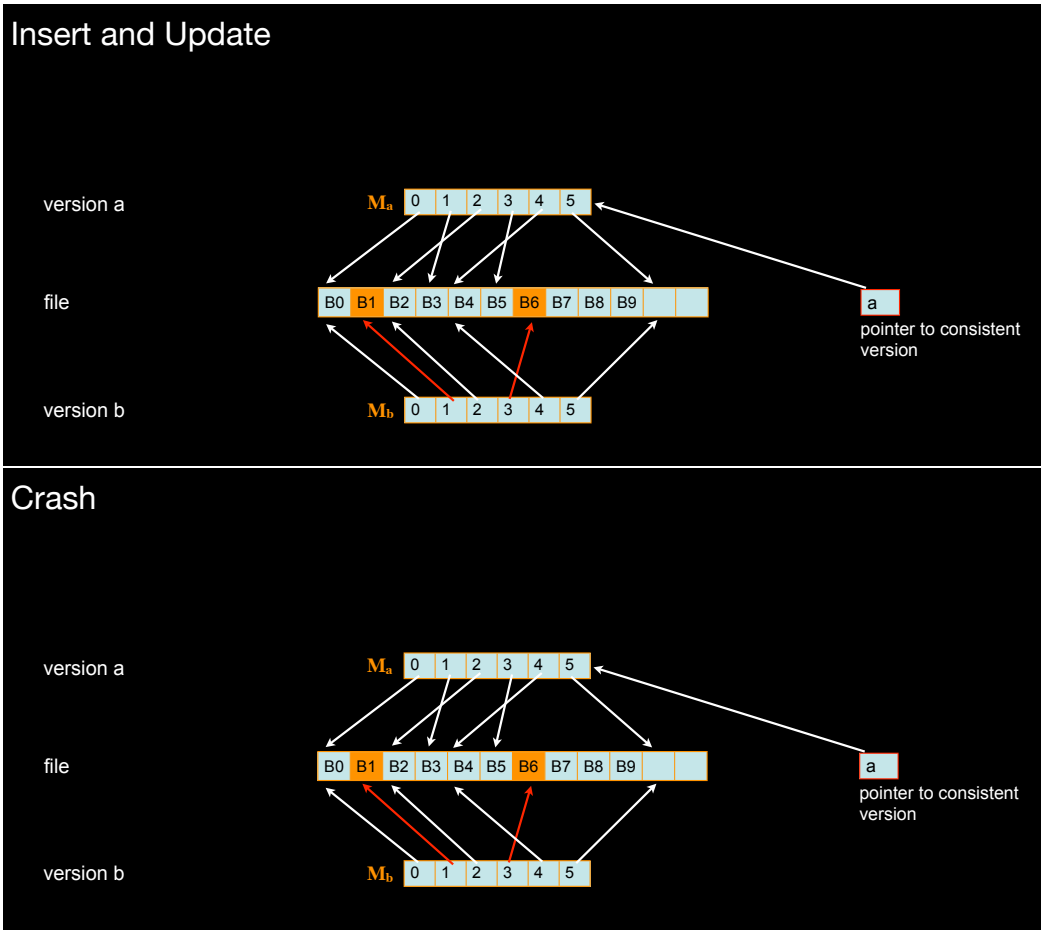


Figure 1.18: Shadow paging: inserting and updating data vs handling a crash

(c) two versions of each block in the database buffer

(d) two versions of each block that is fixed in the database buffer

2. The mapping table in shadow storage is used for:

   (a) translating array indexes to logical block addresses.

   (b) translating the dirty page numbers to physical block addresses.

   (c) translating logical block numbers to physical block addresses.

   (d) translating block numbers of inconsistent blocks to the physical address of the consistent copy of the given block.

3. In case of shadow storage

   (a) in case the transaction crashes, the new transactions will read the consistent version of the block.

   (b) in case the transaction crashes, the new transactions will read the newest version of the block.

   (c) we are dealing with an in-place update method.

   (d) we are dealing with an indirect write method.

4. To abort a running transaction and undo its updates:
   (a) we toggle the version pointer, so that it points to the consistent version of the mapping table.
   (b) we just have to use the mapping table pointed to by the version pointer, and copy its contents over the other mapping table
   (c) we need to erase the inconsistent versions of the blocks.
   (d) we need to overwrite the inconsistent versions of the blocks with their corresponding consistent version.
5. Assume, the currently consistent version is B, the inconsistent version containing some changes is A. The proper order of actions to persist the changes of a transaction that did changes to A are:
   (a) flush the dirty pages, flush M_a, toggle the global version pointer, copy the contents from M_a to M_b.
   (b) flush the dirty pages, flush M_a, toggle the global version pointer, copy the contents from M_b to M_a.
   (c) flush M_a, toggle the global version pointer, copy the contents from M_a to M_b, flush the dirty pages.
   (d) flush the dirty pages, toggle the global version pointer, copy the contents from M_a to M_b, flush M_a.
6. The benefits of shadow storage are:
   (a) undoing changes is easy
   (b) storage requirements increase only moderately
   (c) needs only a version toggle per page
   (d) does not require complex helper data structures compared to twin block

1.3.6 The Copy On Write Pattern (COW)

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

Copy On Write Pattern

COW

When is the Copy On Write Pattern (COW) applicable?

Whenever there is computer memory that may be partitioned into units, we may apply COW.

What is the core idea?

Assume you have two users who both operate on large portions of data (whether that data is in main memory or on disk does not matter). Assume that data is partitioned into units (e.g. pages or anything else). Let S1 be the set of pages from user 1, S2 respectively. Let *dup* = S1∩S2 contain the pages that are byte-equivalent across S1 and S2. Then, it does not make sense to store the pages that are contained in *dup* twice. It is more efficient to only store those pages once and make both users share those pages.

However, what happens if any of the two users, say user 1, wants to modify one of the pages in *dup*? In that case, user 2 would also see the changes. This is typically not what we want. Therefore in exactly this situation COW kicks in: if user 1 modifies a page in *dup* that page is removed from *dup* and duplicated. Now, each user, i.e. sets S1 and S2, has a private version of that page and can modify that page.

Where is it applied?

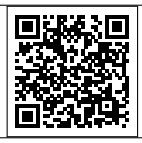
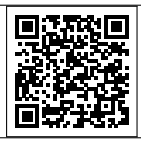
In all kinds of places. It is used as a method for indirect writes in databases. It is also used in virtual memory management: if a process spawns a child process their memory is organized by COW.

Q&As

1. Using the copy-on-write pattern the same logical block addresses
   (a) always translate to different physical addresses.
   (b) might translate to the same physical address.
   (c) might translate to different physical addresses.
   (d) always translate to the same physical address.
2. Using the copy-on-write pattern we can
   (a) share physical blocks among database transactions
   (b) share physical blocks among operating system processes
   (c) hide logical addresses
   (d) have fewer physical blocks allocated than logical blocks
3. Given multiple transactions operating on a set of logical blocks. Using the copy-on-write pattern we keep
   (a) a single physical copy of each logical block
   (b) one physical copy of each logical block per transaction
   (c) we create a new physical copy of a logical block for each transaction modifying that block
4. The benefits of using the copy-on-write pattern are:
   (a) storage requirements are considerably increased over twin blocks
   (b) isolation of transactions
   (c) does not require complex helper data structures
   (d) reduces fragmentation of data

1.3.7 The Merge on Write Pattern (MOW)

Material

Video:	Original Slides:	Inverted Slides:
		

Learning Goals and Content Summary

Merge on Write Pattern

When is Merge on Write Pattern (MOW) applicable?

MOW

In similar situations as COW.

What is the core idea?

Assume you have two users who both operate on large portions of data (whether that data is in main memory or on disk does not matter). Assume that data is partitioned into units (e.g. pages or anything else). Let S1 be the set of pages from user 1, S2 respectively. Let $dup = S1 \cap S2$ contain the pages that are byte-equivalent across S1 and S2. We already considered this use-case for COW, see Section 1.3.6.

However, what happens if any of the two users, say user 1, wants to modify one of the pages in $S1 \setminus dup$ such that after that change the contents of that page are byte-equivalent to a page in $S2 \setminus dup$? This is exactly this situation where MOW kicks in: if user 1 modifies a page in $S1 \setminus dup$, that page is moved from $S2 \setminus dup$ to dup. Now, the two users, share that page.

Where is it applied?

In data deduplication.

What is the relationship to COW?

COW

Again, MOW is the inverse operation to COW.

Q&As

1. The merge-on-write pattern is
   (a) the inverse of the copy-on-write pattern.
   (b) the anti-pattern of the copy-on-write pattern.
2. Given multiple transactions operating on a set of logical blocks. Using the merge-on-write pattern
   (a) we keep a single physical copy of each logical block
   (b) we keep a single physical copy of all logical blocks with the same contents
   (c) we create a new physical copy of a logical block for each transaction modifying that block
   (d) we might free the memory used by a physical copy of a block after updating that block
3. Using the merge-on-write pattern different logical block addresses

   (a) always translate to different physical addresses.
   (b) might translate to the same physical address.
   (c) might translate to different physical addresses.
   (d) always translate to the same physical address.

4. Using the merge-on-write pattern two different logical addresses can point to the same physical address after:
   (a) performing specific updates to a block
   (b) calling the fix method of the database buffer
   (c) copy-on-write

5. The benefits of using the merge-on-write pattern are:
   (a) storage requirements are considerably increased over twin blocks
   (b) it can reduce storage requirements
   (c) it does not require complex helper data structures
   (d) it reduces fragmentation of data

6. The merge-on-write pattern
   (a) is related to compression
   (b) cannot be used together with the copy-on-write pattern
   (c) cannot be applied to shadow storage
   (d) is applied in twin-blocks

Exercise

Assume pages of size 4 Bytes (sic! to allow you to draw the solution more easily) and a sequence of write operations belonging to transactions executed as shown below. Initially, all bits in all pages are set to 0. We assume shadow storage implementing both copy-on-write and merge-on-write. At all times only one transaction is running (no concurrency). Each transaction writes a 4 Byte integer to a page (in other words: the entire page is overwritten).

1. begin
2. P2 w(1)
3. P1 w(10)
4. P7 w(12)
5. commit
6. begin
7. P7 w(7)
8. P2 w(8)

9. P0 w(5)
10. commit
11. begin
12. P2 w(10)
13. P6 w(7)
14. abort
15. begin
16. P2 w(10)
17. P6 w(7)
18. P8 w(7)
19. commit

(a) Show the mappings stored in the page table(s) after performing each of the above operations.

(b) What is the maximum number of physical pages allocated at any given time?

(c) Assume a method Twin Block++ which works as follows: it keeps $2 * n$ blocks and two separate bit lists of n bits each. If bit i is set in a bit list, that means version A is considered the consistent version for this block, otherwise B is the consistent version for this block. We keep a global bit to signal which of those bit lists is considered to reflect the consistent state.

If a new transaction comes in, the inconsistent bit lists (initially a copy of the consistent bit list) is modified as follows: for each block i that needs to be changed, initially the globally consistent version is copied over the inconsistent version, bit i is flipped, and then changes are performed on the inconsistent version. If the transaction commits, all changed blocks are flushed to disk, the global pointer is flipped, ...

Answer (a) and (b) for this method.

(d) What are pros and cons of Twin Block++ vs. Twin Block and Shadow Storage?

1.3.8 Differential Files, Merging Differential Files

Material

Video:	Original Slides:	Inverted Slides:

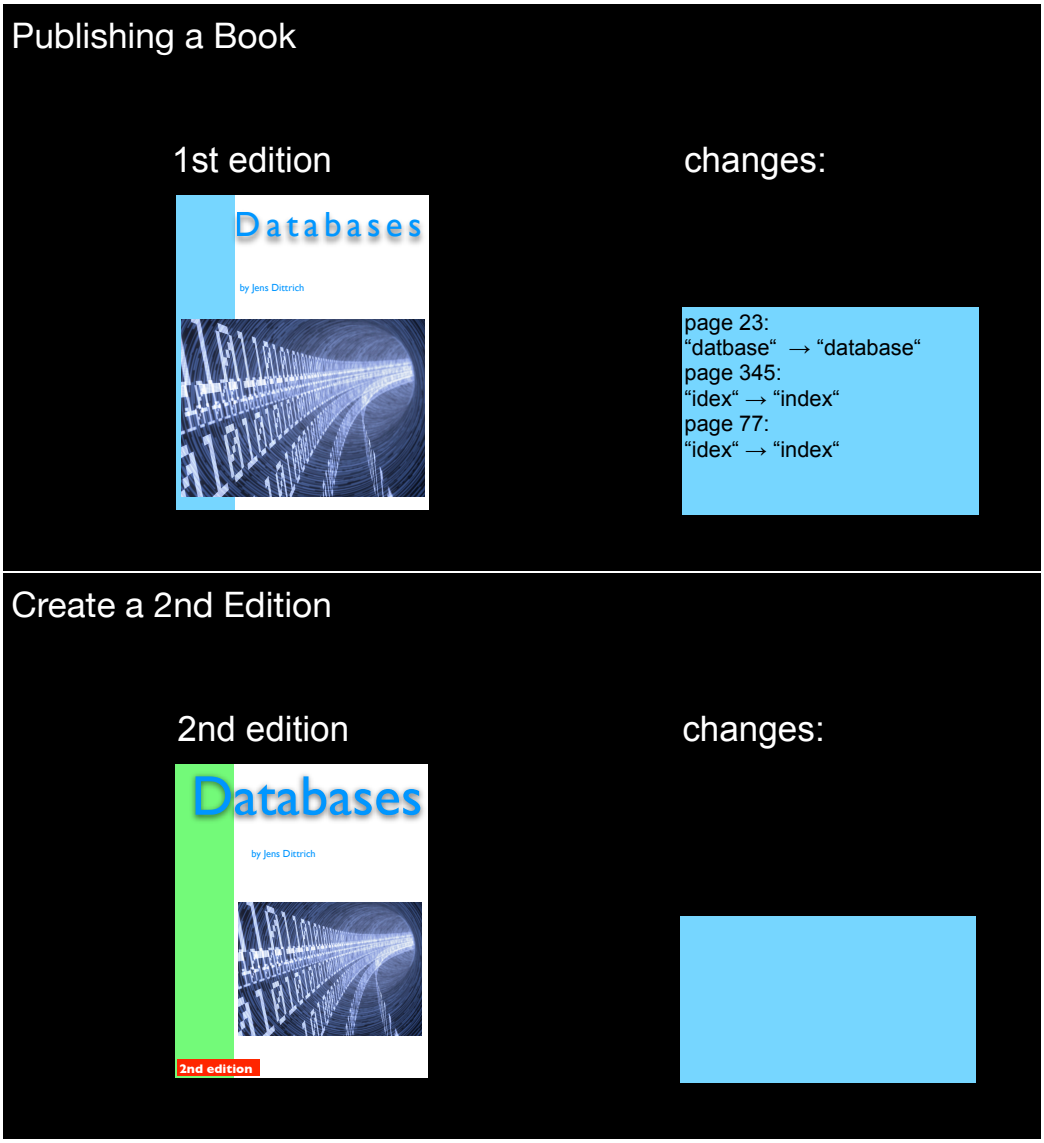


Figure 1.19: Publishing a 1st edition of a book and collecting changes vs eventually creating a second edition

Learning Goals and Content Summary

differential files

What is the main analogy from real life for differential files?

The main analogy of differential files is publishing books. Publishing houses publish a 1st edition of a book. Then they collect errors, typos, and other suggestions to improve the book in a list of changes. Eventually, they merge the 1st edition of the book and everything they collected to create a second edition. This analogy is implemented in differential files where editions are coined *files* and the list of changes is coined *diff file*.

pattern

To which other patterns does this relate?

It relates to The Batch Pattern (Section 1.2.5) and The All Levels are Equal Pattern (Section 1.1.1).

What are the pros and cons of differential files?

In the differential files method, write operations are relatively cheap. In addition, the diff

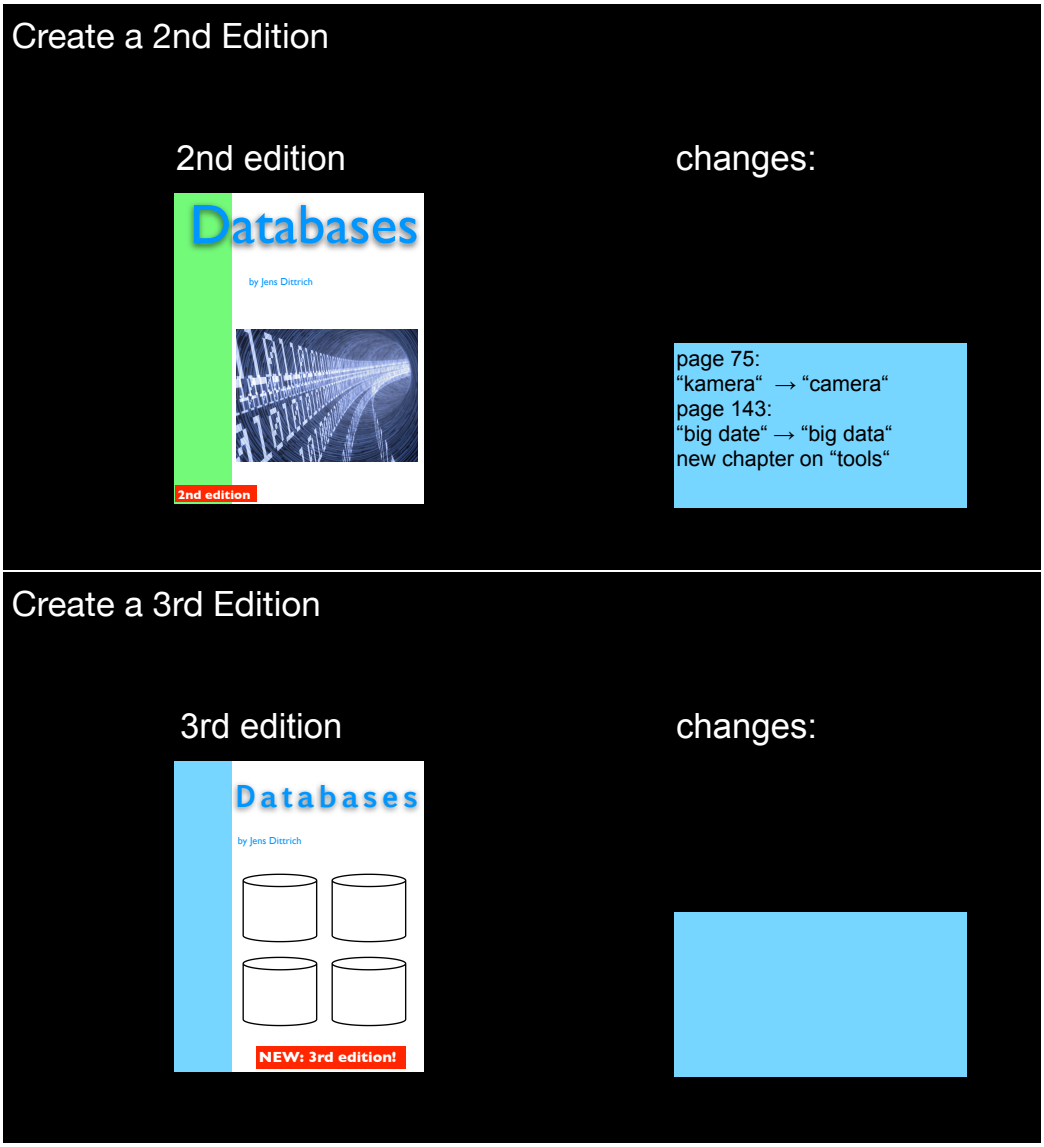


Figure 1.20: Collecting changes over the second edition vs eventually creating a 3rd edition

file corresponds to an incremental backup. The “editions” (or files) may be considered snapshots, whereas the entries in the diff file provide you with an incremental backup which allows you to move the state of the database forward from the most recent snapshot (the file). Moreover, if you perform merges regularly to create new read-only files from old read-only files and growing diff files, this is an opportunity to defragment the storage layout.

How to merge differential files with the read-only DB without halting the database?

read-only DB

This merge operation can be performed without halting incoming read and write operations. To merge an existing file v0 with diff file 0, simply switch diff file 0 to read-only and start a new writable diff file 1. All write operations are now directed to diff file 1. All read operations must consider file v0, diff file 0, and diff file 1 to obtain the most recent state of the database. Now, you merge file v0 with diff file 0 to form a new file v1. This can be done in a background thread. Once that merge operation is finished, all read

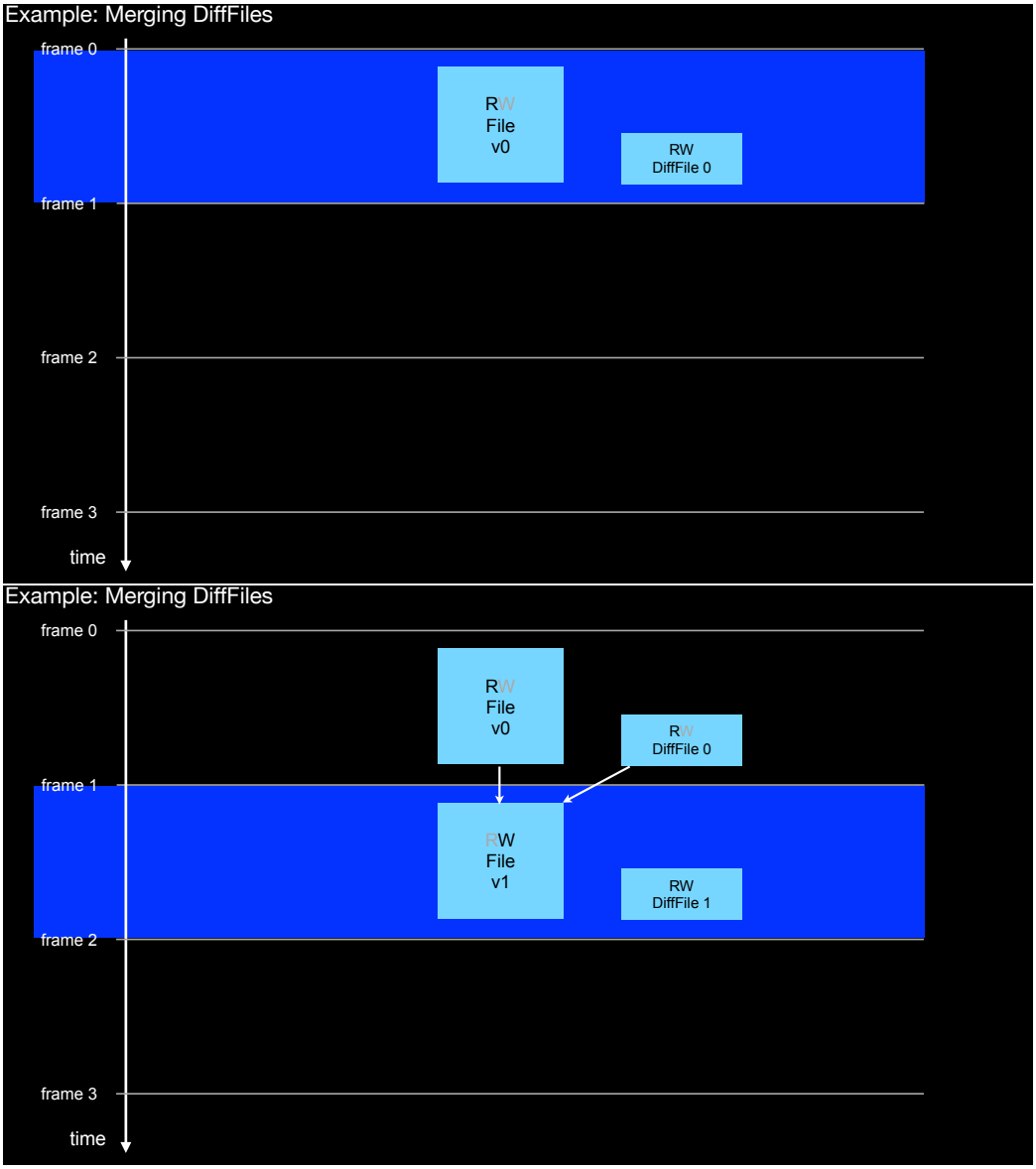


Figure 1.21: Different phases in the merge process: the situation before starting the merge (a read-only file v0 and its DiffFile 0) vs the situation during the merge (the DiffFile 0 is set to read-only as well. Both the read-only file v0 and the DiffFile 0 are merged into a new file v1. Concurrently we collect changes in a new DiffFile 1.)

operations may be redirected to only consider file v1 and diff file 1. Then diff file v0 and diff file 0 may be deleted. Now, we are back at the initial situation with one file and one diff file. Eventually, once diff file 1 has grown, we may repeat the entire process.

Q&As

1. Using Differential files, we

   (a) create a new file by merging the original file and another file, storing the found differences in a separate file.

   (b) collect the changes in a separate file, and eventually merge with the original file.

(c) merge two files together, if their contents differ.

(d) collect the differences of two files and create a merged file not containing these differences, i.e. the complement of the symmetric difference.

2. For differential files, the following pattern applies:

(a) all levels are equal pattern

(b) write-back pattern

(c) batch pattern

(d) data redundancy pattern

(e) copy on write pattern

(f) merge on write pattern

3. The differential file is

(a) at the same time the incremental backup file as well.

(b) at the same time the snapshot backup file as well.

(c) the buffer for the updates.

(d) the buffer for the most recently read pages.

4. Merging the differential file

(a) corresponds to creating a snapshot of the data.

(b) increases the fragmentation of the data.

(c) can only be applied on the file level.

5. To retrieve data from a table organized with differential files:

(a) we have to read one read-only file only.

(b) we have to read one read-only file and one read-write file.

(c) we have to read one read-write file only.

(d) we have to read two read-only files and one read-write file as well.

(e) we have to read up to two read-only files and one read-write file as well.

6. In the Differential File method a differential file can be

(a) read-only

(b) read-write

1.3.9 Logged Writes, Differential Files vs Logging

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

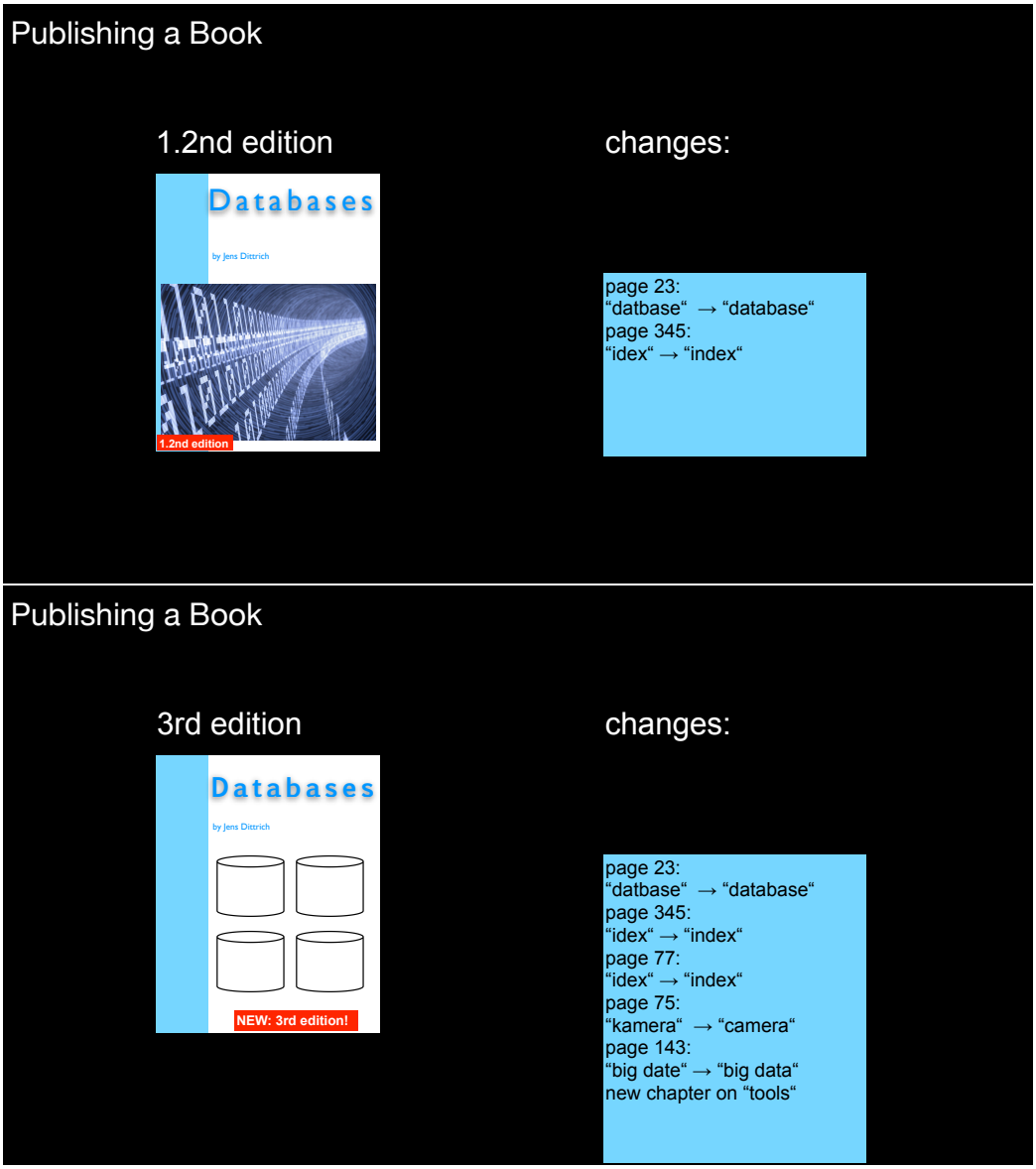


Figure 1.22: Publishing a book using logged writes: changes are not merely collected but every change is also applied to create a new (sub-)edition.

logging

differential files

What is the difference of logging and differential files?

The major difference is that in logging the file is not read-only and kept up-to-date at all times. The log is a redundant mechanism to trace all changes applied to the file.

What are its pros and cons?

Similarly to differential files, logging may be applied at any storage granule: entire databases, files, indexes, tables, blocks (in particular for media that has faster reads than writes like flash memory), etc. Moreover, this method can also be applied on different layers of the storage hierarchy, not only in-between disks and main memory. In contrast to differential files, in logging read-operations are relatively cheap as at all times only one structure has to be considered: the file. And just like in differential files, the log corre-

sponds to an incremental backup that can be archived or shipped, e.g. in a distributed system to synchronize the state across multiple databases. If the log is never pruned, the log contains all changes ever send to the database, i.e. "the log is the database". In that case we could regard the file (or the database if that is the granule used) as a compressed version of the log. We will look in more detail at logging in Section 6.1.2.

Is it possible to combine logging and differential files?

Absolutely, one use-case is to use logging where the read-write file is (internally) implemented using differential files.

Q&As

1. Logging is a technique where we can apply the:
   (a) all levels are equal pattern
   (b) write-back pattern
   (c) batch pattern
   (d) data redundancy pattern
   (e) copy on write pattern
   (f) merge on write pattern
2. To obtain all data belonging to a table:
   (a) it is sufficient to read the log file only.
   (b) it is sufficient to read the database only.
   (c) we have to read the database and the log as well.
   (d) we first have to merge the log into the database.
3. To obtain all data from a table it is usually more efficient:
   (a) to read the log file.
   (b) to read the database.
   (c) to read the database merged with the tail of the log.
4. In case of logging we write sequentially to the:
   (a) log
   (b) database
5. In case of logging we route updates to the:
   (a) log
   (b) database
6. The log is:
   (a) a snapshot of the database
   (b) an incremental backup

7. The benefits of logging over differential files are:
   (a) smaller storage space requirements
   (b) no random I/O
   (c) the possibility of restoring the database to any point of time
8. When combining logging and differential files:
   (a) we collect the updates in the log and in the differential file(s) as well.
   (b) we collect the updates in the differential file(s), and merge them with the log, before applying them to the database.
   (c) we direct the results of the merging to the log.
   (d) we get a non-functioning system.
9. The log file can be
   (a) read-only
   (b) read and allow for writing data at any position
   (c) read and allow for appending data

Exercise

Consider a read-only database using differential files to handle updates to tables on a granularity of records, i.e. if a record changes, the new version of the record is stored in a differential file. You are given a dataset of 1 TB already loaded in the read-only DB. Notice: 1 KB = 1024 Bytes. In addition,

- All I/O-operations are sequential (no seek time, no rotational delay, ignores random I/O, yet simplifies your calculation).
- I/O-operations can be done at a speed of 500 MB/s.
- Capacity of the differential file DF_0 is 64 MB.
- Data access is sequential in differential files.
- Record size is 128 Byte.
- Only updates to records in the existing RO-DB happen! Updates are uniformly distributed over the records in the (initial) RO-database, Neither inserts of new records nor updates to records existing already in some differential file will happen (however, this may not be exploited in the calculations of the merge strategies and also the algorithm must not rely on this property, in particular Strategy 2).

Now consider the following merge strategies:

Strategy 1: When the differential file DF_0 is full, merge the read-only DB and the differential file DF_0 immediately and start a new differential file DF_0'.

Strategy 2: Keep a sequence of differential files $DF_0, \ldots, DF_N$ where the capacity of DF_{i+1} is twice the capacity of DF_i. Whenever the capacity of a differential file DF_i is reached, it is merged into DF_{i+1}. If DF_{i+1} is empty, the storage space used by DF_i is simply reassigned to DF_{i+1} without performing an actual merge. For a suitable $N > 0$, DF_N can be considered the read-only DB.

Assume 960 MB of updated database records have been inserted. For both strategies (1)&(2) determine the following:

(a) number of merges for each differential file,

(b) size of the differential file(s),

(c) size of the read-only DB,

(d) total merge time,

(e) best, worst, and the average query time.

1.3.10 The No Bits Left Behind Pattern

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

Why shouldn't we leave any bits behind? Or in other words: why should we avoid wasting bits?

wasting bits

The main goal here is to avoid loading cold data into any layer of the storage hierarchy. Cold data refers to data that is actually not required to perform a particular operation. Hot data refers to the data requested by an ongoing operation. Typically, this loading of cold data happens when hot and cold data sits on the same storage granule, e.g. a page, and for whatever reason both the hot and the cold data is loaded both to a higher layers of the storage hierarchy. For instance, as data from disks can only be loaded in the granule of pages, and if that page contains cold and hot data at the same time, we still have to load the entire page into the database buffer. This is wasting storage space in the DB-buffer.

What does this mean for data layouts?

data layouts

We should (try to) layout data in a way such that we avoid loading cold data. Obviously, this cannot always be achieved easily as it also highly depends on the workload.

Why would I cluster data with spatial locality on the same virtual memory page, disk page, disk sector, cache line?

spatial locality

virtual memory page

disk page

disk sector

cache line

To fix the problem of wasting storage for cold data explained above: addresses that are referred to within short periods of time should minimize their spatial distance. This

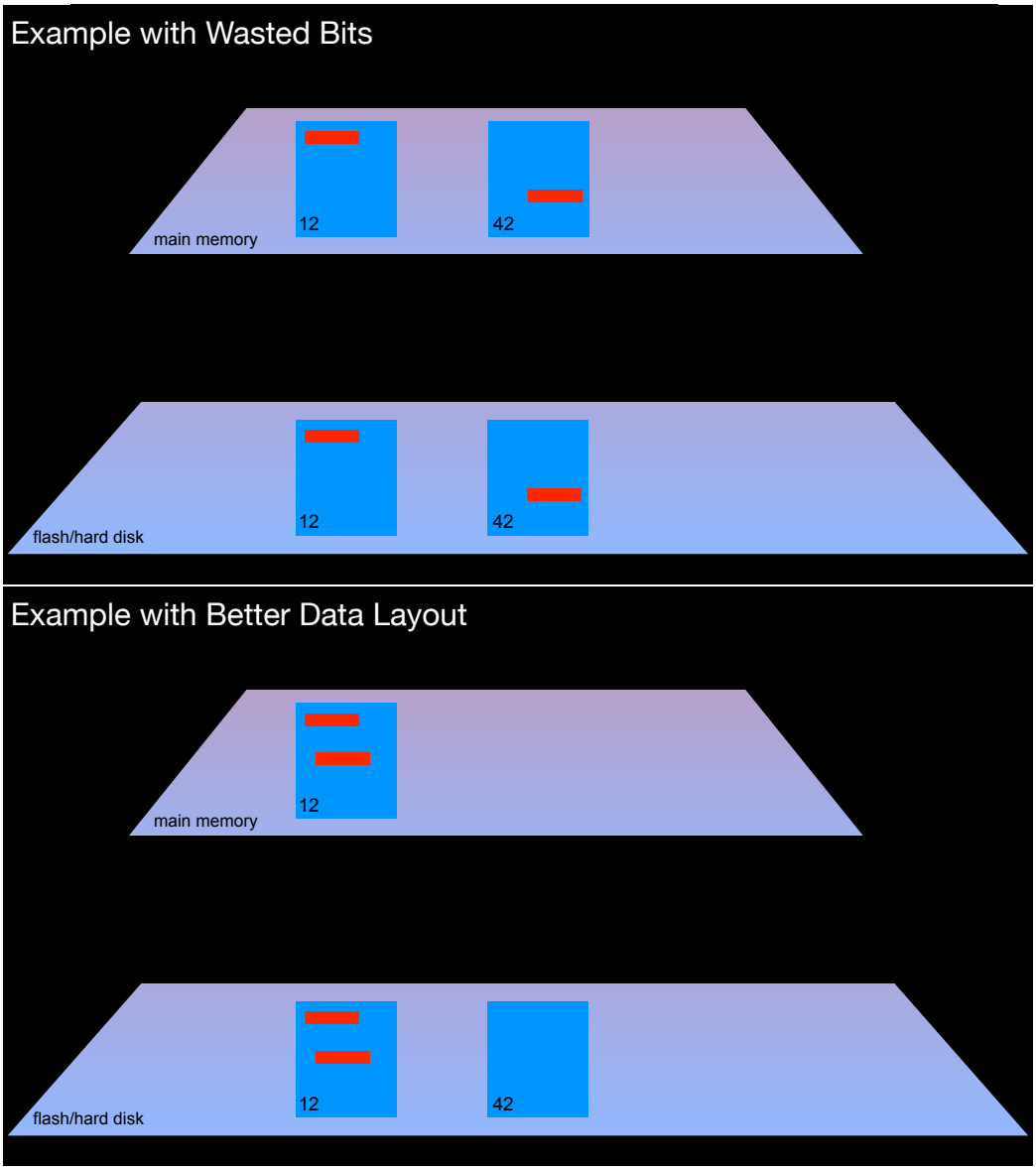


Figure 1.23: Wasting bits by distributing hot data over different pages vs keeping hot data on the same page

implies that they should also minimize the number of boundaries across storage granules. For instance, if two rows are referenced very often at the same time, however those rows reside on different pages, it is worth considering to place them on the same page. Like that for these two rows their is no boundary w.r.t. pages anymore. Still within that page, the two rows may still not reside on the same disk sector (recall: a page is typically a multiple of a disk sector). So, eventually you may decrease the distance of the two rows even further by placing them on the same disk sector. Still, the two rows may be loaded into different cache lines. So again, eventually you may further decrease their distance, i.e. by placing them within a cache line granule. So, basically, the storage layout may be adjusted dynamically to follow the data references of your program (or database management system). See also Section 1.3.1.

Why would I keep data with little spatial locality on different virtual memory pages, disk

pages, disk sectors, cache lines?

If data has little spatial locality, there is no use in keeping that data on the same storage granule.

Q&As

1. How can you avoid to load data into main memory containing at the same time data items that are heavily used by the CPU and data items that are only infrequently used by the CPU?
   (a) Rearrange data items on disk such that frequently used data items are clustered on disk.
   (b) Rearrange data items on disk such that frequently and infrequently used data items are uniformly distributed on disk.
   (c) Compress data as much as possible.
2. How can you make better use of the available bandwidth between memory and caches (or disk and memory) ignoring CPU costs?
   (a) Compress data as much as you can.
   (b) Only read the bytes you are interested in from a cache line (or memory page).
3. When accessing a single byte in main memory, ...
   (a) the whole cacheline containing the byte is brought into the caches.
   (b) only the surrounding word (64 bits) is loaded from main memory.
4. Assume we have a database with a buffer manager and data is in row layout. Accessing a single tuple of a relation ...
   (a) will bring the whole page containing that tuple from disk into the database buffer.
   (b) will only load the needed tuple from disk.

1.4 Virtual Memory

1.4.1 Virtual Memory Management, Page Table, Prefix Addressing

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Literature:
[PH12], Section 5.4

Learning Goals and Content Summary

virtual memory

How are virtual memory addresses translated to physical addresses?

address virtualization

Virtual memory addresses are translated to physical addresses by a combination of software and hardware components. The entire process is also called address virtualization. The central software components are the page table which is maintained in main-memory just as any other data. However, some of its entries are cached in a special hardware cache coined translation lookaside buffer (TLB), see also Section 1.4.2. Address translation is supported by hardware through a memory management unit (MMU). Similar memory mapping problems and techniques are used in the storage layer of a DBMS when implementing rowIDs. As a rowID must identify the location of a particular data item, it makes sense to not store actual physical offsets to memory but rather virtualize those addresses through a page table similar to the one used in virtual memory management.

How does virtual memory address translation work?

A virtual memory address is translated into a physical memory address as follows: a virtual address of k bits is split into two parts: (1) a prefix having l bits and (2) a suffix having $k - l$ bits. The prefix is interpreted as a virtual pageID. The suffix is interpreted as the offset inside the physical page pointed to by that virtual page. Therefore, in order to translate a given virtual address, we need to replace the prefix by the prefix of the physical page ID actually pointed to. This yields a new physical address which has the same suffix like the virtual address, but a new prefix.

page table

What is the role of the page table?

The page table maps virtual prefixes to physical prefixes. It may be organized in different ways and is kept in DRAM, however some of the data is cached in TLB.

prefix addressing

What is prefix addressing?

In prefix addressing the prefix of the virtual memory address is interpreted as the path into a radix tree. For instance, in a multi-level tree (see Figure 1.24), if the prefix has $l = 10$ bits, we may further divide this prefix into 2+4+4 bits. We start at the root

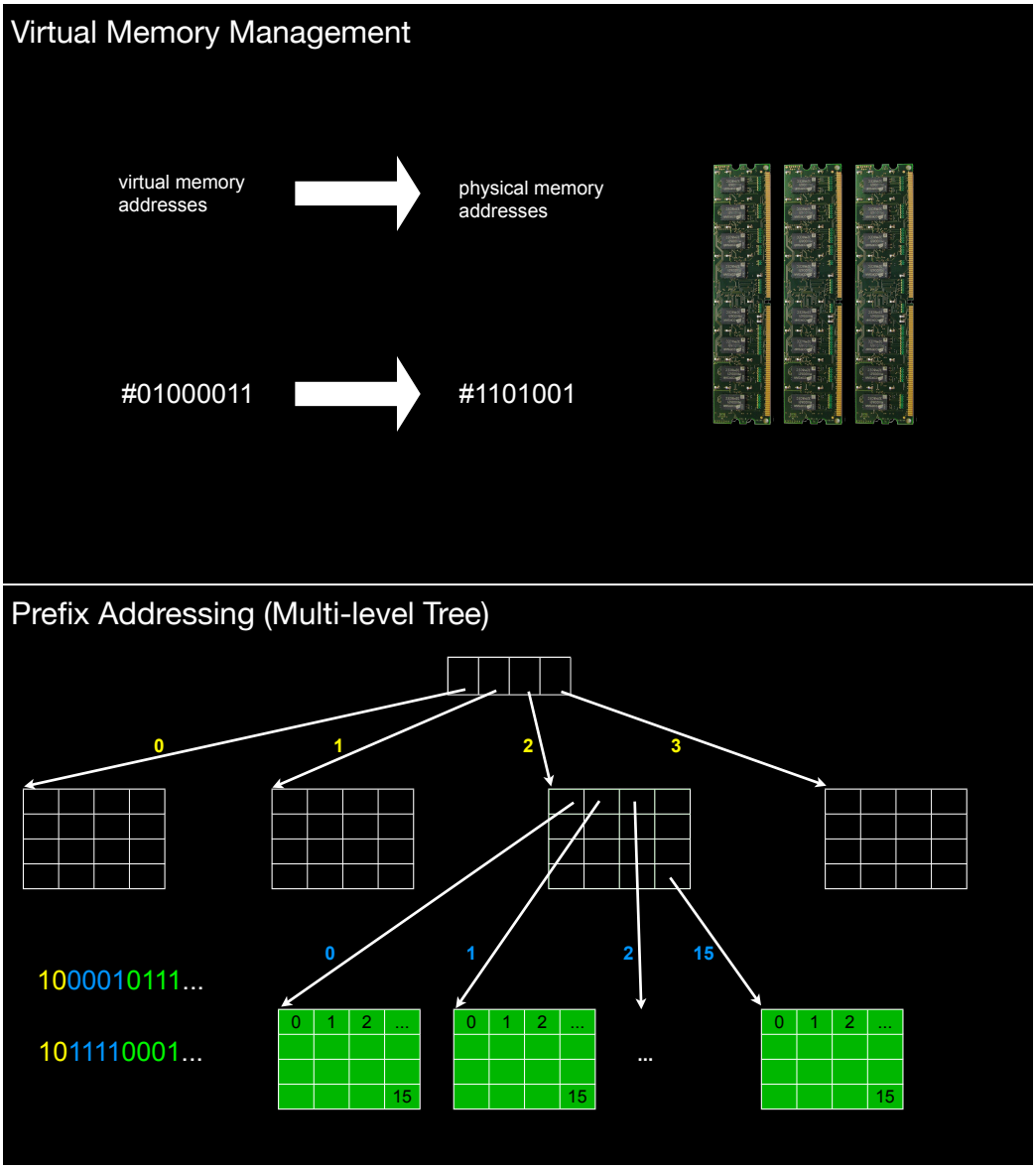


Figure 1.24: Virtual memory management maps virtual address to physical address. This mapping is often implemented using a (multi-level) prefix tree.

node which has at most $2^2 = 4$ entries. The first 2 bits (yellow numbers) of the suffix determine the child node. In that child, which has at most $2^4 = 16$ entries, the next 4 bits (blue numbers) determine the address of the next child to inspect. Again in that child, which has at most $2^4 = 16$ entries, the last 4 bits of the suffix (green numbers) contain the actual physical address. That physical address is the value stored at that particular offset within this node.

What exactly is an offset?

offset

An offset defines the distance from a starting address. In the context of this video, prefix of a certain length define such a starting address, the remaining suffix then defines the offset to that starting address. For instance, if you consider the first two bits to define a starting address, the prefix tree translates this to the starting address of one of the four segments. The remaining 16 bits define the offset from that starting address within that

segment. Recursively, if you address a particular page within a segment using a 6 bit prefix, that 6 bit-suffix lead you to the starting address of a particular page within that segment (again through the prefix tree). Any other address within that page (the offset) can be addressed by the remaining 12 bit suffix.

Q&As

1. Why do operating systems use virtual memory?
   (a) To separate the address spaces of different processes.
   (b) To allow for compiled code to contain fixed addresses, even though the physical location in memory is only known after the process was started.
   (c) To allow for a larger address space than the actually available main memory.
2. How are virtual memory addresses translated to physical addresses, if using a single level of translation, 32 bit addresses, and 4KB pages?
   (a) The first 20 bits are used to find the page table entry, that contains the physical page number and that physical page number is appended by the 12 bit suffix of the virtual address.
   (b) The first 20 bits are used to find the page table entry, that contains an arbitrary physical address (i.e. an address not necessarily aligned to the page size) and the 20 bit suffix has to be added to the physical address.
   (c) The first 12 bits are used to find the page table entry, that contains the physical page number and that physical page number is appended by the 12 bit suffix of the virtual address.
   (d) The first 12 bits are used to find the page table entry, that contains an arbitrary physical address and the 20 bit suffix has to be added to the physical address.
3. Given 8-bit virtual addresses and a page size of 64 bytes. How many virtual pages are there per virtual address space?
   (a) _____
4. How do modern CPUs support virtual memory?
   (a) They provide memory management units that perform the translation in hardware.
   (b) They provide a special cache, the TLB, to speed up address translation of addresses frequently accessed virtual pages.
   (c) They provide a special co-processor to speed up address translation of addresses in the same virtual page.
   (d) They provide a special cache, the TLB, to speed up address translation of addresses frequently accessed physical pages.
5. How many bits from the 64-bit virtual addresses are actually used by current $x86(64)$ CPUs to translate address prefixes from virtual pages to physical pages? Notice that only 48-bits of the 64-bit address space is used in any case, i.e. how many of those 48 bits are used for the page translation prefix for a given memory page size?

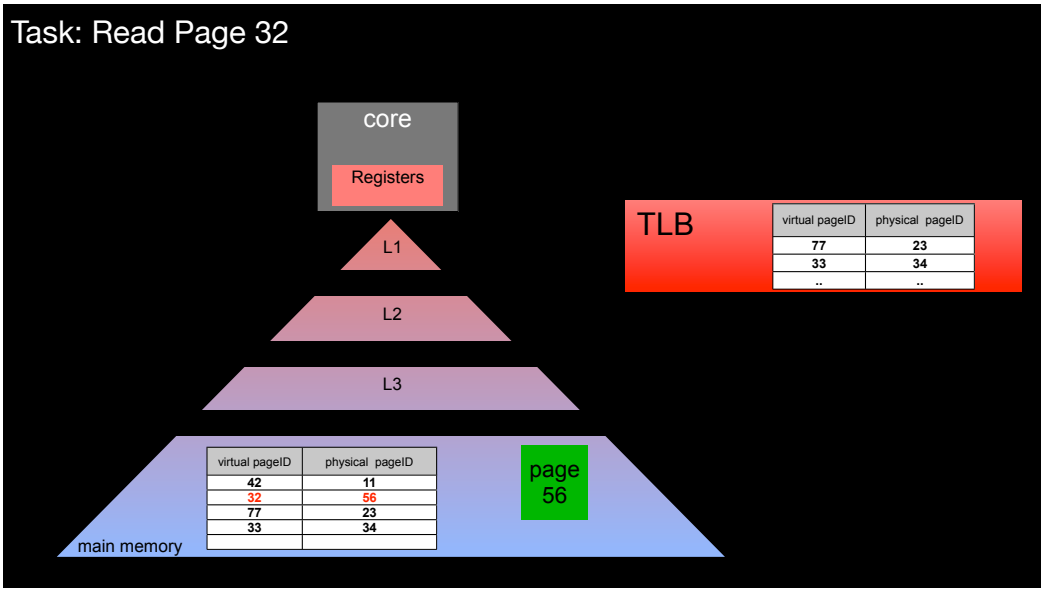


Figure 1.25: Translation lookaside buffer (TLB) and TLB-misses

(a) 12 for 4KB memory pages

(b) 36 for 4KB memory pages

(c) 27 for 2MB memory pages

(d) 18 for 1GB memory pages

1.4.2 Retrieving Memory Addresses, TLB

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

What happens when referencing a specific virtual memory address?

virtual memory

The TLB is inspected whether it has a cached version of that address. If that is not the case, the page table has to be searched for that address.

What kind of translation lookaside buffer (TLB) misses and cache misses may occur?

translation lookaside buffer

TLB

cache miss

If a particular address is not available in the TLB, looking up the page table may lead to cache misses (at multiple levels of the storage hierarchy). Under the assumption that the page table is entirely available in main memory, an address lookup may therefore require the time it takes to randomly fetch data from main memory. Only after that, and using the physical address just retrieved, the actual data is retrieved. This may again lead to cache misses. Bottom line: in a storage hierarchy using virtual memory address translation, we do not only observe cache misses due to fetching data, but additionally

due to fetching addresses.

Q&As

1. What is the translation look-aside buffer used for?
   (a) It stores mappings from virtual page addresses to physical page addresses.
   (b) It stores mappings from physical page addresses to virtual page addresses.
2. How many last level cache misses can occur when reading a single value from memory if three-level address translation is used? Notice: if an architecture has L1, L2, and L3, then L3 is called the last level cache (LLC).
   (a) _____
3. How can the pressure on the TLB be reduced?
   (a) By using larger pages.
   (b) By disabling address translation (assuming this is possible).
   (c) By addressing a given page at most once.

Exercise

A current 64-bit UNIX operating system uses only the lowest 47 bits for memory addressing, the other 17 bits are filled with 1s. Let's assume your OS uses 64 KB memory pages and uses shadow-storage (without merge-on-write) as the strategy for indirect writes.

Assume an empty page table initially, and that write operations to unmapped virtual addresses allocate a new physical page (numbered by $P_1, ..., P_N$). Recall that a read-operation to a page that was never written to before will be redirected to a system-wide, globally visible read-only page filled with zeros. Only if the first write operation happens to a particular physical page, a page fault happens and a writable page is alocated and an entry inserted into the page table. Also notice that whenever a process forks a child process that child process receives a copy of its parent's page table.

Consider the following sequence of operations to the specified virtual memory addresses executed by processes 1 and 2 in that order. Notice that 0xFFFF,FEAD,0001,0000 is a multiple of 0x1,0000 (=64 K base 1024).

(1) process 1: write 0xFFFF,FEAD,0001,0004

(2) process 1: read 0xFFFF,FEAD,0002,0100

(3) process 1: write 0xFFFF,FEAD,0002,0004

(4) process 1 forks child: process 2

(5) process 2: write 0xFFFF,FEAD,0001,0002

(6) process 2: write 0xFFFF,FEAD,0001,0004

(7) process 2: write 0xFFFF,FEAD,0007,1234

(8) process 1: write 0xFFFF,FEAD,0001,0008

(9) process 1: write 0xFFFF,FEAD,0002,0142

(10) process 1: read 0xFFFF,FEAD,0007,4321

(11) process 1: write 0xFFFF,FEAD,0007,4321

Your task is to show the mappings stored in the page table(s) after performing each of the above operations.

Chapter 2

Data Layouts

2.1 Overview

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

What are the principal mapping steps to map a relation to a device?

mapping steps
relation
device

The different mapping steps as displayed in Figure 2.1 are

linearize
serialize
devirtualize
materialize

(1a) linearize: Two-dimensional relations (sets of tuples) are mapped to a one-dimensional sequence of values,

(1b) serialize: a one-dimensional sequence of values is mapped to bytes on virtual pages,

(2) devirtualize: virtual pages are mapped to physical pages,

(3) materialize: physical pages are mapped to storage devices.

Which of those steps is related to data layout?

data layout

Steps 1a and 1b.

Why linearize values?

A relation is a set and hence does not define an order. A relation can be considered a two-dimensional address space, i.e. each attribute value can be uniquely addressed by the pair (rowID, attribute name). In contrast, the address space in memory (be it on physical or virtual pages) is a one-dimensional sequence of bytes, it has an order. Therefore we have to force tuples' attribute values into a particular order. How we define this order may have a tremendous impact on query performance in the database system.

Why serialize values?

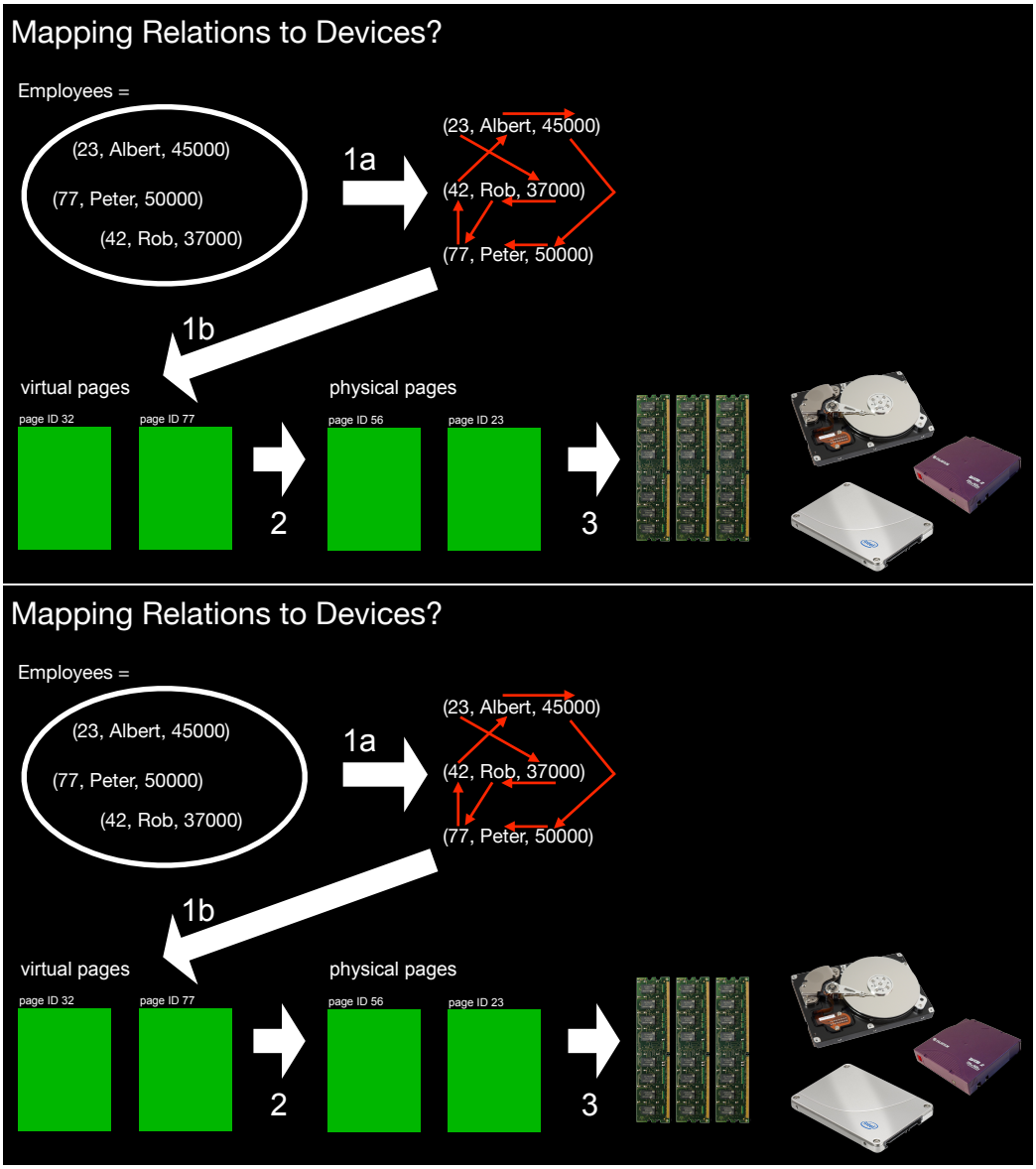


Figure 2.1: The different mapping steps required when mapping two-dimensional relations to storage devices.

We have to serialize values (step 1b) in order to convert a higher-level representation, say a sequence of values like integers or strings, into a lower-level representation, say a sequence of bytes, which can then be stored on virtual pages.

What is done first: linearize or serialize?

Whether we first perform step 1a (linearize) and then step 1b (serialize) or alternatively perform both steps in a single operation is an implementation decision. Just like step 1a, step 1b may also impact query processing later on depending on the type of serialization we use, see also Section 2.4.

Q&As

1. Linearizing values incorporates:
   (a) assigning an order to tuples without assigning any particular order to the data values
   (b) sorting the tuples on the key attribute
   (c) removing duplicates to confirm to the set-semantics of the data structure
   (d) assigning an order to individual data values
2. The linearization order:
   (a) has a huge impact on the query performance
   (b) does not change the result set of a query
   (c) confirms to the physical order on hard disk
   (d) confirms to the physical order in RAM
3. Order the following concepts from left (abstract) to right (physical):
   (a) relations - virtual pages - physical pages - storage blocks
   (b) relations - tuples - fields - storage blocks
   (c) virtual pages - physical pages - tuples - storage blocks
4. The correct order of the mapping steps is the following:
   (a) linearize - serialize - virtualize - materialize
   (b) linearize - serialize - devirtualize - materialize
   (c) virtualize - serialize - linearize - materialize
   (d) linearize - deserialize - devirtualize - materialize
5. The linearization order allows for:
   (a) storing all fields of a tuple contiguously
   (b) storing all values of a given attribute contiguously
   (c) storing only some of the attributes of a tuple contiguously
   (d) assigning an arbitrary order of data values even across tuples

2.2 Page Organizations

2.2.1 Slotted Pages: Basics

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

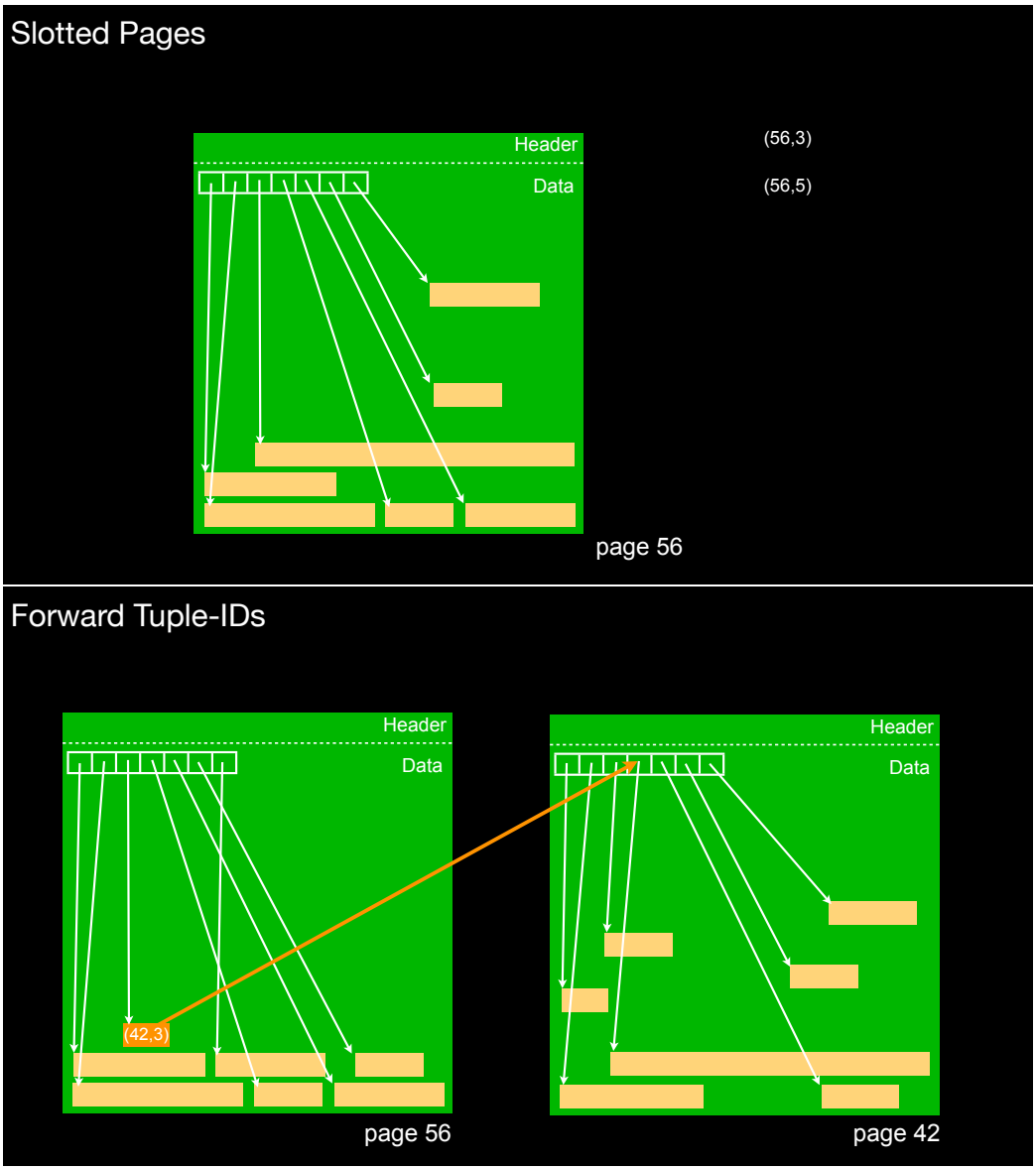


Figure 2.2: Slotted pages and Forward Tuple-IDs

slotted page

What is the core idea of a slotted page?

> "All problems in computer science can be solved by another level of indirection."
>
> (Butler Lampson).

Slotted pages are yet another instance of this quote as the core idea of slotted pages is to introduce an indirection to hide physical addresses from users. Recall, that virtual pages (see Section 1.4) introduce an indirection to hide physical page addresses, slotted pages virtualize addresses inside the page. Therefore, slotted page are just another application of address virtualization.

address virtualization

Slotted pages allow us to address chunks (which may be rows, but don't have to)

inside pages without having to expose physical offsets of chunks outside the page. This is achieved by keeping an array (aka slot array) of physical offsets inside each page. This array plays a similar role as the page table in virtual memory. A (pageID,slot)-pair may be used to identify any chunk kept on a page. Here, the slot simply denotes an index into the array. The pageID is the virtual address of the page, it is translated by virtual memory management. The slot is the (virtual) offset within that page, it is translated by by the slot array. A (pageID,slot)-pair can easily be implemented by concatenating pageID and slot to a single value. If the chunks stored in slotted pages are rows, we term that concatenation a rowID.

slot array

slot

rowID

So, in summary, virtual memory virtualizes the prefix (pageID) of a memory address. In contrast, the suffix (slot) virtualizes the offset inside a page.

What is a slot?

A slot is a particular entry in the slot array at the beginning of each page pointing to a particular chunk. The chunk pointed to may be a

1. row (if the data is in row layout) or
2. a fraction of a row (if a column grouped layout is used or a single row exceeds the size of the page) or
3. a column (if column layout is used) or
4. a fraction of a column (if PAX is used or the column exceeds the size of the page) fraction of a column.

What is the relationship of slotted pages to physical data independence?

physical data independence

The classical notion of physical data independence makes a database schema independent from the storage structures used underneath. This is achieved by introducing an indirection: in terms of the user front-end (SQL) the data and metadata stored in a database system is not coupled to a particular physical organization. Those organizations can (or should) be interchangeable but not tightly coupled to the data. The same indirection technique is used in slotted pages: the indirection of offsets (the WHAT) through slots hides a physical property (the HOW), i.e. the offset of a chunk inside a page. Thus, this is a lightweight form of physical data independence.

Is it possible to move data within a slotted page? Like how?

Absolutely, just move the chunk to a different unoccupied position within that page and update the offset in its slot (ideally in an atomic operation). This does not change the (pageID,slot)-pair of that chunk.

What is a forward tuple-ID?

forward tuple-ID

A forward tuple-ID (a more general and better name would be a forward chunk-ID), is a workaround which allows us to move chunks to another page without changing their (pageID,slot)-pair. For instance, assume the chunks stored on each page are tuples. If we want to move a tuple from, say page 56 to page 42, we place a special forward tuple-ID on page 56 which redirects all requests to that tuple to page 42 (the slot array of page 42 to be precise).

What are forward tuple-IDs good for?

Again, like using forward tuple-IDs we do not have to change a particular (pageID,slot)-pair.. This may be useful in cases where these pairs are used outside the page in other data structures in the database system, e.g. in indexes. We save the costs for updating all those references.

What might be a problem when using forward tuple-IDs?

The additional lookup comes at a price: when looking up such a chunk we have to inspect two pages: the old page, page 56 in the example containing the forward tuple-ID, and the page actually containing the data, page 42 in the example. So, the general trade-off here is: by introducing forward tuple-IDs we save update costs, as we do not have to update all references of that chunk anymore. However, at lookup time, we may need an extra lookup to get to the actual page. When to use forward tuple-IDs or not heavily depends on the workload of the database system. Yet, as a rule of thumb, you should not use more than one indirection through forward tuple-IDs, e.g. a forward tuple-ID pointing to a forward tuple-ID pointing to a chunk.

Q&As

1. Slotted pages:
   (a) provide an indirection which has a similar effect as logical data independence
   (b) provide an indirection which has a similar effect as physical data independence
   (c) allow for marking data as deleted rather than actually deleting it
   (d) allow for logging changes done to the actual data
2. Slotted pages consist of:
   (a) header
   (b) slots
   (c) data chunks
   (d) footer
3. In case of slotted pages to access a given tuple we need the following:
   (a) pageID
   (b) table name
   (c) column name
   (d) slot number
4. The following components have always a fixed starting position within a slotted page:
   (a) header
   (b) the slot-array
   (c) the individual data chunks
   (d) footer

5. When moving a tuple inside a slotted page we have to:
   (a) update the offset to the slot-array in the header
   (b) update offsets inside the slot-array
   (c) change the slotID of that page
   (d) none of these
6. When moving a tuple to another slotted page using a forward reference we have to:
   (a) update the header in the source page
   (b) update the slot-array in the source page
   (c) update the slot-array in the target page
   (d) update the data chunk in the source page

2.2.2 Slotted Pages: Fixed-size versus Variable-size Components

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[RG03], Section 9.6.2

Learning Goals and Content Summary

fixed-size components

How to organize fixed-size components on a slotted page?

slotted page

In order to store fixed-size components we simply use linear addressing inside that page.

slot array

Where to store the slot array on a slotted page using fixed-size components?

We do not need it anymore. The array got replaced by a function translating the suffix of the rowID to an offset inside that page.

linear addressing

What is linear addressing?

Linear addressing means that there is a linear relationship between the suffix used as the input to the function and its output.

variable-sized components

What can we do with variable-sized components?

Basically, for each row, we partition the attributes into two sets: the fixed-size attributes and the variable-sized attributes (Notice that this is an application of both vertical partitioning, see Section 2.3.3 and PAX, see Section 2.3.4). The fixed-size attributes are stored from bottom to top just like before using linear addressing. In addition, we add an extra artificial attribute for each row in the fixed-size part with an intra-page offset pointing to the variable part which is stored from top to bottom. Like that the fixed-size

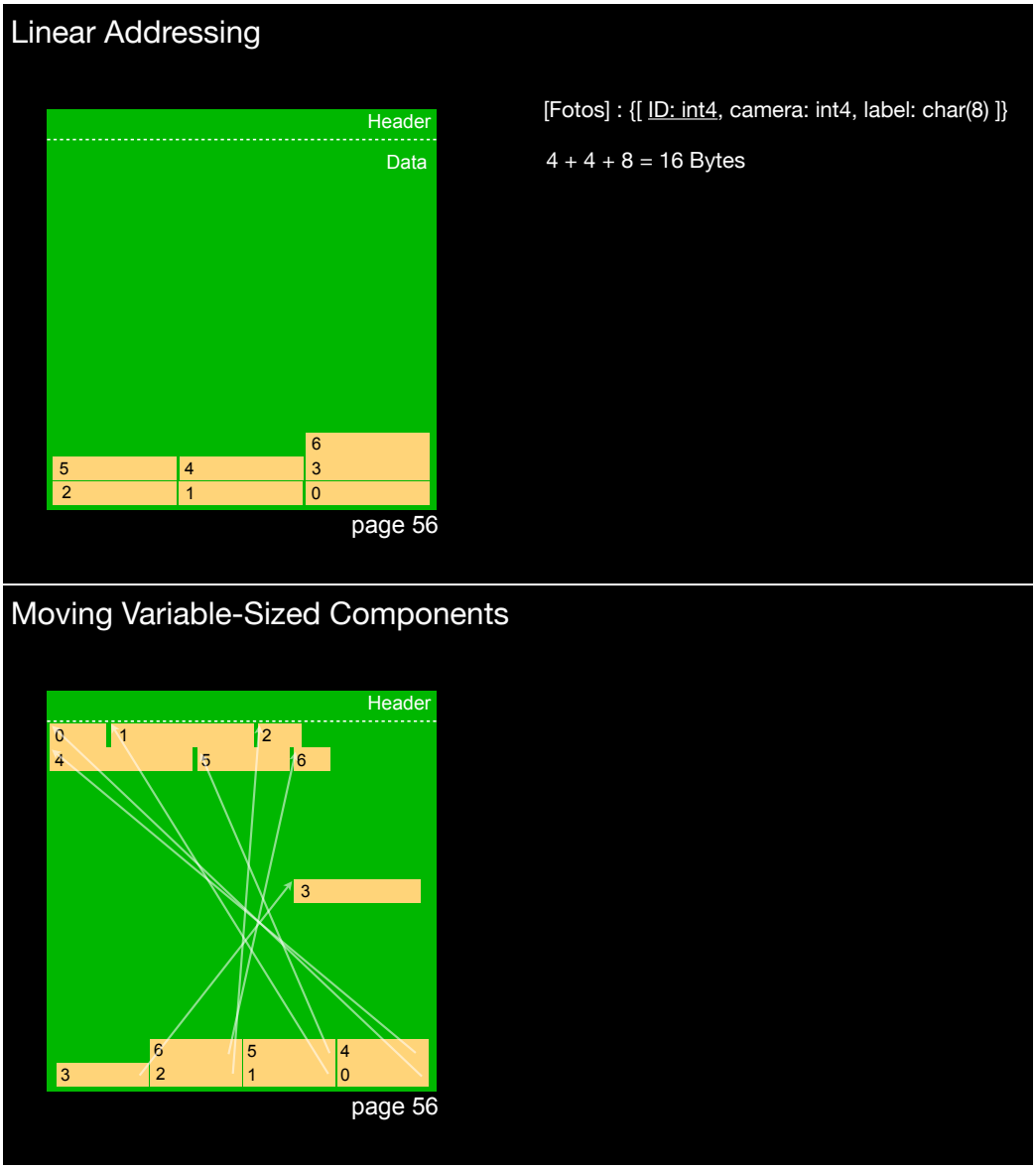


Figure 2.3: Linear Addressing and Variable-Sized Components

part takes over the role of the slot array. However, the slots are only used to address the variable-size fraction of each row.

Can we move the fixed-size and variable parts around on a page?

The fixed-size parts cannot be moved except if we change the function used for linear addressing inside a pages. The variable-size components may be moved just like in the basic slotted pages method. The offset that was previously kept in the slot array is now part of the fixed-size part of a tuple.

Q&As

1. Consider the following table: CREATE TABLE person (id INTEGER4, name CHAR(52), city CHAR(200)), and a slotted page with a size of 4 KB. What is the offset of the second tuple?

   (a) 3840

   (b) 3584

   (c) 512 + size of the header

   (d) 256 + size of the header

2. In case of a 4 KB slotted page how many bits do you need for each offset if you want to address every byte?

   (a) 12

   (b) 32

   (c) 16

   (d) 64

3. Consider the following table: CREATE TABLE person (id INTEGER4, name CHAR(52), city CHAR(200)), and a slotted page with a size of 4 KB, with a 100 byte header. How many tuples can you store in a single page?

   (a) 8

   (b) 14

   (c) 15

   (d) 16

4. Which of the following components have a fixed position within a slotted page?

   (a) header

   (b) fixed-size components of tuples

   (c) variable-size components of tuples

   (d) footer

5. Variable-size components of tuples:

   (a) are stored together with the fixed-size components

   (b) are stored where the slot-array would be stored

   (c) require more storage space than their actual net size

   (d) require storage space equal to their maximal size

Exercise

Consider the following table definition:

```
CREATE TABLE person (
    id INTEGER,
    name VARCHAR(50),
    city VARCHAR(100)
);
```

and a slotted page with a page size of 4 KB, with a 128 Byte header. Assume 4 Byte integers, linear addressing, and that variable-sized components are stored in a space-efficient way.

Insert the following tuples in the listed order into the same, initially empty page:

(1) (41, “Doe", “Merzig")

(2) (42, “Smith", “Eppelborn")

(3) (43, “Foobar", “Saarwellingen")

Show the page after the insertions. What is the total space occupied by the inserted tuples? What is the amount of free space in the page after the insertions?

2.2.3 Finding Free Space

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[RG03], Section 9.3.1

Learning Goals and Content Summary

segment

What is a segment?

A segment is a contiguous sequence of physical pages. Therefore, a physical page within a segment can be addressed linearly. It is exactly the same idea as linear addressing of chunks inside a slotted page. The only thing that changes is the size of the granules: We address physical pages inside a segment.

	What is considered a “page”?	**What is considered a “chunk”?**
Slotted Pages	page	chunk
Segments	segment	physical page

Table 2.1: Special-cases of memory indirection and virtualization

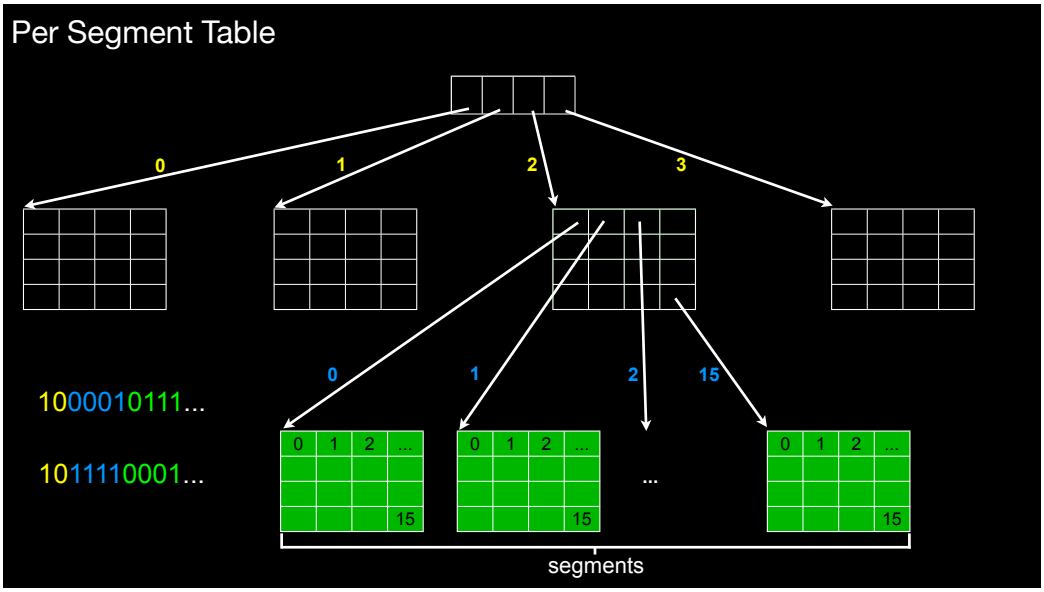


Figure 2.4: Per segment prefix table as a tree

Where do we store metadata about free space or any other metadata anyway?

free space

We need to keep track of how much space is left in our database and where that space is available. So, the general question is: where should we keep that metadata, i.e. data about the actual data stored in the database? Examples of metadata kept in a database system include: free space information, statistics, indexes, schema information and so forth. Most database systems keep that information in the so-called catalog which is itself a relational database with a vendor-defined schema. However, some metadata is also kept in different places for efficiency-reasons. This includes data about free space management. That metadata can be stored on a per page or on per segment level. Several database design techniques combine metadata and data on a granule that is smaller than the entire database. In other words, metadata for a page is kept on that page, or: metadata about a segment is stored on that segment.

metadata

catalog

This is again another example for fractal design which we will generalize in Section 2.3.5.

Q&As

1. Why do we need to manage the free space of the pages available in the database?
   (a) To make effective use of the available memory.
   (b) Because we cannot allocate more pages over time.
   (c) To leave no bit behind.
2. Why not store the exact number of free bytes per page in the free space management table?
   (a) The required space to store this number might be too high. It is often enough to know an estimation of the available space to decide if a new data item fits into that page.

(b) We should store the exact number. Otherwise we don't know if a new item can fit on the page.

3. Can we immediately write a new data item to a page that has enough available space?

   (a) Not necessarily. Maybe we need to defragment the stored items first, to create enough contiguous room.

   (b) Definitely.

2.3 Table Layouts

2.3.1 Data Layouts: Row Layout vs Column Layout

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

linearization

How could we phrase the tuple linearization problem formally?

Assume we have a set of tuples $T = \{t_1, \ldots, t_j, \ldots, t_y\}$ where all tuples $t \in T$ have the same schema, i.e. $t_j = (a_{1,j}, \ldots, a_{i,j}, \ldots, a_{x,j}) \in A_1 \times \ldots \times A_i \times \ldots \times A_x$ where A_i is called a domain.

We want to find a linearization function $L : \texttt{int} \times \texttt{int} \mapsto \texttt{int}$ which maps two-dimensional attribute values to a one-dimensional space, i.e. for each index (i, j) of an attribute value $a_{i,j}$ we assign a one-dimensional value z_k:

$$L(i, j) \mapsto z_k \tag{2.1}$$

such that

$$\forall (i_1, j_1) \mapsto z_{k_1}, (i_2, j_2) \mapsto z_{k_2}, i_1 \neq i_2 \vee j_1 \neq j_2 \Rightarrow z_{k_1} \neq z_{k_2} \tag{2.2}$$

In other words, equation 2.2 states that no two attribute values are mapped to the same z_k-value. Any linearization L fulfilling that equation is called non-redundant.

non-redundant

row layout

How do we linearize tuples into a row layout?

We define a linearization L_{row} with a lexicographical (aka recursive) ordering:

$$L_{\text{row}}(i, j) \mapsto L_y(j) \bigoplus L_x(i) \tag{2.3}$$

This means, each index (i, j) is mapped to a z_k-value where the first part of z_k (the prefix) is determined by a linearization function $L_y(j)$ defining the order among **rows**. The second part of z_k (the suffix) is determined by a linearization function $L_x(i)$ defining the order among **columns** *within one particular* ***row***.

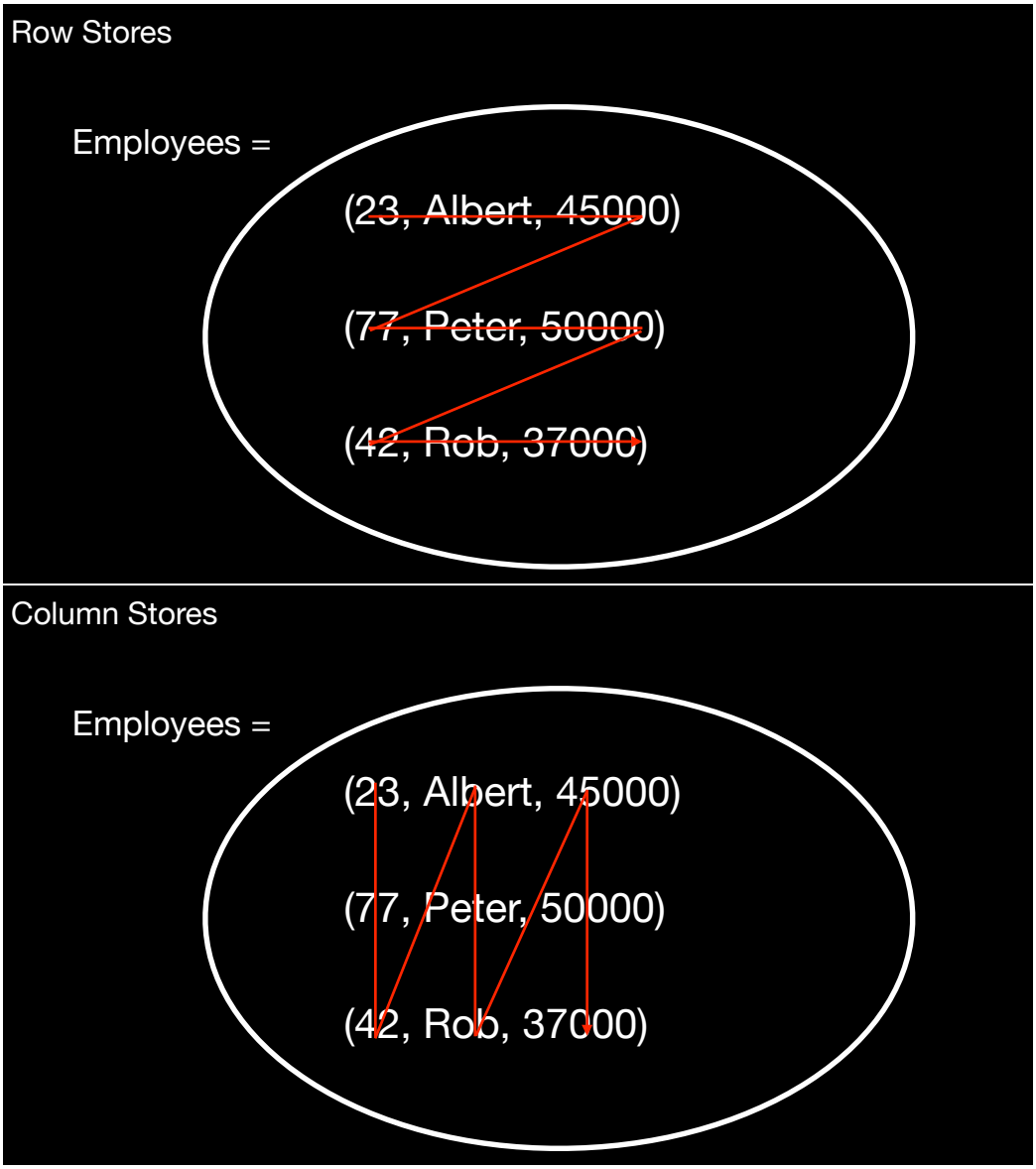


Figure 2.5: Row store vs column store: the major difference is the linearization order of attribute values.

How do we linearize tuples into a column layout?

column layout

We swap the roles of columns and rows in equation 2.3. This means, we define a linearization L_{column} with a lexicographical ordering:

$$L_{\text{column}}(i,j) \mapsto L_x(i) \bigoplus L_y(j) \tag{2.4}$$

This means, each index (i, j) is mapped to a z_k-value where the first part of z_k (the prefix) is determined by a linearization function $L_x(i)$ defining the order among **columns**. The second part of z_k (the suffix) is determined by a linearization function $L_y(j)$ defining the order among **rows** *within one particular* ***column***.

Why would we linearize tuples in row layout?

Row-wise linearization is useful for queries that need to access several attributes of each

tuple at a time but at the same time access only few rows. This is typically the case for insert-, update-, and delete-operations.

Why would we linearize tuples in column layout?

Column-wise linearization is useful for queries that need to access few attributes of each tuple, but at the same time have to access many tuples. This is typically the case for analytical queries, grouping and/or aggregating a large subset of the tuples of a table on few attributes.

Which type of layout is better for which type of query?

It depends on the attributes required to compute the result to a query. In particular the relationship of number (and size) of accessed attributes over the size of the entire row.

row store

What is a row store?

A row store is a database store that uses row layout to linearize data.

column store

What is a column store?

A column store is a database store that uses column layout to linearize data.

Q&As

1. When linearising tuples of a single relation, given the ordering of the tuples:
   (a) we can choose either row- or column layout for each tuple in the relation.
   (b) we can choose either row- or column layout for the whole relation.
   (c) we get the same linearisation order using any layout for relations containing a single attribute only.
   (d) we get the same linearisation order using any layout for relations containing a single tuple only.

2.3.2 Options for Column Layouts, Explicit vs Implicit key, Tuple Reconstruction Joins

Material

Video:	Original Slides:	Inverted Slides:
		

Learning Goals and Content Summary

correlated columns

What are correlated columns?

Two columns are correlated if the order of their attribute values w.r.t. their rowIDs is the same in both columns. In other words, assume two columns C_i and C_j containing attribute values of rows $r_0, \ldots, r_{N-1}$. Let SO be an arbitrary sort order of the sequence $r_0, \ldots, r_{N-1}$. Then, both C_i and C_j must contain their attribute values in the same sort order SO.

implicit key

What is an implicit key?

Figure 2.6: Implicit key vs explicit keys in a column store

An implicit key is a key that is not materialized but rather used as an input parameter to a function. For instance, in a column layout, if arrays are used to store fixed-size attribute values, the implicit key times the size of a single data value yields the offset within the array. This is the function computed implicitly by the programming language compiler. So again, this is another example of linear addressing (implicitly done by the compiler).

Could columns also be uncorrelated?

uncorrelated

Yes, however, then we need to make sure that we know the rowID for each attribute value.

What are the consequences of using uncorrelated columns? What is an explicit key?

explicit key

This implies that we need to store the rowID with each attribute value in each column. So, each attribute of the original table is now represented by a column having two attributes: an explicit key and the actual attribute value.

tuple reconstruction join

What is the impact of explicit keys on a tuple reconstruction join?

Tuple reconstruction joins get more expensive as they cannot rely on the same order of attribute values across columns. In contrast, in a layout using *implicit* keys for tuple reconstruction joins, a simple merge join, see Section 4.1.3, or a direct lookup, see Section 5.3.5, may be used.

How do we convert a data layout using explicit keys into a layout using implicit keys?

We simply sort all columns on their explicit key column. Then you throw away all explicit key columns and represent all data values in all columns in arrays. Be careful: this does not work when done in a purely relational model. As the relational model does not imply a sort order among tuples in a relation, we must ensure that data values are managed using sequences.

Q&As

1. In case of an uncorrelated column layout
   (a) the values in each column have to be ordered according to the same sort order.
   (b) the values in the individual columns do not correlate to the values of the key column.
   (c) the values at a given position within each column do not necessarily belong to the same tuple.
   (d) the values at a given position within each column must belong to the same tuple.
2. Assume a column layout with uncorrelated columns. Assume we reconstruct tuples by first sorting the columns on their explicit keys and then merging the sorted columns (this is called a sort-merge join). The following statements about this approach are true:
   (a) we have to sort the columns on the values and merge them into tuples.
   (b) we can directly merge the columns into tuples.
   (c) we have to sort the columns on the explicit key and merge them into tuples.
   (d) we merge the columns into tuples, and sort them on the explicit key.
3. Explicit keys are required for
   (a) tuple reconstruction in correlated column layouts.
   (b) preventing duplicate elimination in uncorrelated column layouts.
   (c) tuple reconstruction in column layouts where the values at a given position of each column do not necessarily belong to the same tuple.
4. The benefits of column layout with uncorrelated columns over column layout with correlated columns are:
   (a) faster tuple reconstruction
   (b) smaller storage requirements

(c) faster inserts, in case there is a sort order on the tuples of the column layout with correlated columns

(d) allows for sorting a subset of the columns independently

(e) faster full scans for SELECT *-queries

5. Whenever a table in column layout is used in a query

(a) the table has to be transformed into row layout.

(b) the whole tuple has to be reconstructed.

(c) a partial tuple reconstruction might be sufficient.

(d) the other tables in the query also have to be in column layout.

2.3.3 Fractured Mirrors, (Redundant) Column Grouping, Vertical Partitioning, Bell Numbers

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[RDS02]

Learning Goals and Content Summary

What is the relationship of fractured mirrors to row and column layouts? **fractured mirrors**

When using fractured mirrors the data is linearized in row and column layout redundantly, i.e. data is kept in row layout and column layout redundantly.

First, we need to drop the condition of equation 2.2 to allow for redundancy, also recall our discussion in Section 2.3. Any linearization L not fulfilling equation 2.2 is called redundant and marked L^{R}. Notice that L^{R} is not a function anymore in the mathematical sense as it returns a set of values. **redundant**

More formally : We define a linearization $L^{\mathrm{R}}_{\mathrm{frm}}$ with a lexicographical ordering:

$$L^{\mathrm{R}}_{\mathrm{frm}}(i,j) \mapsto \left\{L_{\mathrm{row}}(i,j), L_{\mathrm{column}}(i,j)\right\} \tag{2.5}$$

This means, each index (i,j) is mapped redundantly to two z_k-values: one in row layout, one in column layout. Obviously the z_k-values mapped to by $L_{\mathrm{row}}(i,j)$ and $L_{\mathrm{column}}(i,j)$ should be disjoint.

Why would fractured mirrors make sense?

This makes sense for a workload of queries where some queries are executed against a row layout and others against a column layout. As the data is available in both layouts, we may route each query independently to the most suitable layout.

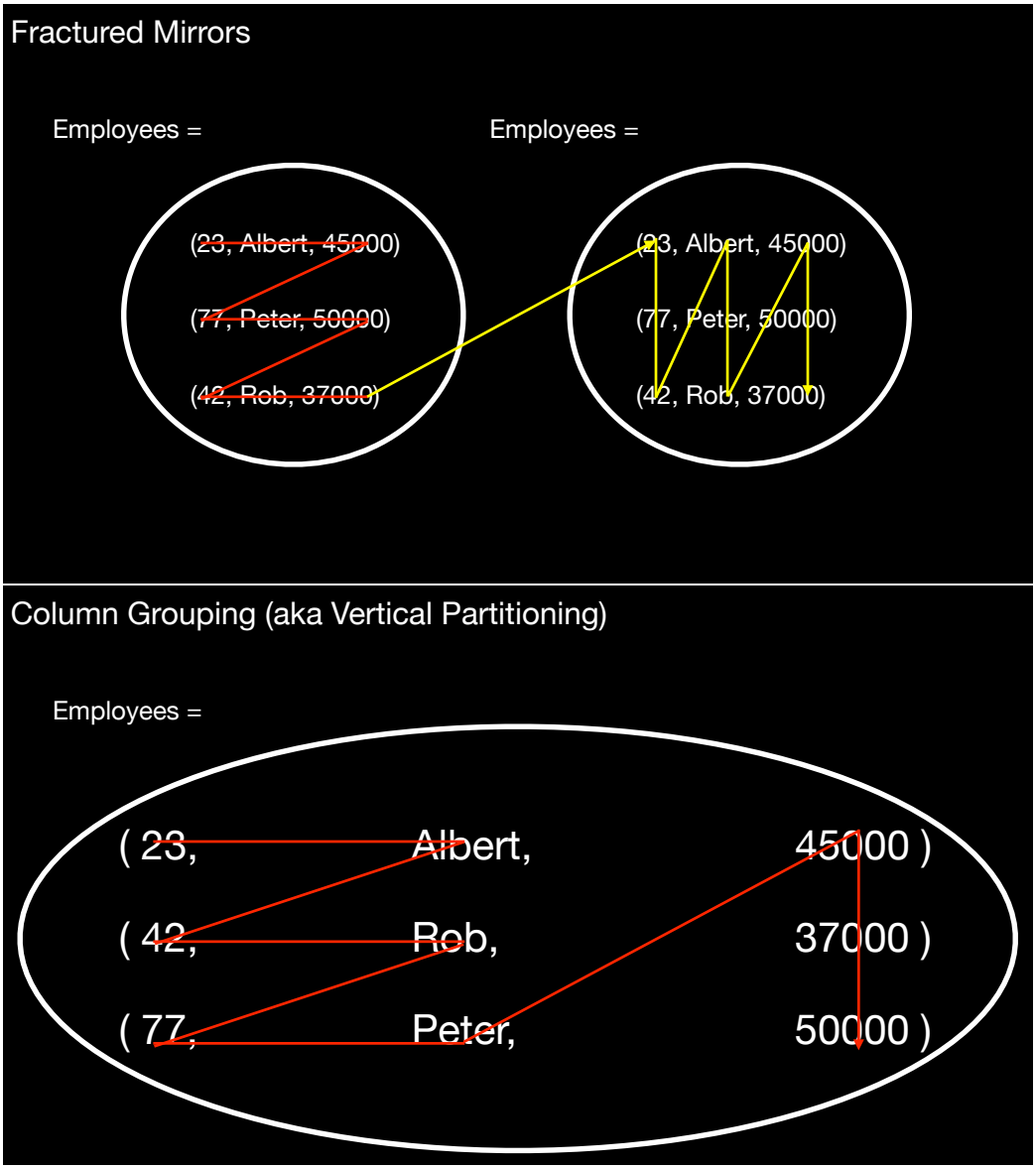


Figure 2.7: Fractured mirrors vs column grouping

What are the drawbacks of fractured mirrors?

As we keep data redundantly, we need more storage space for this layout. In addition, we have to make sure that updates are reflected in both stores, to be more precise: even under updates all queries must return consistent results. For instance, if a query Q1 modifies the row layout only, but not the column layout, it may sometimes still be correct to compute the next query Q2 against the outdated column layout. But, only if Q2 is not affected by the previous change, i.e. whether it is computed against row layout or column layout, it returns the same results. Notice also that there is a strong relationship of fractured mirrors to differential files, see Section 1.3.8. For instance, the read-only database may be organized in column layout whereas the differential file may be organized in row layout. Other examples include SAP HANA which as of writing these lines keeps its data in both layouts in main memory.

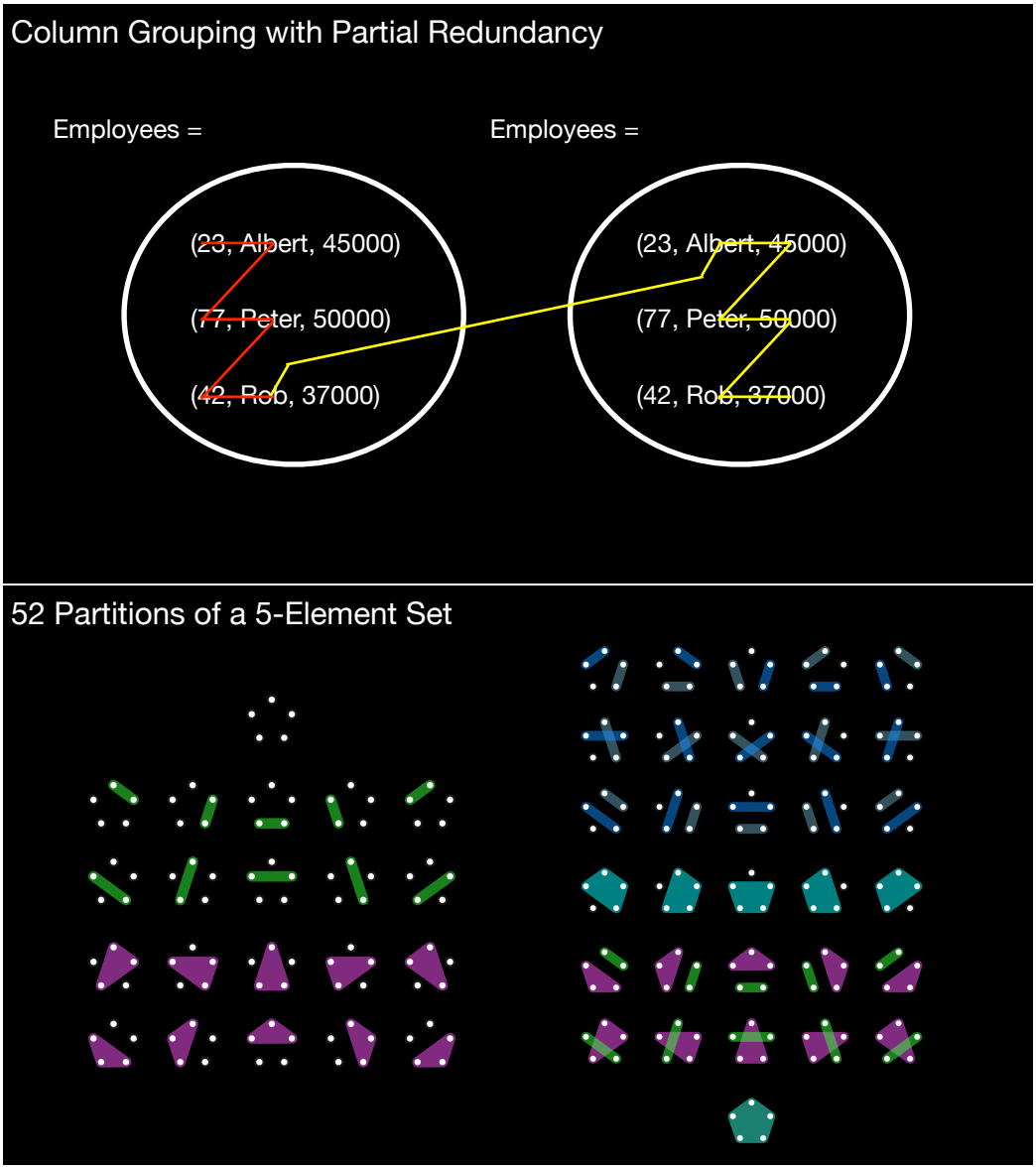


Figure 2.8: Column grouping with partial redundancy and the 52 possible partitionings of a 5-element set

What is column grouping and its relationship to vertical partitioning?

column grouping

vertical partitioning

Given a source relation, we can partition it vertically such that each vertical partition contains a subset of that source relation. We can identify two extremes in vertical partitioning: the one extreme is a column layout (aka a full vertical partitioning). A column layout is conceptually similar to a vertical partitioning of the source relation where each partition contains one attribute of the original table. How that vertical partitioning is implemented is another story, see Section 2.3.2. Also notice that simply partitioning a table in SQL into vertical partitions does not have the same effect as implementing this vertical partiitiong natively in the database store and enriching the query optimizer to be aware of the column layout.

The other extreme of vertical partitioning is a row layout, i.e. there is only a single "vertical partition" containing all attributes of the source relation. In-between these two

extremes there is room for a layout where the vertical partitions contain more than one attribute from the source relation. These layouts are called column grouping.

Again, what are important special cases of vertical partitioning?

Assume a table foo with attributes $A_1, \ldots, A_x$. Assuming that the vertical partitioning is complete and disjoint, we can identify the following special cases of vertical partitioning:

Number of disjoint vertical partitions	**Layout name**
x	column store
$1 < i < x$	column grouping
1	row store

However, keep in mind: by creating x different vertical partitions of a table in a row store, still performance-wise this is typically far off from the performance of a native column store. The reason is that column stores typically do not only change the data layout, but also perform considerable changes in terms of how queries are processed, e.g. tuple reconstruction joins. We will get back to this in Section 5.3.5.

Which type of queries would benefit from column grouping?

Queries touching a subset of the attributes of the source relation where that subset contains more than one attribute may benefit from such a layout.

data redundancy

How would we introduce data redundancy to column grouping?

We introduce data redundancy by representing some attributes of the source relation in multiple vertical partitions redundantly.

partitionings

How many vertical partitionings are there?

Bell number

Given a source relation with n attributes, the number of vertical partitionings is given by the Bell number B_n. For $n \leq 1$ B_0=B_1=1 and for $n > 1$ the Bell numbers satisfy the following recurrence relation:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \cdot B_k.$$

Starting with B_0, the first few Bell numbers are 1, 1, 2, 5, 15, 52, 203, 877, 4140. For instance, a source relation with $n = 5$ attributes can be vertically partitioned in $B_5 = 52$ different ways.

Q&As

1. In case of fractured mirrors the throughput is potentially improved by factors over row- as well as column layout for
   (a) inserts
   (b) updates
   (c) deletes
   (d) selects

2. In case of fractured mirrors the throughput of read-only queries

   (a) is doubled, since we can direct the incoming queries to alternating copies.

   (b) gets worse, since we have to read both copies of the data.

   (c) can be improved by directing the queries to the copy stored in the layout more suitable for the query.

   (d) is the same as for row layout.

3. In case of fractured mirrors

   (a) we keep the first half of the table in row layout, and the second half in column layout.

   (b) we keep two full copies of the tables in any layout.

   (c) we keep two full copies of the tables, one in row layout and one in column layout.

   (d) the copy of the table in column layout can only be stored in column layout with correlated columns.

4. A non-redundant vertical partitioning is

   (a) a complete and disjunct partitioning of the set of tuples of a table.

   (b) a complete and disjunct partitioning of the set of attributes of a table.

   (c) a complete partitioning of the set of attributes of the normalized table.

   (d) a complete partitioning of the set of distinct tuples of a table.

5. A table with X attributes can be partitioned into subsets along those attributes. Each of those vertical partitions consists of

   (a) attributes stored in column layout.

   (b) attributes stored in row layout.

   (c) attributes stored in any kind of layout.

   (d) attributes stored in the layout most suitable for the query workload.

6. Vertical partitioning with partial redundancy

   (a) is basically the same as fractured mirrors.

   (b) is vertical partitioning plus storing the heavily accessed tuples multiple times.

   (c) is vertical partitioning that is not disjunct.

   (d) can improve over non-redundant vertical partitioning in case of read-only workloads.

7. The complexity of the vertical partitioning problem

   (a) is polynomial in the number of attributes.

   (b) is linear in the number of attributes.

   (c) can be expressed by the function calculating the Bell-numbers.

Exercise

In order to choose a suitable data layout of a table, we usually need a representative set of queries describing the workload of the given table. Let's consider a variant of the usage matrix of the TPC-H Orders table from the industry-standard TPC-H benchmark[1] as displayed in Figure 2.9. In this matrix each row lists the attributes referenced by a given query (marked with 'X').

Let's assume that these queries do nothing else but a full scan on the Orders table stored on disk projecting the marked attributes. In case of any vertically partitioned layout (including a column layout), each query has to read any partition that contains an attribute referenced by the query. When doing this, it performs a single seek and a full scan for each referenced partition. Thus the execution time of a query is the sum of the time needed to read all partitions it references. The workload execution time is the sum of the execution times of each query in the workload. Caches are completely cold, i.e. the data is not available in main memory and all data required by a query must be fully read from disk for each query.

Queries \ Attributes	ORDERKEY 4 bytes	CUSTKEY 4 bytes	ORDERSTATUS 4 bytes	TOTALPRICE 4 bytes	ORDERDATE 8 bytes	ORDERPRIORITY 4 bytes	CLERK 4 bytes	SHIPPRIORITY 4 bytes	COMMENT 200 bytes
Q3	x	x			x			x	
Q4	x				x	x			
Q5	x	x			x	x			
Q7	x	x							
Q8	x	x			x				
Q9	x				x				
Q10	x	x			x				
Q12	x					x			
Q13	x	x	x						x
Q18	x	x		x	x				
Q21	x		x						
Q22		x							

Figure 2.9: The usage matrix of the TPC-H Orders table

The sizes of each attribute are listed on Figure 2.9 under their names. The TPC-H scale factor of our database is 10, which means the Orders table contains 15,000,000 rows. The server has a hard disk with an average seek time of 3 ms and a read bandwidth of 130 MB/s. The disk-, operating system-, and DB block sizes are all 8 KB. Main memory is limited so that we have a read buffer that can store 10 MB (note: kind of small, right?). Notice that data loaded into the buffer eventually needs to be reconstructed into a temporary internal tuple containing all attributes required by this query at a time.

We have a copy of the table in row-layout, column-layout, and fractured mirrors. For simplitcity assume implicit keys for column-layout and other vertical-partitioned layouts.

(a) When executing Q18 on the Orders table in each layout, what is the minimum amount of data you have to load into main memory (and very likely into the caches), respectively?

(b) How long does it take in terms of disk I/O to execute Q18 for each of the layouts?

[1] http://www.tpc.org/tpch/

(c) How long does it take in terms of disk I/O to transform the whole table from row layout to any of the other layouts? The disk write bandwidth is 90% of the read bandwidth.

(d) Your task is to suggest a “suitable" vertical partitioning (without replication) by intuition, and using the cost model described above compare it’s estimated workload execution time with that of row- and column-layout.

2.3.4 PAX, How to choose the optimal layout?

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[ADHS01]

Further Reading:	
W Apache Parquet	

Learning Goals and Content Summary

What is PAX about?

PAX

PAX is a hybrid layout that first divides the source relation horizontally into horizontal partitions.

More formally (compare our discussion in Section 2.3): We define a linearization L_{PAX} with a lexicographical ordering:

$$L_{\text{PAX}}(i,j) \mapsto L_{y1}(j) \bigoplus L_x(i) \bigoplus L_{y2}(j) \tag{2.6}$$

This means, each index (i, j) is mapped to a z_k-value where the first part of z_k (the prefix) is determined by a linearization function $L_{y1}(j)$ defining the order among **horizontal partitions**. In other words, this is the horizontal partitioning function. The second part of z_k is determined by a linearization function $L_x(i)$ defining the order among **columns** *within one particular* ***horizontal partition***.

The third part of z_k (the suffix) is determined by a linearization function $L_{y2}(j)$ defining the order among **rows** *within columns that are within one particular* ***horizontal partition***.

How does PAX relate to horizontal partitioning?

horizontal partitioning

PAX can be considered a hierarchical partitioning: at the top-level (the root node) we

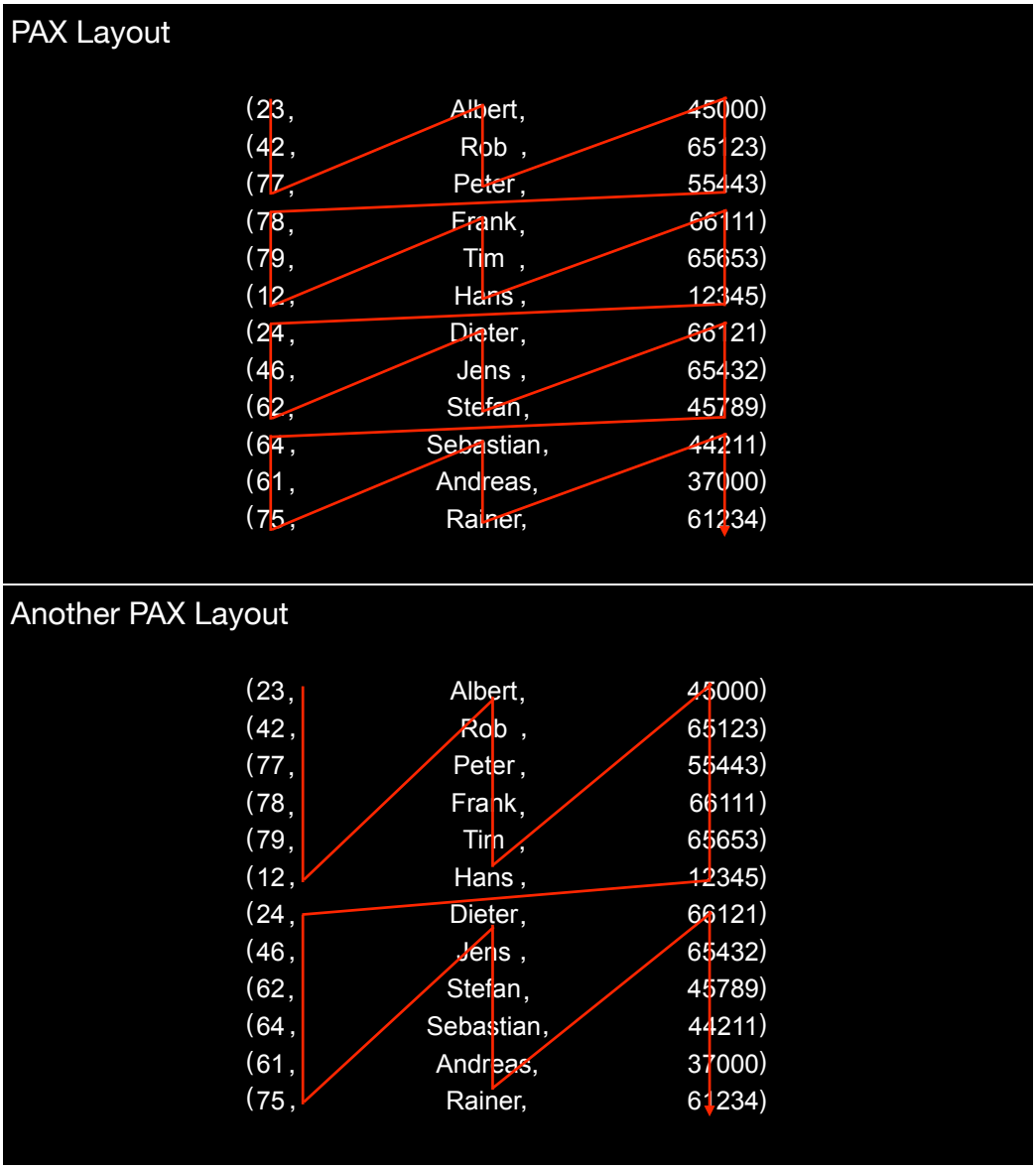


Figure 2.10: Two different variants of PAX layout: three rows per block vs six rows per block

apply a horizontal partitioning, underneath (the leaf-level) we apply vertical partitioning. Obviously, we may introduce other partitioning levels in this hierarchy. For instance: we could add another level under the leaf-level partitioning each column of a horizontal partition again horizontally (horizontal → vertical → horizontal). This may be useful to allow compression algorithms to read only parts of the column, see also Section 2.4. And, yes, this is yet another example of fractal design, see Section 2.3.5.

block

How does the horizontal partitioning relate to blocks?

The sizes of the horizontal partitions may be chosen according to your system, workloads and needs. For instance, it is natural to choose the horizontal partitions to correspond to a relatively small database page of 64KB, like originally proposed in [ADHS01]. This would allow us to keep the same data on the page, we would just layout that data differently *within a page*. In this particular case, the overall system would behave unchanged

w.r.t. page I/Os, however w.r.t. reads we may observe some performance improvements as queries may benefit from the column layout inside a page. However, page sizes may also be relatively large, like for instance in HDFS, where page sizes are typically bigger than 64 MB. This is the core idea of modern HDFS-page layouts like Apache Parquet which is basically a refinement of PAX.

How does PAX relate to column layouts? **column layout**

Again, a column layout is used within each horizontal partition independently. Assume we have a table T with N tuples and at least two attributes. Now, we can differentiate the following cases:

Number of horizontal partitions	**Layout name**	**tuples per horizontal partition**
1	column layout	N
$1 < i < N$	PAX	$\lceil N/i \rceil$
N	row layout	1

In other words, the higher the number of tuples per partition, PAX becomes close to a row layout. Vice versa, the lower the number of tuples per partition, PAX becomes close to a column layout.

What are the advantages of PAX?

Overall, PAX trades read performance w.r.t. columns with update performance w.r.t. tuples. Recall what happens if we want to insert a tuple having ten attributes into a pure column layout: we will have to touch ten different, possibly far apart, storage locations: one for each column. In contrast, in a pure row layout, we will only have to touch one storage location. PAX sits in-between those two extremes: we still have to touch ten different storage locations, however, the maximum distance among those storage locations is limited by the size of the horizontal partition.

How do we get the optimal data layout anyway? **optimal data layout**

Well, first make sure you understand what "optimal" means to you, see also Section 3.3. So for you, is "optimality" defined w.r.t. some runtime model, i.e. counting CPU operations? Or are you counting page accesses and/or cache misses? Or is it wall-clock time you have in mind? Second, the right physical database design for your needs depends on your workload, i.e. the types of queries you want to execute. Third, the database schema and the data distributions may also, obviously, impact performance. If all that information is available to you, you may compute the optimal layout using a suitable optimization technique. And then keep your fingers crossed that neither query workload nor data distributions nor schema change as that may require a different optimal layout. So, in summary: it is difficult to come up with an optimal layout. You should rather use a layout that is good enough and also robust in case workload, data distributions and/or schema change. For most workloads, PAX is not optimal, but: it is relatively robust and performs very well on many workloads.

Q&As

1. The PAX layout

(a) is actually the same as fractured mirrors

(b) is a hybrid of row layout and column layout

(c) simulates a column layout for each table of a row store

(d) is an imitation of row layout inside a column store

2. The properties of the linearisation order of the PAX layout are:

(a) we linearise as in column layout inside a chunk of tuples

(b) we take a chunk of tuples with as many tuples as attributes in the table

(c) we linearise as in column layout for the whole layout

(d) we linearise as in row layout inside a chunk of tuples

3. In the PAX layout

(a) horizontal partitions typically match the page size

(b) horizontal partitioning is applied inside the page

(c) inside the page the tuples are laid out exactly as in slotted pages

(d) we use a column layout inside the pages

4. A workload could be

(a) a set or sequence of queries executed against a database

(b) the I/O statistics of a DBMS collected over a given amount of time

(c) the utilisation levels of the various storage devices storing the database

(d) the layout of the data

5. The optimal layout of the data depends on

(a) the workload

(b) the storage media used

(c) the DBMS

(d) the performance (throughput, latency, etc.) preferences of the customer

2.3.5 The Fractal Design Pattern

Material

Video:	Original Slides:	Inverted Slides:

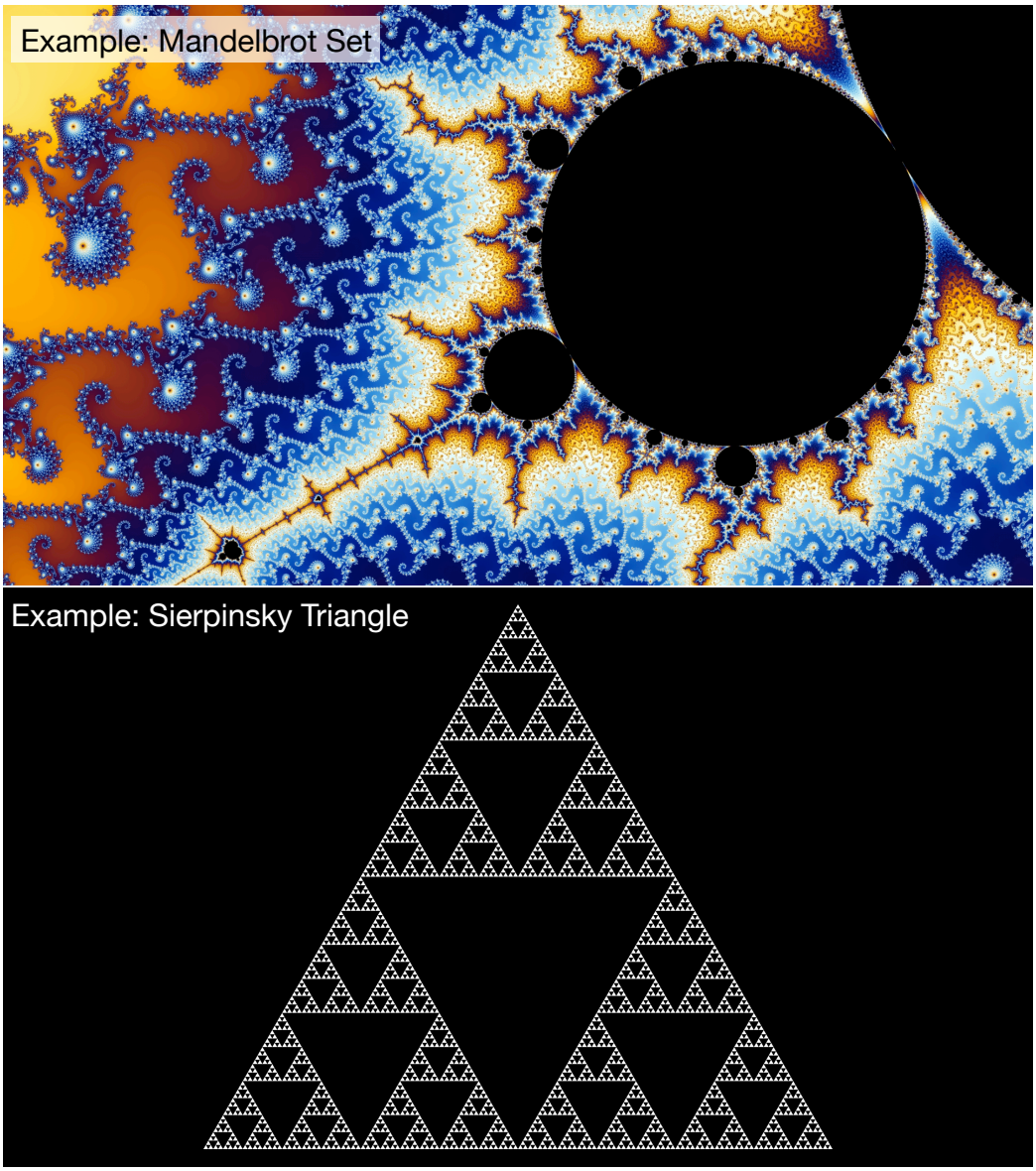


Figure 2.11: Example fractals: the Mandelbrot Set and the Sierpinsky Triangle

Learning Goals and Content Summary

What is fractal design (or self-similar design) in the context of database systems?

fractal design

self-similar design

Fractal or self-similar design occurs when a method X operating on granule Y can be adapted to work on a different granule Z as well.

A "granule" may be a layer of the storage hierarchy (e.g. L1, DRAM, disk), a computer system of any size (single node, cluster, datacenter), or a storage unit (cache line, page, chunk).

For instance, RAID, see Section 1.2.6, was originally proposed to operate on "inexpensive disks". However, disks are not the only granule where RAID may be applied. Other granules where RAID are applied include, RAID-systems (for nested RAID), SSDs, flash storage chips (for the SSD-internal RAID 5), data centers, and cloud storage providers. See Figure 2.12.

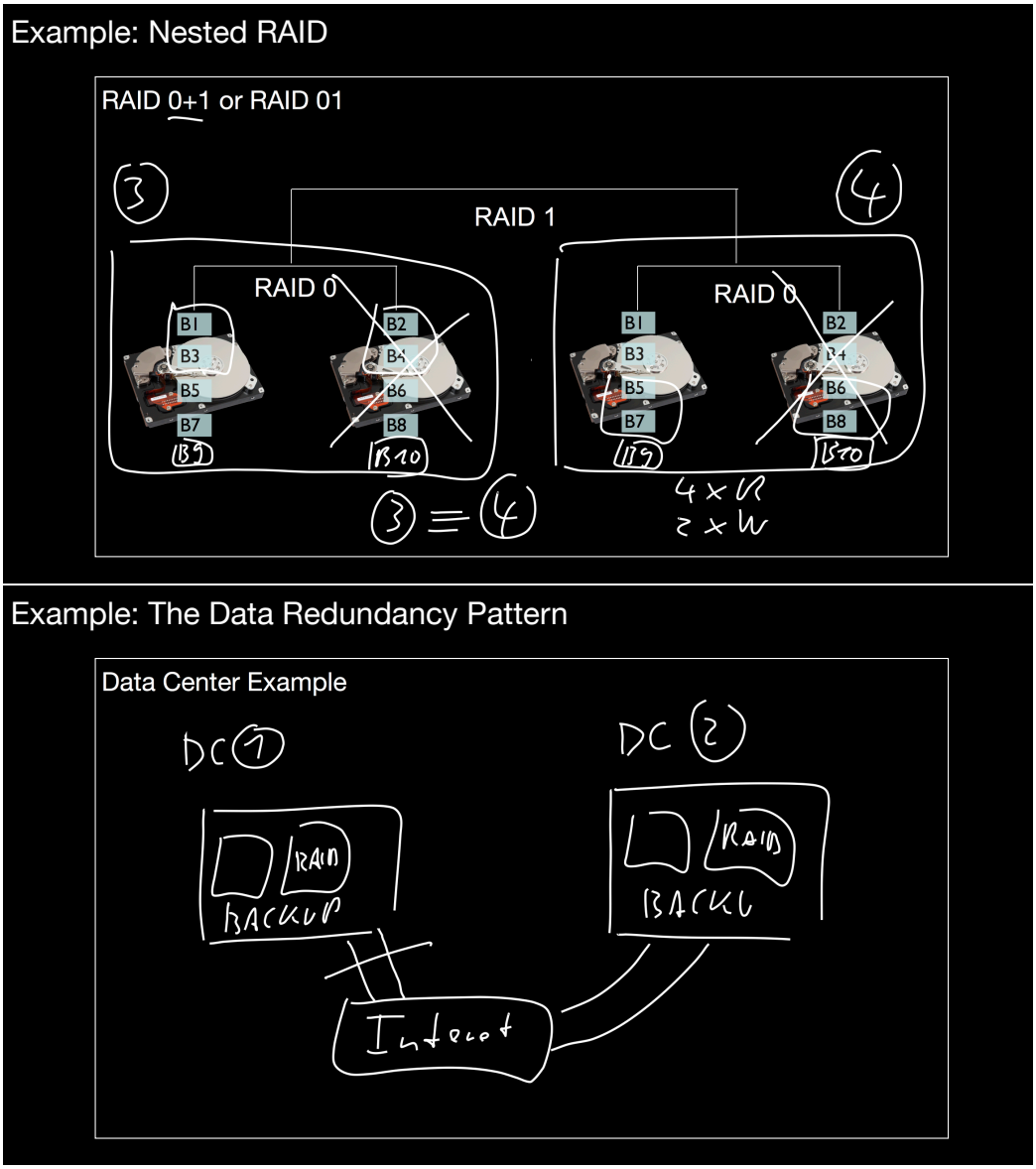


Figure 2.12: Examples for self-similar design in databases: nested Raid and data redundancy

Why does fractal design help us to devise effective algorithms?

If you already know a solution works on granule Y, maybe it can be applied easily to work on a different granule Z. This may be more effective than reinventing everything from scratch. So the first thing when designing a new algorithm is to understand whether an already existing method X may be adapted to work on granule Y.

How does the fractal design pattern relate to The All Levels are Equal Pattern?

The All Level are Equal Pattern, see Section 1.1.1, is just a special case of fractal design where granules are restricted to the different layers of the storage hierarchy. In fractal design, we drop that restriction. Recall again, that both patterns are guidelines rather than strict rules.

Can you name a couple of examples of fractal design in databases that we have already seen?

We have seen this already in many different places.

1. linear addressing is used to address pages inside a segment (inter-page addressing), but also to address chunks (rows or columns) inside a page (intra-page addressing), see Section 2.2.2. In addition, the organization and virtualization of storage using prefix-trees, see Section 1.4.1, is self-similar for different granules when changing the length of a prefix.

2. RAID, see Section 1.2.6, is used to combine multiple hard disks or SSDs into a virtual disk that is more reliable and depending on the RAID-level also faster. Those virtual disks may be combined with other virtual disks applying RAID again. Like that we have a two-level (nested) RAID, see Section 1.2.7. This is self-similar design as the idea of RAID is applied on two different levels. If the physical SSDs used in that RAID-system uses RAID 5 internally for error correction, see Section 1.2.9, this adds a third layer of self-similarity. And finally, if we assume that the entire RAID storage system is replicated across data centers this adds a fourth layer of self-similarity.

3. PAX is another example of self-similar layouts, see the discussion in Section 2.3.4.

Q&As

1. What patterns are special cases of the Fractal Design Pattern?

   (a) All Levels Are Equal Pattern

   (b) Batch Pattern

   (c) No Bit Left Behind Pattern

2. Which of the following might be examples of the Fractal Design Pattern?

   (a) Nested RAID

   (b) Datacenter replication

   (c) Tape

3. How many levels of self similarity can you find in a RAID-55 system with SSDs?

   (a) _____

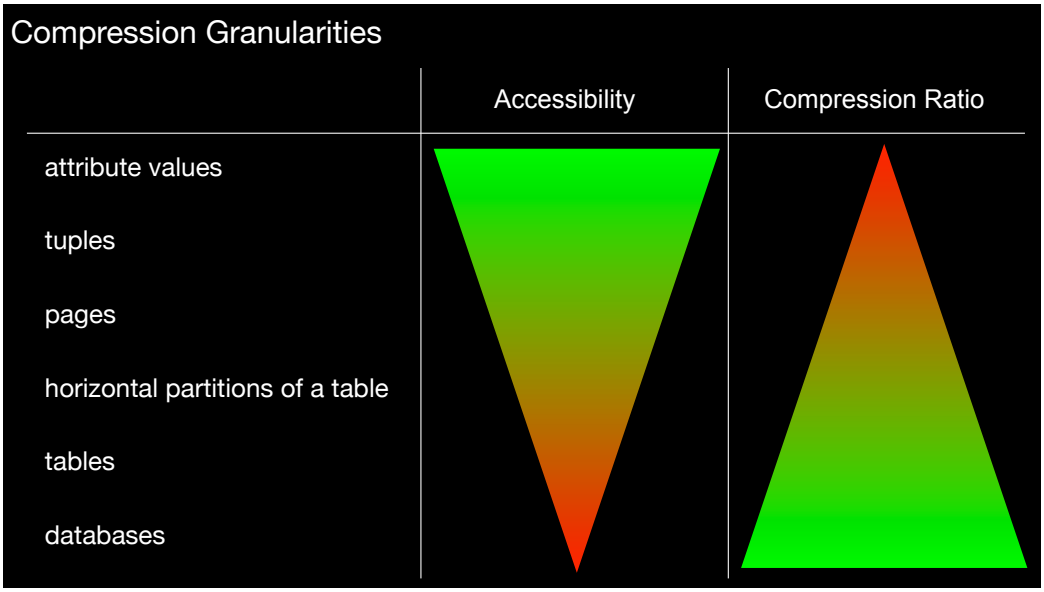


Figure 2.13: Compression granularities and their trade-offs w.r.t. accessibility of data and compression ratio

2.4 Compression

2.4.1 Benefits of Compression in a Database, Lightweight Compression, Compression Granularities

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	Further Reading:
[WKHM00]	[HRSD07]

Learning Goals and Content Summary

storage space

Compression is mainly about saving storage space, right?

Not only. In databases the more important effect is saving bandwidth in all kinds of situations when data is transferred over a wire: from/to disk, over the network, over a bus like the memory bus.

Compressing data costs something in addition! You can only lose w.r.t. overall query response times, right?

That is simply not true. We have to keep in mind the total costs for

1. compressing the data,
2. transferring it, and then

3. decompressing it.

Those steps may actually overlap, e.g. the CPU time for step 1, compressing, may overlap with step 2, the actual transfer time. In addition, if step 1 only needs to be executed once, but step 2 is done several times, the costs for compressing may be amortized over several executions of step 2. Moreover, query processing may in several situations work directly on the compressed data. Thus, the costs for step 3 may actually become zero, see Section 2.4.2.

What is the major trade-off we have to keep in mind when compressing and decompressing data?

Steps 1, 2, and 3 should be less expensive than just executing step 2 on uncompressed data.

What are compression granularities?

compression

In general, a compression algorithm foo(X)↦Y is executed on an input item X and compresses that item to a byte-sequence Y. The input item X has a specific size which is coined the *granule of X*. Examples for X, of increasing granule, are: a single attribute value, a tuple, a data page, a horizontal partition of a table, an entire table or even an entire database.

How do the different compression granularities affect accessibility and compression ratio of your data (in general)?

accessibility

compression ratio

For most compression algorithms if we want to access a smaller data item Z that is contained in a compressed byte-sequence Y, we cannot simply retrieve it from Y. In order to access Z, we have to decompress Y from the beginning of the compressed byte-sequence Y until we find Z (or we even have to decompress Y entirely). This means, accessing Z inside Y triggers some overhead for decompressing other data we are actually not interested in. On the other hand, several compression algorithms work best if they are applied to a large input item X, e.g. if X is an entire table, we will likely gain a lot. In contrast, is X is only a single attribute value, we do not gain much.

To sum up: usually larger chunks of data allow for a better compression ratio, however, accessing data within a large compressed chunk may be expensive if only a small data item within that chunk needs to be retrieved.

Q&As

1. Consider the following inequality from the video: $time(decompression) + time(readCompressed) < time(readUncompressed)$. Let us assume the inequality does not hold, i.e. $time(decompression) + time(readCompressed) \geq time(readUncompressed)$, but it is still worth processing the compressed data. What could be a reason for that?

   (a) decompression can overlap with reading the compressed data, therefore we should only consider the time it takes to read the compressed data.

   (b) decompression can overlap with reading the compressed data, and decompression is also faster than the time spent reading the compressed data.

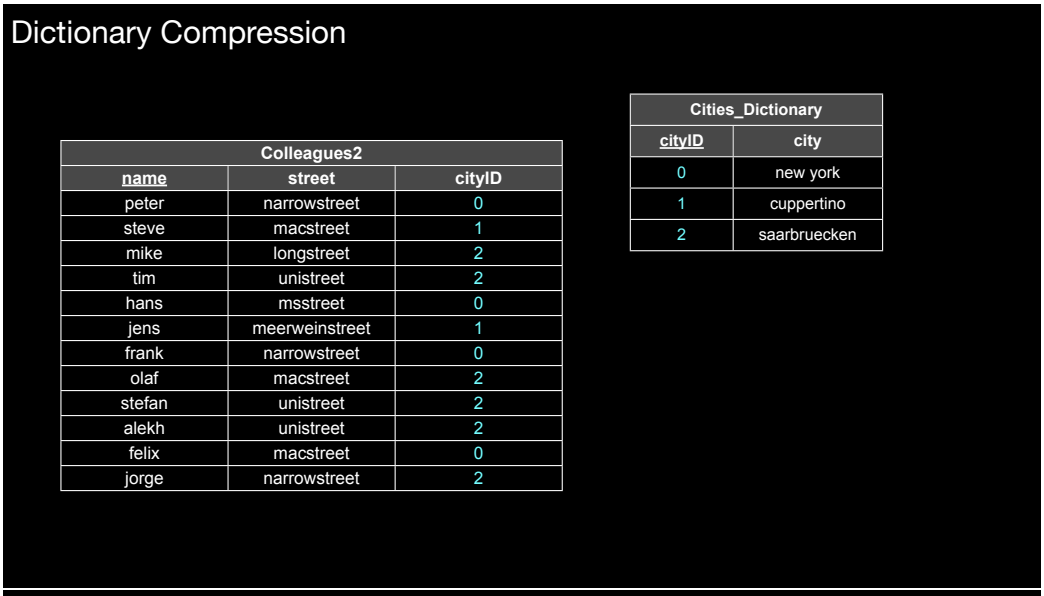
   (c) this simply cannot happen.

2.4.2 Dictionary Compression, Domain Encoding

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary



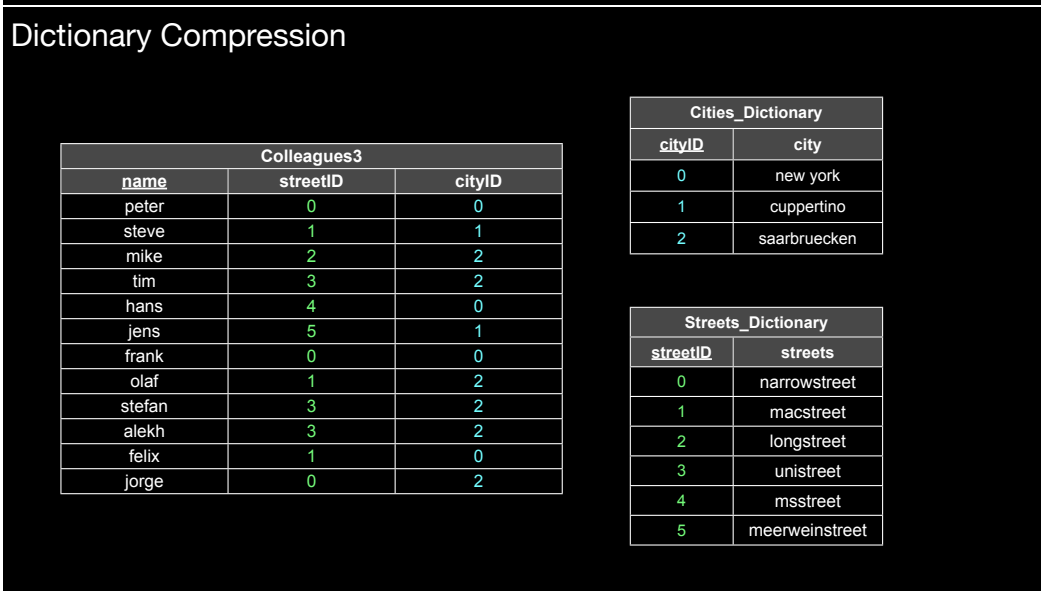


Figure 2.14: Dictionary compression: just column `cityID` vs both `streetID` and `cityID`

dictionary

What is a dictionary?

A foreign language dictionary translates terms from language X to terms in language Y. In the context of a database system a dictionary does something very similar: a database

dictionary translates terms from a domain X to terms of domain Y. Terms in domain X are typically short and memory-efficient keys, e.g. integers. In contrast, terms in Y are typically long, e.g. strings.

dictionary compression

How do we apply dictionary compression?

Assume a table `foo` with attributes `A1,...,An`. In order to dictionary compress column `Ai`, we split `foo` into two tables which are connected to a foreign key relationship as follows:

1. compute the set of distinct values of `Ai`,
2. assign an artificial key `ID` to each distinct value found,
3. create a new table `Ai_Dictionary` with schema [`Ai_Dictionary`, `term`], i.e. `Ai_Dictionary` is the key,
4. change the schema of attribute `Ai` in table `foo` to be a foreign key to column `ID` in table `Ai_Dictionary`, replace values in `foo.Ai` by corresponding `Ai_Dictionary.ID`, rename `foo` to `foo_New`,
5. create a (dynamic) view `foo_view` with a natural join on `foo_New` and `Ai_Dictionary` showing all attributes except `Ai_Dictionary.ID`; in other words, this view returns the same result set as table `foo`,
6. replace all occurrences (in all SQL statements and views) of table `foo` by the view `foo_view`.

Notice that all of these steps can easily be done (and should be done) in SQL.

What do we gain by using dictionary compression?

If the sum of the sizes of tables `foo_New` and `Ai_Dictionary.ID` is smaller than the size of table `foo`, we gain space. Whether we gain space depends on the data distribution of values in `foo.Ai`, in particular the number of repetitions.

In addition to gaining space, we may also speed-up certain types of queries which may operate on dictionary-compressed data without decompressing the data.

What do we loose by using a dictionary compression?

For each dictionary-compressed column, we introduce an additional join. So, in the worst case, if dictionary compression is applied to all attributes `A1,...,An` of table `foo` and we have a query referencing all attributes that query will have to use n additional joins. These join may lead to considerable costs in query processing.

Actually, decompressing a dictionary during query processing can be considered a variant of an anti-projection or tuple reconstruction (see Section 5.3.5). In dictionary decompression, tuple reconstruction is done using attribute values rather than rowIDs.

CREATE DOMAIN

How does dictionary compression relate to CREATE DOMAIN in SQL?

If you create a domain in SQL, you often explicitly list the set of allowed values in the domain's definition. Once, the domain is defined, you may use that domain to define attribute types in any table of your database. For data inserted into such a table, the database management system then has two options:

1. insert the actual value into that table, or
2. insert a dictionary key linking to a dictionary representing that domain.

However this is implemented by the database system, its implementation does not have to be exposed to the user. For instance, assume you define a domain that is allowed to contain three possible string values only like "foo", "bar", and "whatever". Now, for each row having an attribute typed with that domain the database system may store the actual string. Obviously this is not very efficient. A better solution is to map the three strings to a dictionary { 0↦ "foo", 1↦ "bar", 2↦ "whatever"} The dictionary key only requires 2 Bits. The drawback of this approach is that, depending on the query, we need to lookup the dictionary to "understand" the dictionary keys. In summary, creating a domain in SQL defines a type where the instances of that type may be represented using a dictionary. In contrast, creating a dictionary does not create a type.

Is a dictionary something a user has to be aware of?

Dictionaries should be hidden from the user. In SQL dictionaries should be hidden using (dynamic) views.

domain encoding

How does dictionary compression relate to domain encoding?

Once you applied dictionary compression to a particular column, it may pay off to also apply domain encoding. For instance, assume the original column `foo` was of type varchar(32). Now, we apply dictionary compression replacing each strings in that column by its corresponding key in the dictionary. Let's assume we use an int (4 Byte) type for this. So now, our column `foo` is of type int. This requires 4 Bytes for each entry. However, if our dictionary contains less entries than the 2^{32} different keys that can be represented by int, we may use a smaller type: If our dictionary has n entries, we need at most $\lceil \log_2(n) \rceil$ bits per entry. For instance, if $n = 42$, we require $\lceil \log_2(42) \rceil = 6$ bits.

Can we apply domain encoding at different levels?

Domain encoding can be done at two different levels: (1) by choosing the right type for your attributes when defining/changing the schema; or: (2) by implementing domain encoding internally, i.e. inside the DBMS. Notice that what the DBMS claims to be the type of a column to the outside does not necessarily have to be implemented like that by the DBMS internally.

Q&As

1. Dictionary compression
   (a) is used to bring tables into the third normal form
   (b) can be used to eliminate the redundancy of the attribute values
   (c) can be used to reduce the storage requirements of the attributes values
   (d) can only be applied for row stores
2. The key idea of dictionary compression is
   (a) to reduce the storage requirements by transforming the values into a domain whose values need less storage

(b) to reduce the storage requirements by denormalizing the table

(c) to speed-up processing of the compressed column by counting the distinct values in the column

3. When rewriting a point-query to be executed on a dictionary-compressed column, it is most natural to use

(a) a sub-query to look up the compression key of the value used in the selection predicate

(b) a sub-query to look up the value for the compressed key used in the selection predicate

(c) a join between the compressed table and the dictionary of the column

4. If we need 3 bits to store each key in a dictionary compressed column in main memory

(a) it is always best to use 3 bits to store each key

(b) it is best to use exactly as many bits for each key as the size of a processor word, e.g. 32 or 64 bits

(c) it can be beneficial to use the smallest directly accessible storage granule that can hold the keys, i.e. a single byte, to avoid the overhead caused by bit-shifting

(d) it can be beneficial to use the smallest amount of bits, in this case 3 bits, to store each key, to maximise throughput of full-scan queries

5. The relation of dictionaries and domains is the following:

(a) they are basically the same constructs

(b) a domain is only a set of values present in a given column, while the dictionary is a mapping between the keys and these values

(c) the dictionary is a mapping between the keys and the set of values present in a given column, while the domain defines the possible values of the column

(d) domains only exist in PostgreSQL, while dictionaries are available in the majority of DBMSs

2.4.3 Run Length Encoding (RLE)

Material

Video:	Original Slides:	Inverted Slides:

Figure 2.15: Run length encoding: on `streetID` vs `streetID` and `cityID`

Learning Goals and Content Summary

run length encoding

How does run length encoding (RLE) work?

RLE

Run length encoding takes as its input a sequence of values. Whenever that sequence contains a subsequence of at least two equal values, those repetitions are represented as a pair $(v, count)$ where v is the data value and *count* the number of repetitions. Like that all repetitions in that sequence may be represented by a single pair rather than repeating each value v *count* times.

sorting

What is the relationship of RLE to sorting?

Obviously, RLE works best if the data was sorted as this increases the likelihood of subsequences having repetitions.

lexicographical sorting

What is the relationship of RLE to lexicographical sorting?

In a database it may pay off to recursively sort, and hence implicitly group the data, before applying RLE. For instance, for a table `Colleagues` with attributes `streetID` and `cityID`, we may sort these columns lexicographically. This means, we first sort on `streetID`. In that process, for all values in `streetID` that are equal, we sort their `cityID` values. Like that the rows are ordered w.r.t. both `streetID` and `cityID`. Thus, we may achieve a higher compression ratio as when sorting only on a single attribute. However, whether this strategy pays off and how many columns should be sorted lexicographically for a particular table depends on the data distribution, in particular the number of different values in these columns.

What are possible pros and cons of RLE?

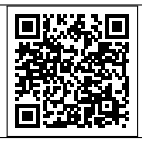
RLE may not pay off in cases where the column does not contain duplicates, e.g. if the column is a candidate key. In general, whether RLE pays off depends heavily on the cardinality of the column, i.e. the number of different values in that column and its relationship to the total number of values in that column.

Q&As

1. Run length encoding only makes sense if the column to be compressed
   (a) is sorted (or same values are clustered)
   (b) is dictionary encoded
   (c) stores numbers
   (d) stores strings
2. Which of the following operations suffer in performance from applying run length encoding to a column:
   (a) aggregation and grouping both on the RLEncoded column
   (b) a selection after grouping on the RLEncoded column (i.e. a HAVING clause)
   (c) point lookups
   (d) projecting the RLEncoded column

2.4.4 7Bit Encoding

Material

Video:	Original Slides:	Inverted Slides:
		

Additional Material

Further Reading:	
W UTF-8	

Learning Goals and Content Summary

What is the core idea of 7Bit Encoding? **7Bit Encoding**

The core idea of 7Bit encoding is to adapt the number of bytes used for storing an individual value. In 7Bit encoding we logically split each byte into two parts: the first bit is considered a signal bit, the remaining seven bits represent the actual data. A data value is represented by a sequence of $k \geq 1$ bytes where the signal bit of the last byte is unset, all other signal bits are set. Like that, conceptually a data value is represented by a list of bytes. If the input data value has n bits, we require $\lceil n/7 \rceil$ bytes. This idea is also used when representing characters in UTF-8.

How does 7Bit encoding relate to domain encoding? **domain encoding**

In 7Bit encoding the adaptation happens for each individual data value rather than an entire domain, i.e. an entire database column. Usually, 7Bit encoding should be the last compression method to try. If the data values to compress are of varying length, it may

Figure 2.16: 7Bit encoding: up to two vs three bytes

be more beneficial to use dictionary compression followed by domain encoding (assuming there are duplicate values).

storage space

How much storage space is lost or gained when using 7Bit encoding?

In terms of net storage, for every eight bits we lose one signal bit (metadata). In other words, one out of eight bits is unused. In general, if before compression you used a fixed-size domain, say of eight bytes for every data value, in 7Bit encoding those 8 bytes can only represent $8 \cdot 7 = 56$ bits rather than $8 \cdot 8 = 64$ bits. So every uncompressed value requiring more than 56 bits will need more than 8 bytes for storage in 7Bit encoding. In summary, whether 7Bit encoding pays off depends heavily on the data distribution.

Q&As

1. When should you use 7-bit encoding?
   (a) 7-bit encoding is always beneficial.
   (b) When there is a lot of variance in the sizes of the binary representation of the values.
2. What are drawbacks of 7-bit encoding?
   (a) Different values inside the same column can have different sizes.
   (b) Not all bits are used to store data.
   (c) Every value needs at least one byte.

3. How can you access an arbitrary position in a column with 7-bit encoded values?
   (a) Simply calculate the offset of the value.
   (b) Scan through all values.
   (c) Jump to the nearest known offset and scan from there.
4. How many bytes are needed to represent a 32 bit integer using 7-bit encoding?
   (a) _____

Exercise

Assume the following database tables as shown in Figure 2.17.

(a) Assume the tables are stored in column layout. Which compression techniques presented in the lecture would you choose for the following tables? Find at least one compression technique as well as the sort order for each table that reduces the amount of data considerably. You are allowed to combine compression techniques. Determine the compression ratio,
i.e. “compressed data size"/“uncompressed data size".

(b) Assume the tables are stored in PAX layout. What is the impact on the compression techniques chosen in (a)? Hint: factor in the size of a PAX-block in your argumentation. Keep in mind that in reality both tables and PAX block sizes are much larger.

(**Note:** do not think about possible future data for this table, just try to compress the given data as well as possible.)

visitors		
ID	Downloads	IP_address
13	217	138.92.122.175
81	0	138.92.122.195
42	6	138.92.122.182
25	4	138.92.122.181
21	52	138.92.122.177
56	4	138.92.122.188
78	2	138.92.122.191
30	1	138.92.122.185
23	0	138.92.122.179
80	2	138.92.122.193
27	3	138.92.122.183
82	0	138.92.122.197

issues		
ID	Status	Subject
1	In Progress	Migrate Moodle site to Turnkey VM
2	In Progress	Come up with backup strategy
5	New	Test recovery of Moodle site
6	Resolved	Drink fifth coffee
7	Resolved	Buy next coffee
8	Resolved	Test group selection for students
9	Resolved	Set up Moodle site

users		
ID	Name	Origin
12	Frank	Boring (Oregon, USA)
2	Robert	Boring (Oregon, USA)
50	Florian	Boring (Oregon, USA)
32	Dominik	Boring (Oregon, USA)
33	Viktor	ii (Finland)
10	Christian	ii (Finland)
14	Stefan	ii (Finland)
1	Martin	ii (Finland)
9	Jan	ii (Finland)
21	Josef	ii (Finland)
15	Achim	ii (Finland)
16	Salim	Truth Or Consequences (New Mexico, USA)
34	Dominik	Truth Or Consequences (New Mexico, USA)
30	Hassan	Truth Or Consequences (New Mexico, USA)
42	Kamran	Truth Or Consequences (New Mexico, USA)

Figure 2.17: The tables to compress

Chapter 3

Indexes

3.1 Motivation for Index Structures, Selectivities, Scan vs. Index Access

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	Further Reading:
[LÖ09], Tree-based Indexing	[LÖ09], Index Tuning

Learning Goals and Content Summary

What are the major analogies of indexing in real life?

indexing

We can observe several examples of indexing in real life: street signs show you the way and thus allow you to cut down the search space. The same holds for room plans inside buildings which allow you to go straight to the right room inside the building rather than searching all rooms. Similarly, a (printed) phone book is ordered alphabetically by name. This allows you to find entries using binary search.

Why do we use indexes?

The core idea of indexing is to prune the search space. Rather than inspecting all of the entries in a database, you only inspect a subset of the entries. That subset may still be a superset of the entries you are actually looking for (this type of index is then called a filter index). In that case the elements returned by the index still have to be post-filtered. Alternatively, an index may return a precise result which does not have to be post-filtered anymore. Getting back to our street sign analogy: indexes often use multiple street signs.

filter index

post-filter

Figure 3.1: Indexes work just like street signs.

For instance, in a binary search tree, each internal node can be considered a street sign allowing you to make a decision to either continue your search on the left or the right subtree. Like that at every street sign (or node) you cut down the search space by a factor two (assuming a perfectly balanced tree).

selectivity

What is selectivity?

Selectivity is a measure quantifying the fraction of elements returned by a function or method. Selectivity is always defined as a *ratio* of the elements returned over the total number of elements, i.e. selectivity is the ratio $0 \leq \frac{|\text{result set}|}{|\text{set}|} \leq 1$ where: result set $\subseteq$ set. For instance for a table R and a selection $\sigma_{a=42}(R)$, the selectivity is $\frac{|\sigma_{a=42}|}{|R|}$, in other

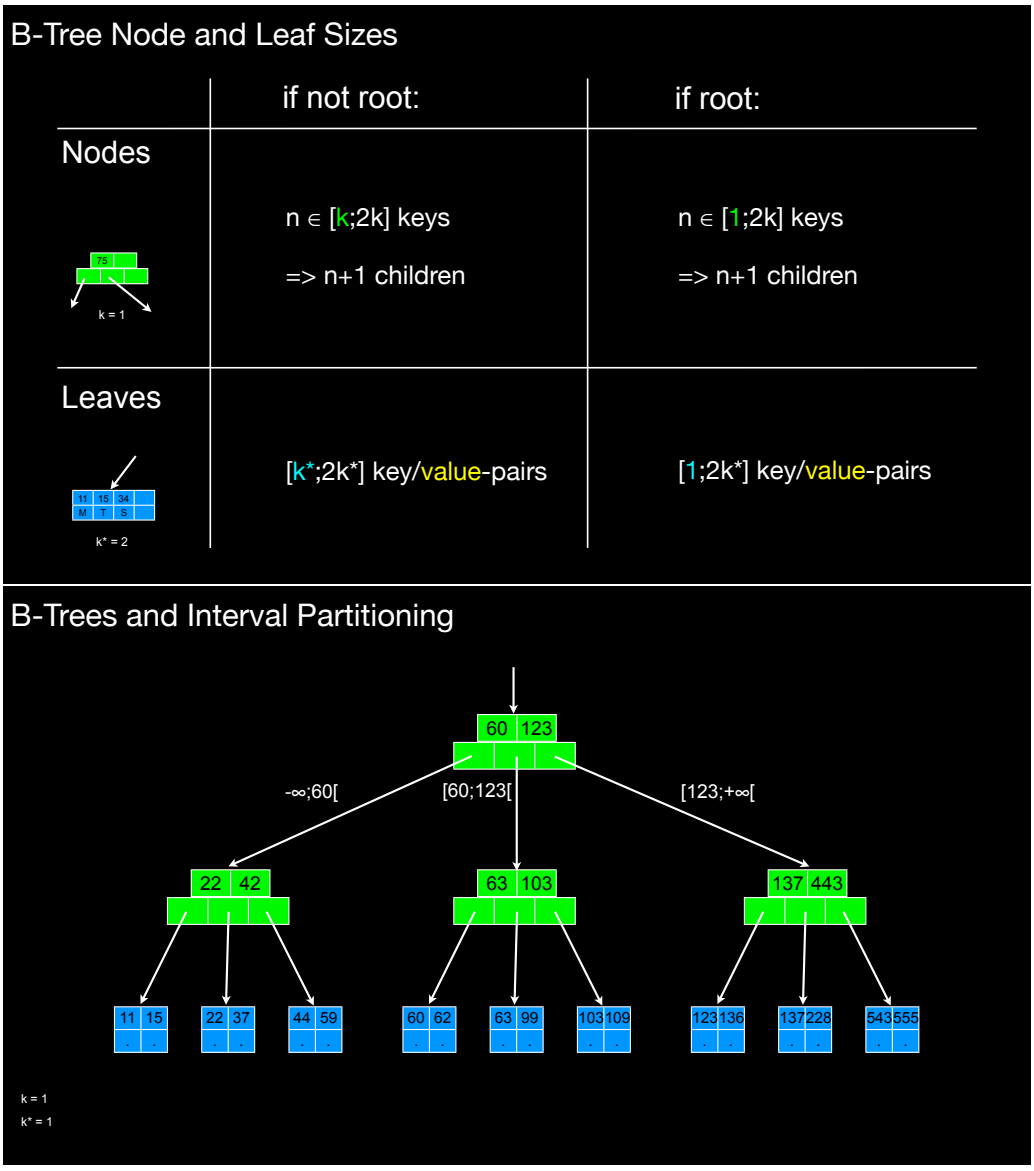


Figure 3.2: B-tree properties and their recursive interval partitioning

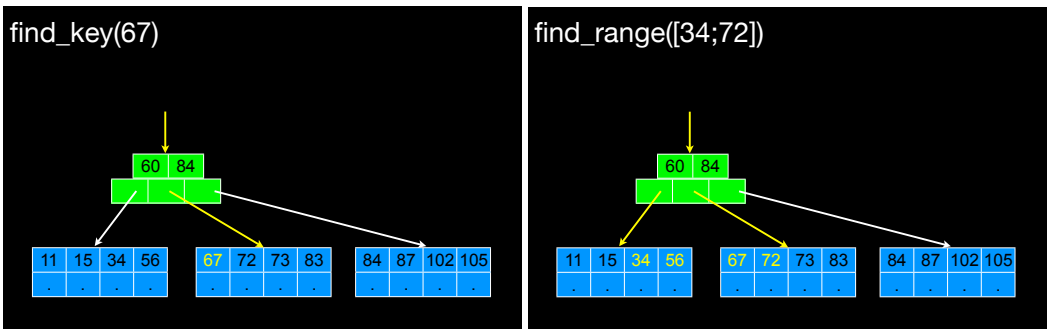


Figure 3.3: Searching keys and intervals in a B-tree

mapping BSTs to disk pages is rather difficult. If you map each node to a separate disk page, a search down that tree will trigger one disk seek in the worst case. A

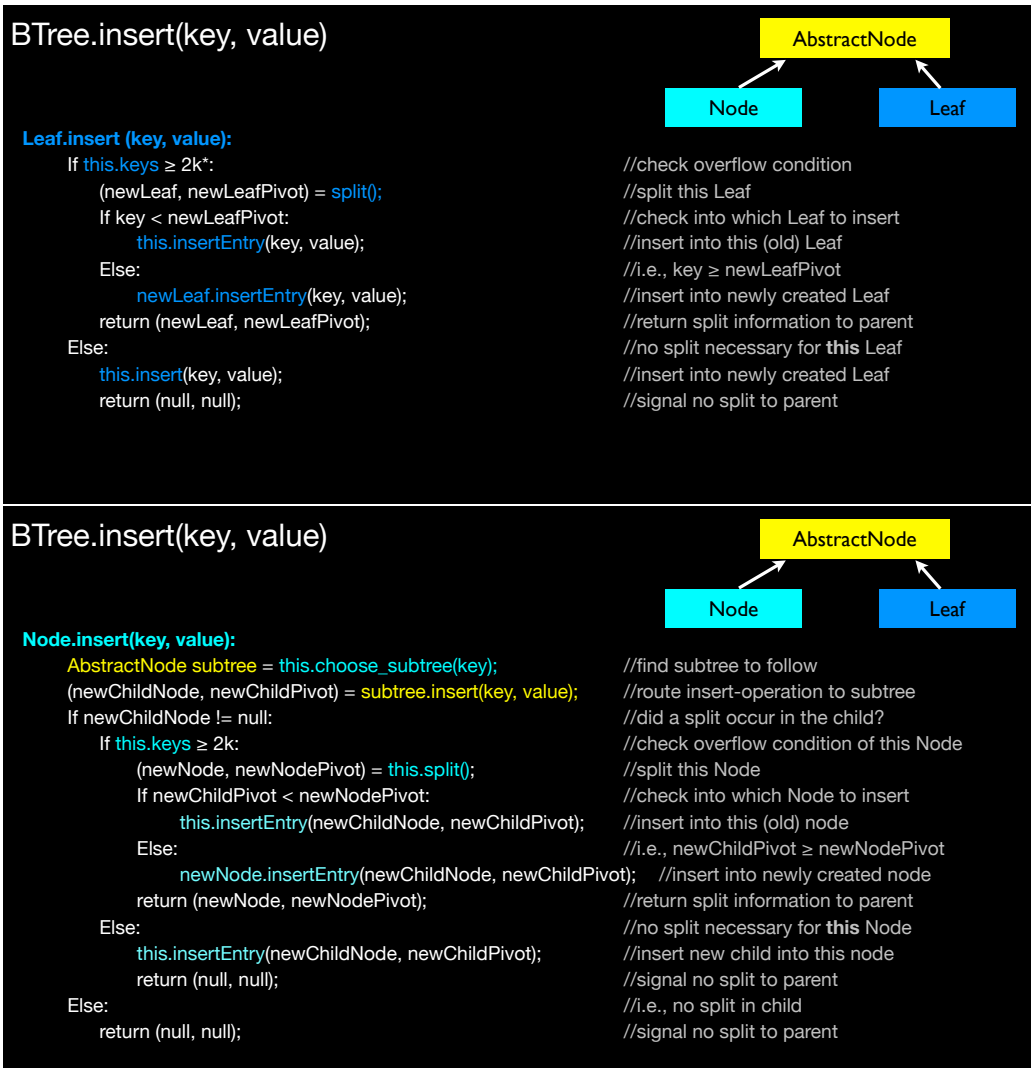


Figure 3.5: inserting into a leaf vs inserting into a node

The split-operation may be implemented elegantly by exploiting polymorphism. Introduce an AbstractNode either as an interface or as an abstract class. AbstractNode defines the signatures for the insert and split methods. Introduce two classes, Node and Leaf, both inheriting/implementing from AbstractNode. Like that walking down the tree in an insert operation can be implemented without knowing in the implementation whether the child pointed to is a node or a leaf. In addition, return-values of the split-method may be used to signal whether a split occurred. This allows you to easily detect whether any node visited when walking down has to be split.

delete

How do we delete data from a B-tree and its leaves and nodes?

A delete operation in a B-tree can be considered the inverse of an insert operation. When deleting a key/value-pair from a leaf, that leaf may underflow, i.e. we violate the constraint that a leaf (if it is not the root) should have $n \in [k^*, \ldots, 2k^*]$ key/value-pairs. In order not to violate the constraint we may therefore merge this leaf with a sibling leaf. In that process we also remove a pivot and a pointer from the parent node. In turn the parent may underflow as well. Then we need to merge the parent with one of its siblings. This

merge

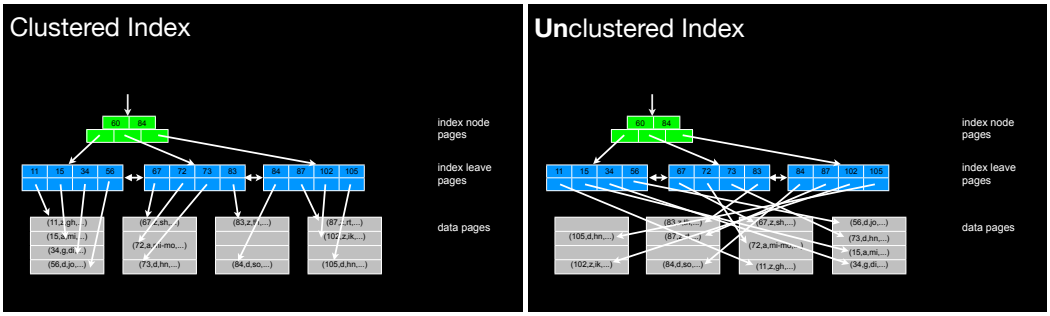


Figure 3.7: Clustered vs unclustered index

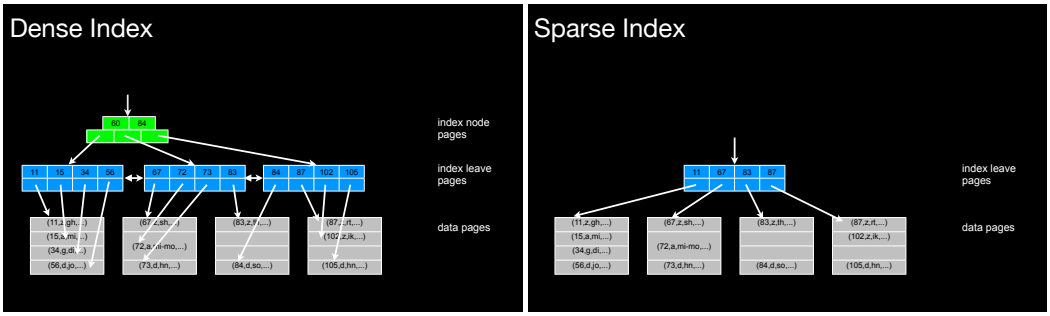


Figure 3.8: Dense vs sparse index

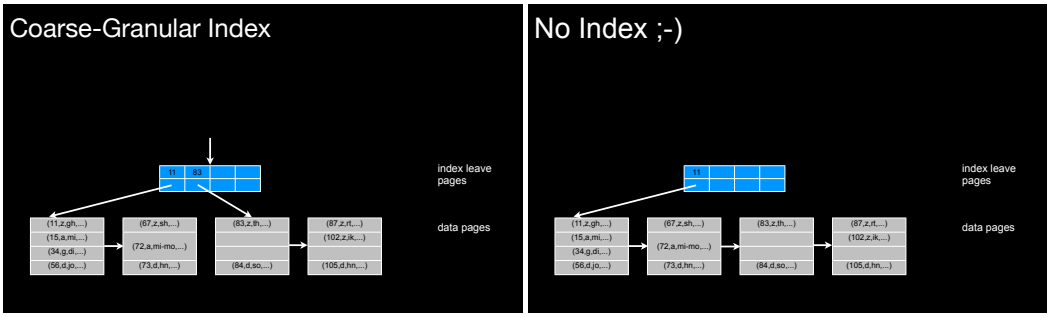


Figure 3.9: Coarse-granular vs no index

How many clustered indexes are possible for a table?

In the general case, i.e. if the attributes of the table are not correlated, only one clustered index may be created as sorting the table on one attribute will destroy the sort order on all other attributes. If some columns are correlated, multiple clustered indexes may be created (though few DBMS support this). If we combine replication with clustered indexes, we may create multiple clustered indexes even when columns are not correlated, e.g. keep the table in two sort orders and create different clustered indexes on these tables.

What is an unclustered index?

unclustered index

In an unclustered index the sort order of keys in the leaves *does not necessarily* corresponds to the sort order of tuples on the data pages. This implies that a range query can be answered only at relatively high costs: (1) execute a point query, (2) for each entry in a leaf visit the data page pointed to, and (3) continue with ISAM on the leaf-level and for each entry execute (2) until the end of the range is found. In other words, the index

3.2.5 Covering and Composite Index, Duplicates, Overflow Pages, Composite Keys

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[RG03], Section 10.7

Learning Goals and Content Summary

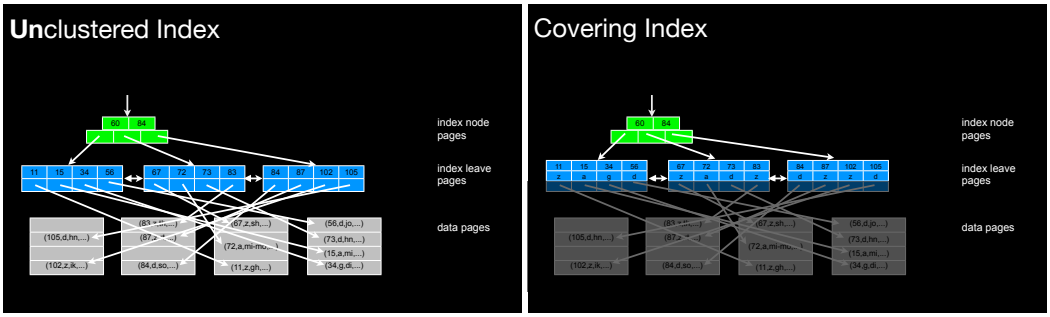


Figure 3.10: Non-covering vs covering index

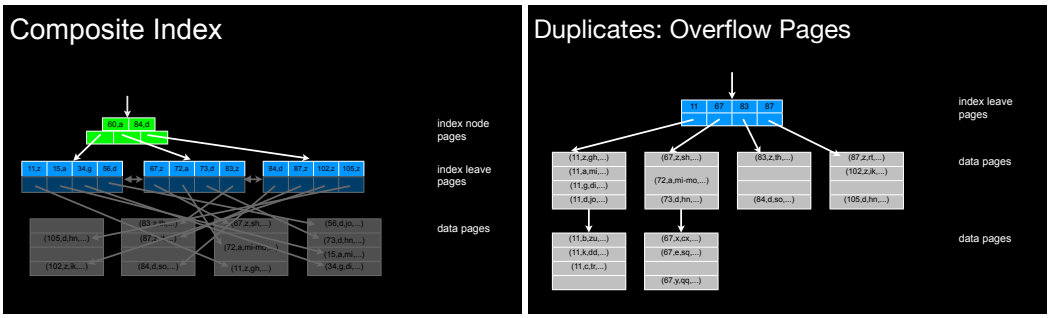


Figure 3.11: Composite (key) index vs overflow pages

covering index

What is a covering index?

In contrast to an unclustered index, a covering index stores additional attributes at the leaf-level — not only the key/value-pairs kept by an unclustered index. The advantage is that certain queries may be answered by considering the covered attribute(s) rather than looking up data on the data pages. For instance, assume the index maps from attribute A to rowIDs but additionally covers attribute B. Now, whenever we have a query SELECT A, B WHERE A>=72 AND A <= 87, we can compute the result to this query with an index only plan, i.e., we execute the range query on attribute A and for all entries qualifying we also output B. Compare this to an unclustered index where for every

index only plan

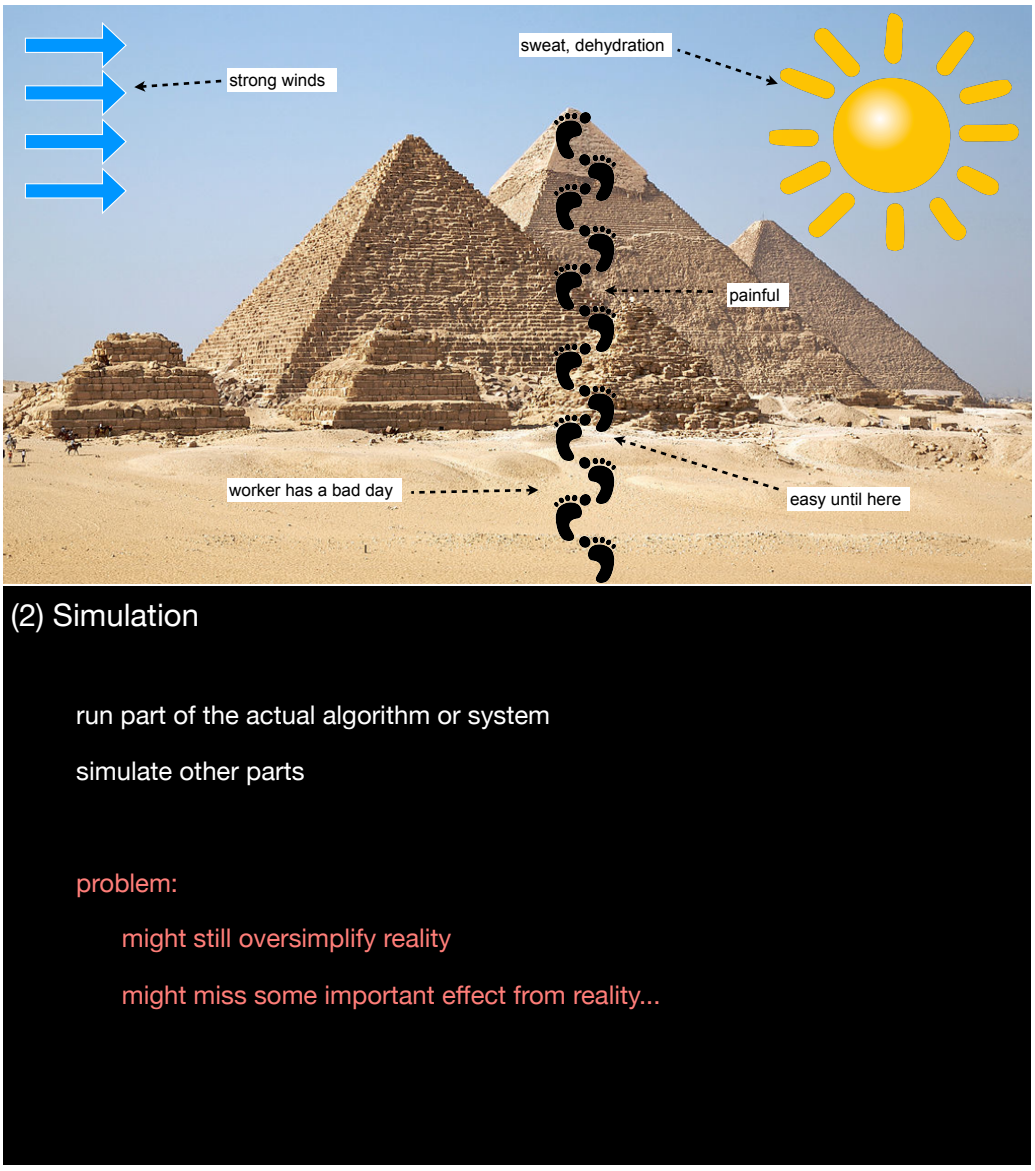


Figure 3.13: Simulating a pyramid

It is the root of one of the biggest confusions in computer science to solely reduce performance analysis to analytical modeling. As stated above, there are three different methods to analyze performance. Each method has its pros and cons. Just using analytical modeling, which typically argues along big O-notation, is often not helpful as analytical models tend to oversimplify. Therefore these models allow us only to draw relatively vague conclusions, e.g. algorithm X is of complexity class Y. This is unfortunate in situations where many algorithms are in the same complexity class, but vary in wall-clock time by orders of magnitude. The latter situation is the 90% case in database systems.

asymptotic complexity

What is asymptotic complexity?

Asymptotic complexity classifies the growth rate of a function w.r.t. a parameter. The input parameter is typically the problem size, e.g. the size of a dataset. The function itself models the runtime or memory consumption of an algorithm. We use big O-notation to

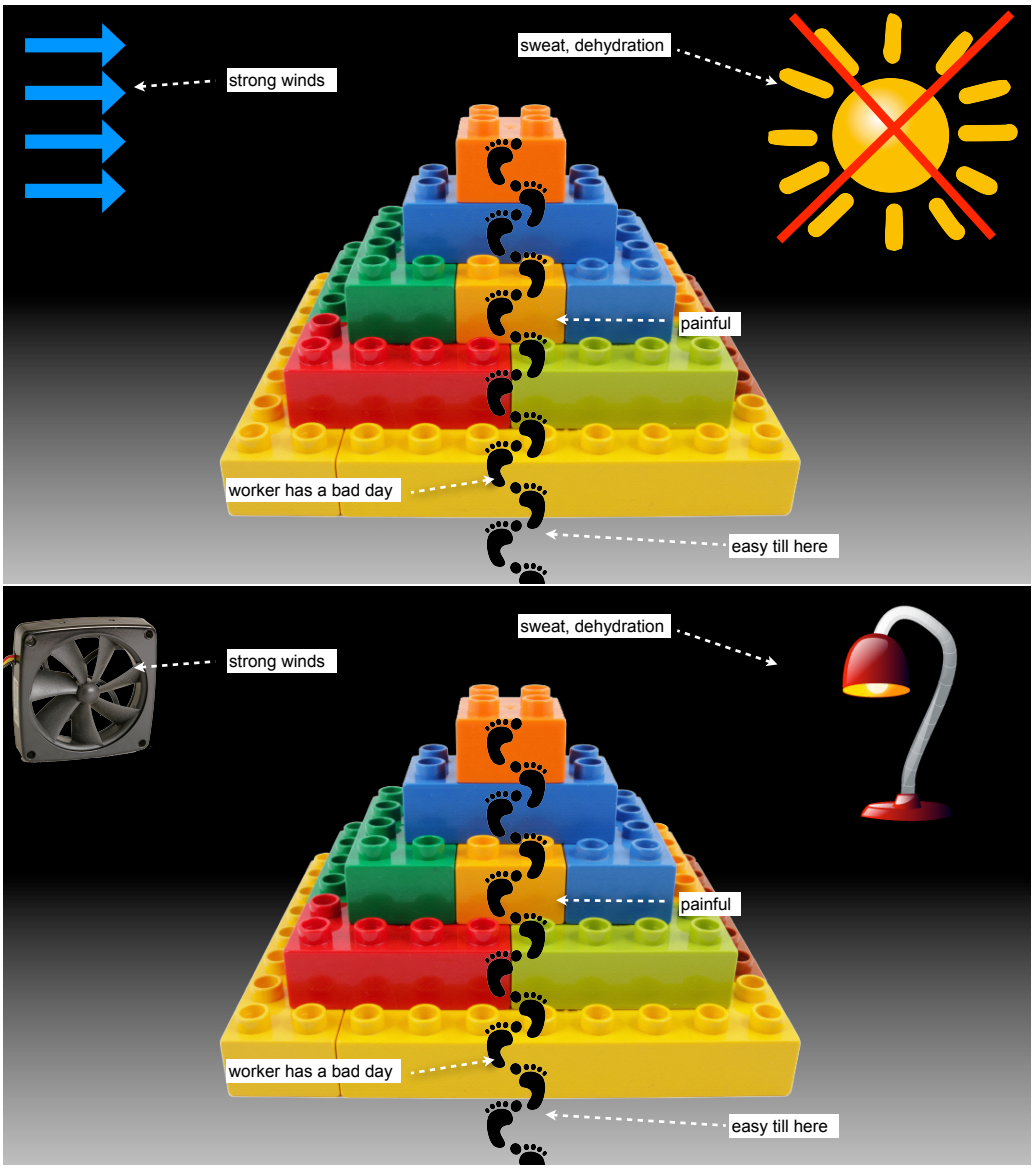


Figure 3.14: Additional influence factors when simulating a pyramid

characterize a function's growth rate.

cost model

What is a cost model?

A cost model goes beyond asymptotic complexity in that we do not only determine the complexity class of an algorithm or function, but additionally try to compute a cost estimate. For instance rather than saying that an algorithm is in complexity class $O(n \log n)$, we add constants, units, and terms of lower complexity. So we may come up with a cost formula like

$$C(n) = \frac{1}{42} \cdot \log_2(n) + 3.2 \cdot n + 2.5 \text{ microseconds}$$

.

Cost formulas are not only useful in performance analysis, but also to predict performance in a database system. Precise cost estimates are of utmost importance in cost-based optimization, see Section 5.2.

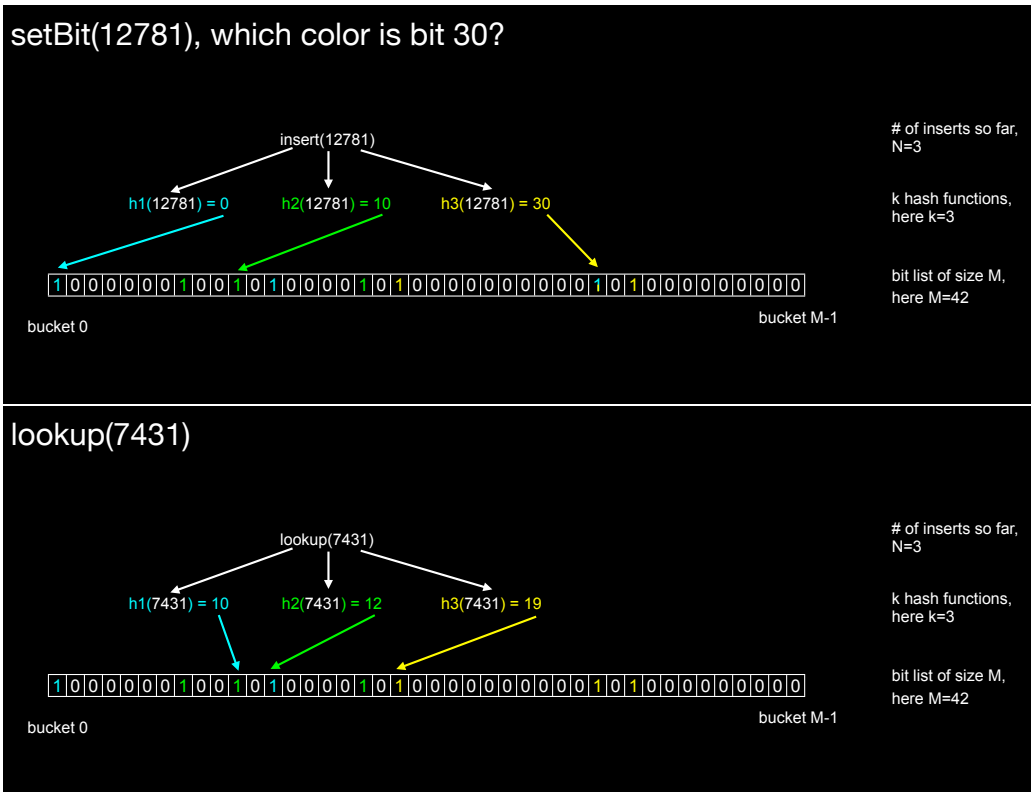


Figure 3.21: setting and looking up bits in a bloom filter

Learning Goals and Content Summary

What are the lookup semantics of a bloom filter? **bloom filter**

Recall that in a deterministic data structure (like binary search trees, B-trees, and value bitmaps) it holds that:

$$\text{index.hasKey}(key) = \texttt{true} \quad \Leftrightarrow \quad key \text{ exists in database.} \tag{3.1}$$

In contrast, a bloom filter is a probabilistic index structure. Here, condition 3.1 is weakened to: **probabilistic index**

$$\text{index.hasKey}(key) = \texttt{true} \quad \Leftarrow \quad key \text{ exists in database.}$$

This implies there are cases where

$$\text{index.hasKey}(key) = \texttt{true} \quad \wedge \quad key \text{ does } \textbf{not} \text{ exist in database.} \tag{3.2}$$

The case described by condition 3.2 is called false positive, i.e. the index claims that the underlying table contains the key, however, in reality the table does not contain that key. Hence, a bloom filter, just like sparse and coarse-granular indexes (recall Section 3.2.4), is a filter index. **false positive** **filter index**

What is the core idea of a bloom filter?

The core idea of a bloom filter is to support the method index.hasKey(key) through

Figure 4.5: Foreign-key constraint typically imply duplicates in one table only; relationship of joins to cogrouping

However, we may also assign *multiple* labels to each input tuple and thus replicate tuples into multiple partitions.

$$\texttt{CoGroup}(I_1, ...I_n, p_1(), ..., p_n()) \mapsto \left\{ \{O_1, \ldots, O_n\}_{l_1}, \ldots, \{O_1, \ldots, O_n\}_{l_m} \right\}.$$

Here, $I_1, \ldots, I_n$ are input relations. $p_1(), \ldots, p_n()$ are labeling functions where $p_i : I_i \mapsto \{L\}$. Here L is a discrete domain of labels. Every input to a p_i is mapped to a subset of L.

In other words, for each input tuple in any of the input sets I_i we assign a set of labels. For each distinct label $l_j \in L$ there will be one group of outputs $\{O_1, \ldots, O_m\}_{l_j}$.

cogroup

join

Why are cogroups a general property of equi-joins rather than a property of a specific join algorithm?

This follows directly from the definition of cogrouping. An equi-join identifies pairs of rows having the same join key. This is a property of the join as defined in relational algebra. It is not specific to a specific implementation of a join.

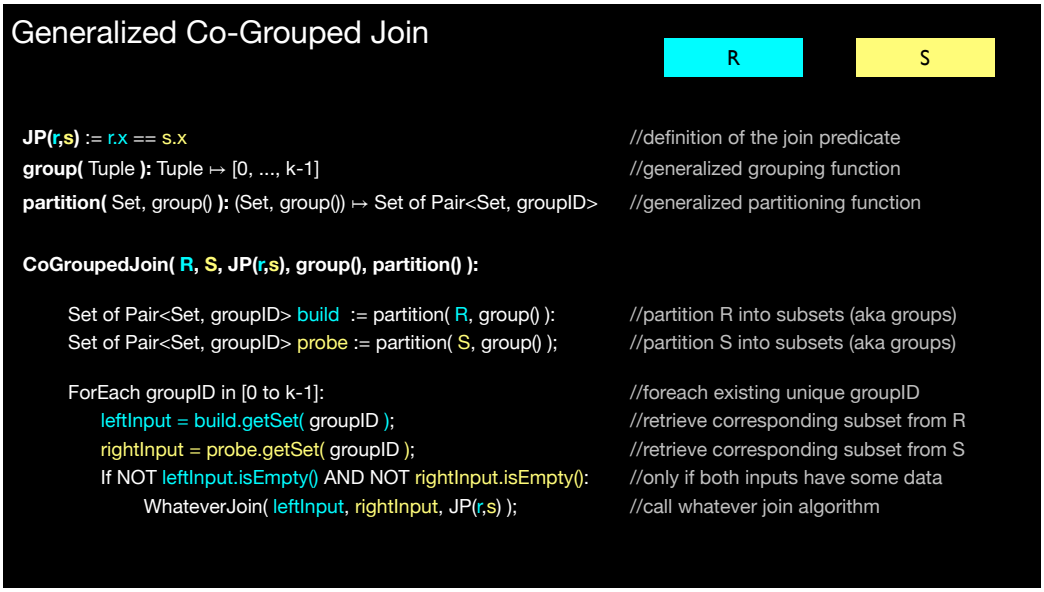


Figure 4.6: Pseudo-code of generalized cogrouped join

How can we express an equi-join using cogrouping?

We could express this join through a cogrouping operation in multiple ways:

1. Grouping by equality: all labeling functions $p_1(), ..., p_n()$ simply return the join key of their corresponding table. Like that, all tuples within a cogroup have the same key. In order to compute the join result, it is enough to compute a cross product within each cogroup.

2. Grouping by partitioning: all labeling functions $p_1(), ..., p_n()$ simply return a function of the join key. The same function is used for all input values. Like that, all tuples within a cogroup have the same return value for that function. However, tuples within the same cogroup may have different values in their join columns. Hence, we still need to perform a join within each cogroup.

Let's go back to our join predicate JP(r,s):= r.x==s.x. Assuming, $I_1 = R$ and $I_2 = S$, we could express this join as either:

1. Grouping by equality: $p_1() = r.x$ and $p_2() = s.x$.

2. Grouping by partitioning: $p_1() = r.x/k$ and $p_2() = s.x/k$. Here k is the number of cogroups.

Could we use a cross product inside a cogroup rather than a join algorithm?

cross product

Again, yes, if the grouping is done by equality.

Which three special cases can be implemented with this algorithm?

1. Grace Hash Join: a join algorithm that partitions data into buckets on disk. Then each cogroup is joined using a main-memory join.

4.3.3 Late, Online, and Early Grouping with Aggregation

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

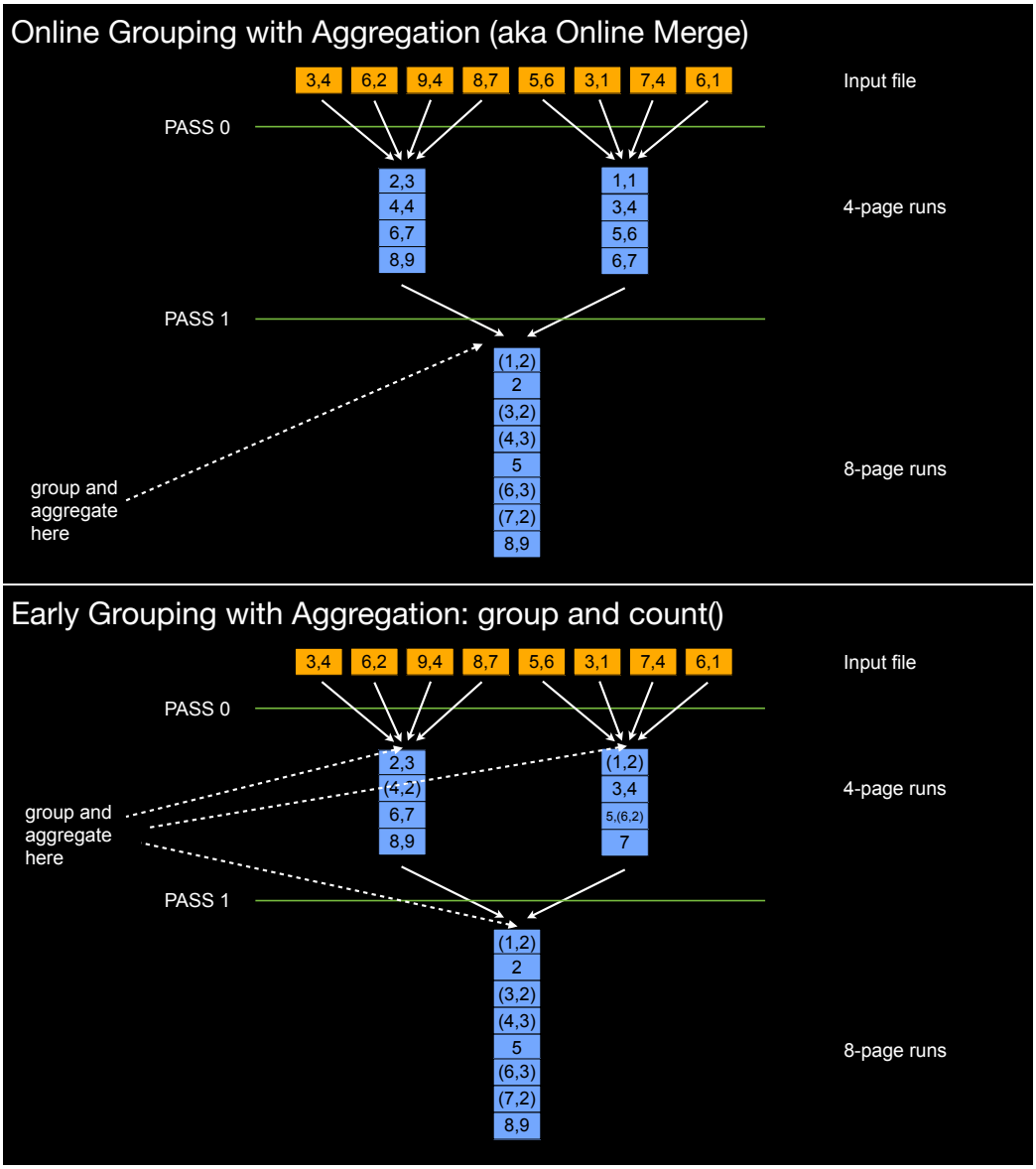


Figure 4.17: Grouping and aggregation: online vs early

Why should we be careful with terminology here?

Typically in database literature the term "aggregation" is assumed to include a grouping operation as well. So, any particular aggregation method may also additionally include a

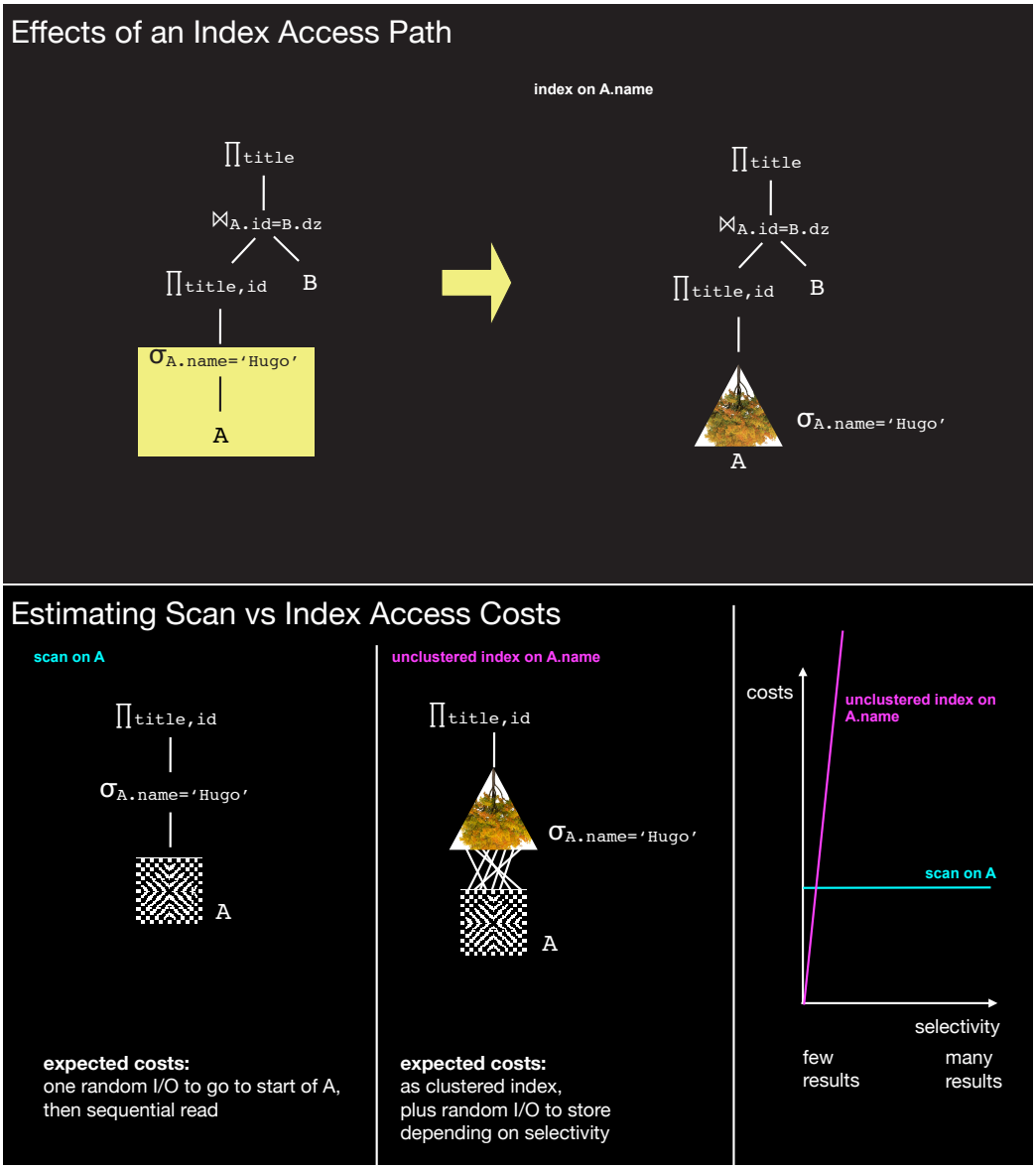


Figure 5.5: Effects of an index access in a plan; estimating costs for scans vs estimating costs for index accesses

qualify). Hence, in order to pick the right access plan, we have to understand for each table mentioned in a query: what type of indexes exists for which attribute? What are the costs of using that index rather than scanning the table? In order to answer this question, the query optimizer requires statistics about the data distributions and good estimates on predicate selectivities.

How does the query optimizer decide which access plan to take?

The query optimizer has to estimate the costs of the different alternatives. Based on these estimates, the query optimizer picks the plan with the lowest estimated costs. Keep in mind that those estimates may be wrong, i.e. the optimizer may pick a plan that is not the most efficient plan in reality.

What are the possibly different costs of picking a scan, clustered index, unclustered index or covering index in a disk-based system?

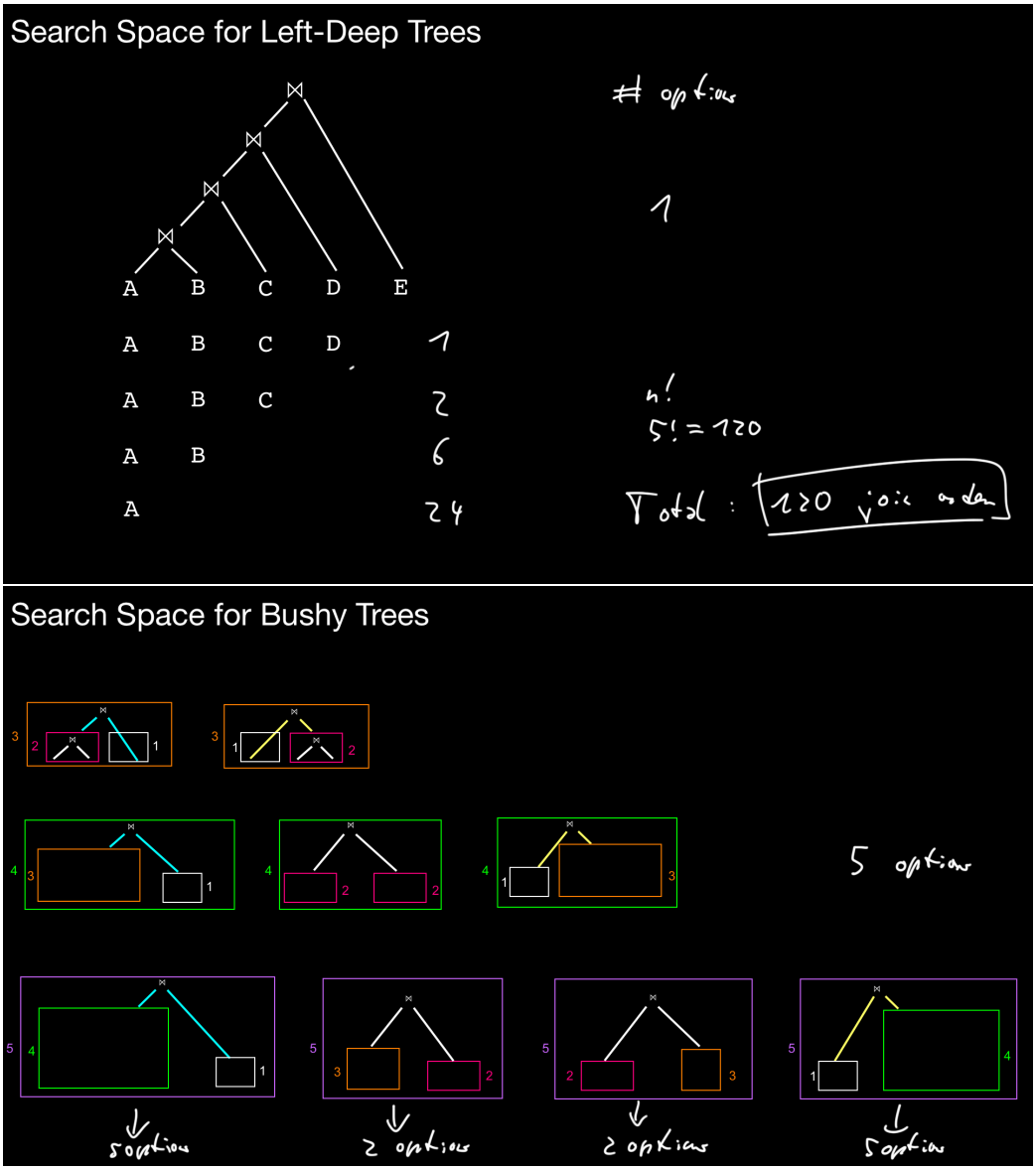


Figure 5.6: Search space for left-deep and bushy trees

What is the number of possible bushy trees for n input relations?

This is given by the Catalan number C_{n-1} which is defined as:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)n!n!} = \frac{(2n)!}{(n+1)!n!}$$

or as a recurrence relation:

$$C_0 = 1 \text{ and } C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \text{ for } n \geq 0.s$$

For $n = 0, 1, 2, 3, \ldots$ this yields:

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, \ldots$$

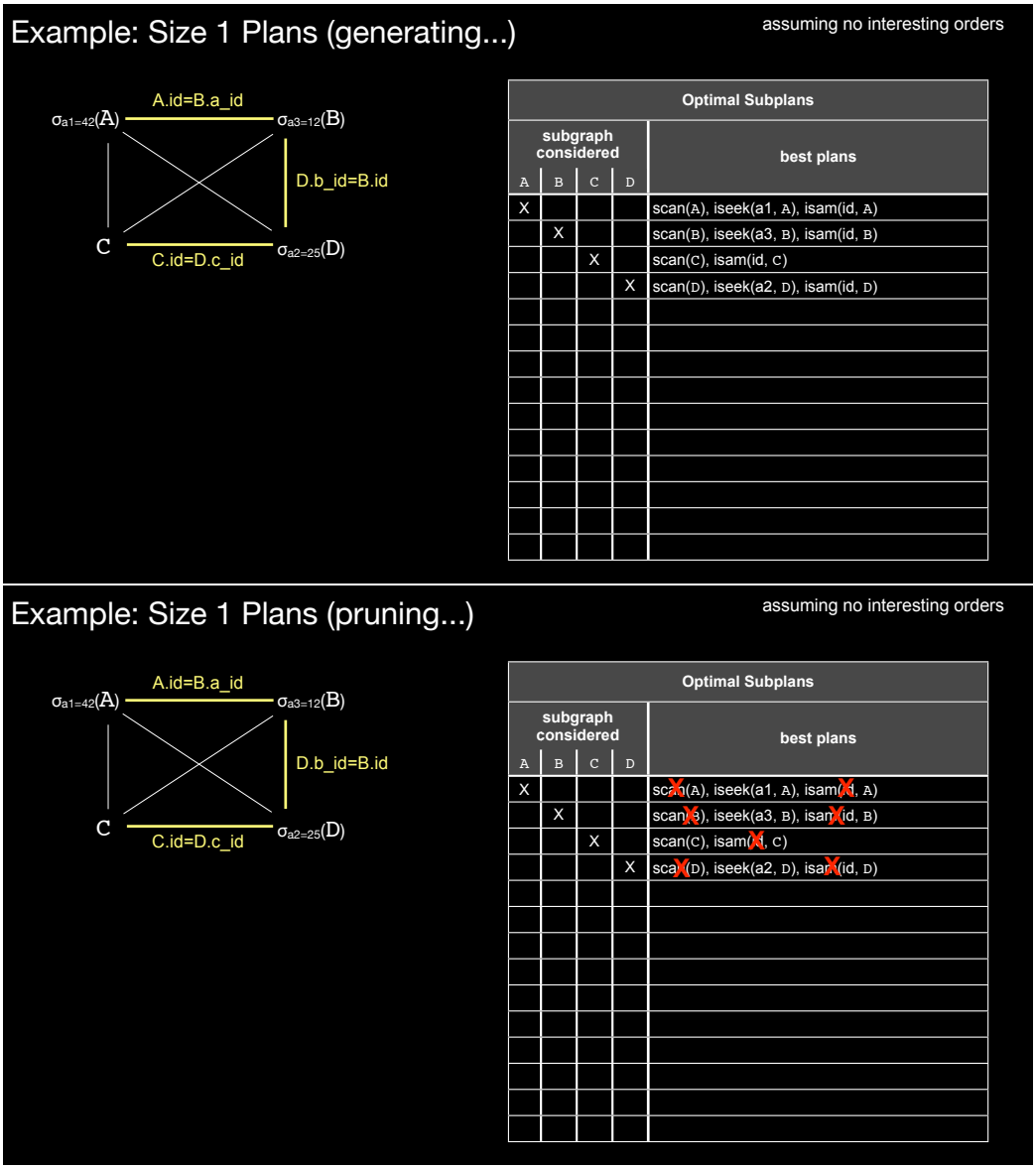


Figure 5.8: Dynamic programming: size 1 plan generation and pruning

The pruning step removes all subplans that are expected to have higher costs than other subplans on the same subset of input relations. In other words, for each subset, we keep a single plan that is estimated to have the lowest costs.

iteration

How many different entries does the table have at each iteration step?

A single iteration for *plansize* adds $\binom{n}{plansize}$ entries to the table, i.e. the number of subsets of size *plansize*. This calculation assumes that all subplans are enumerated even though they contain cross products. If plans with cross products are ignored, fewer plans will be inserted. In the worst case, the total number of entries required is:

$$\sum_{plansize=1}^{n} \binom{n}{plansize} = 2^n - 1.$$

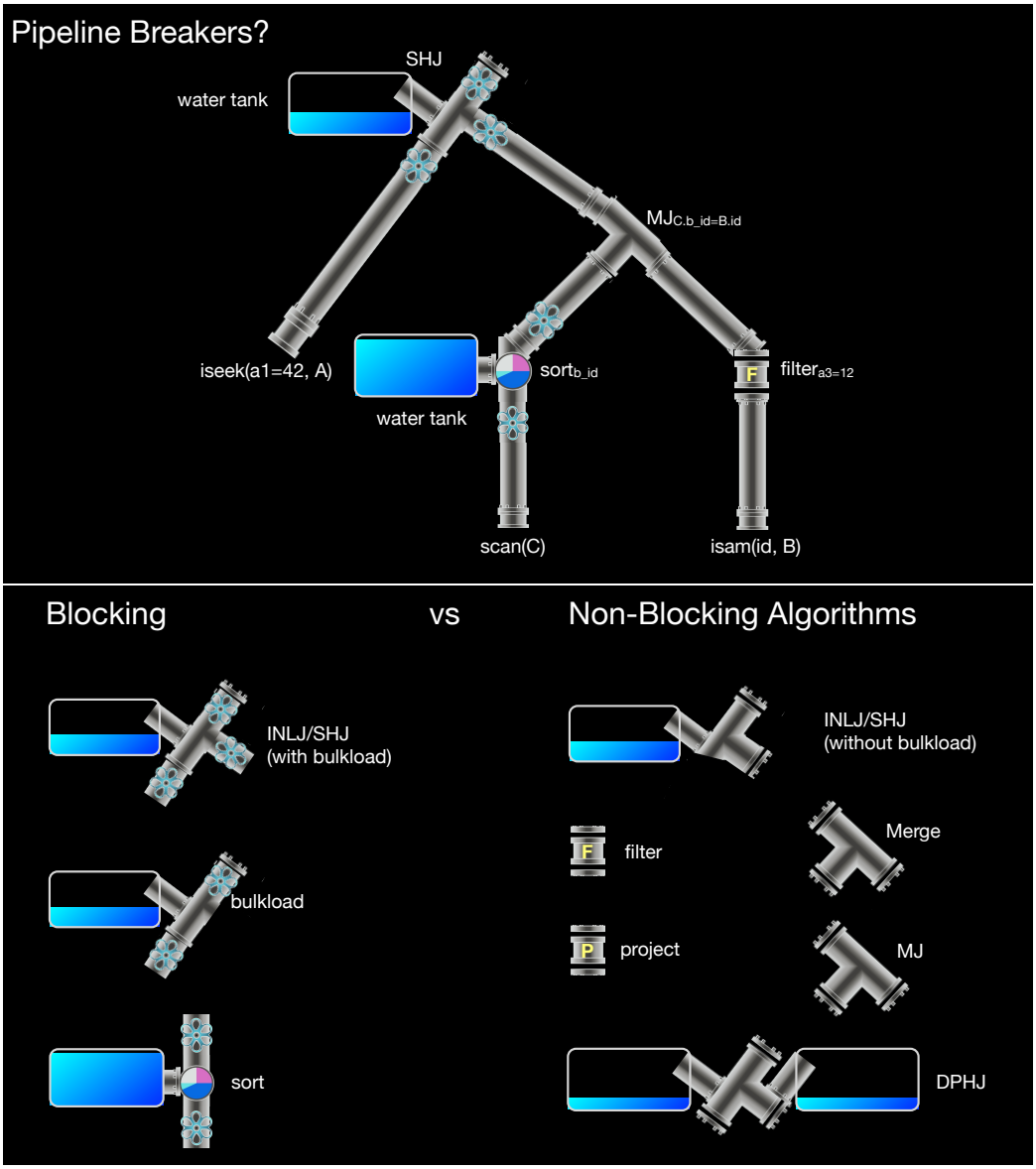


Figure 5.12: Blocking and non-blocking algorithms used in query pipelines

to all those functions separately.

2. precompiled modules: we wrap each physical database algorithm into an operator. Operators are connected through some sort of pipelining mechanism.

3. code compilation: we compile the physical query plan into executable code.

What is the problem with translating a query plan using a function library?

function library

If we translate a query to a series of function calls, each function will run to completion, materialize its entire result and only then returns it as a parameter to the (possible) next function call. This may be a bad idea, in particular for large intermediate results. For instance, consider a join producing a large intermediate result which is the input to another join. The entire result of that join needs to be materialized and temporarily stored. This consumes both storage space and storage bandwidth.

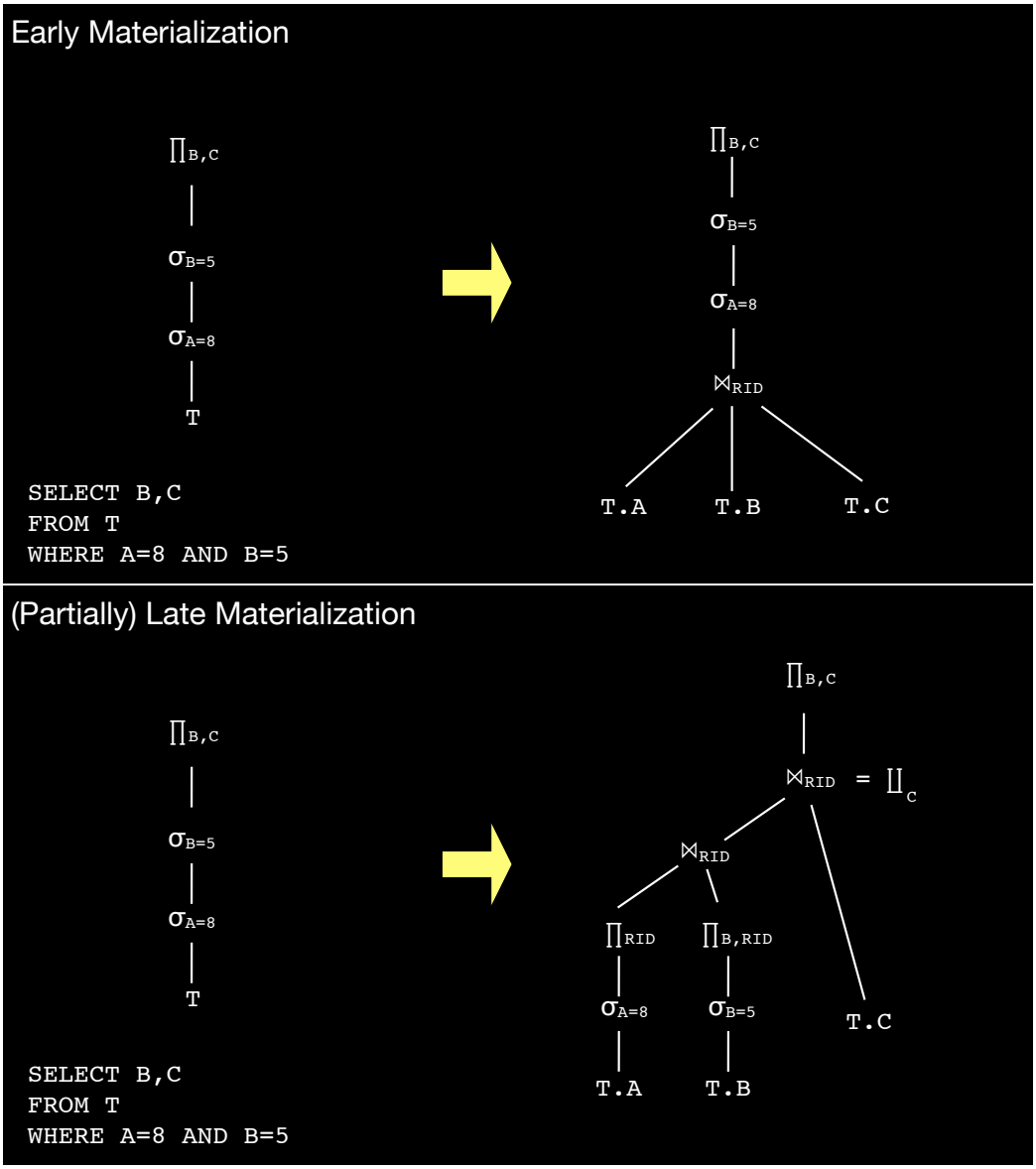


Figure 5.15: Early vs (partially) late materialization

Whether we name this technique partially *late* materialization or partially *early* materialization does not make a difference. The key property of this technique is that some attributes are read early as in early materialization, however other attributes are read later in the pipeline as in late materialization, e.g. when they are actually required in a particular operation, e.g. a join or in order to produce the final result tuples. Thus this technique is a combination of both early and late materialization. An example for partially late materialization is shown in Figure 5.15. Here, attributes A and B are materialized early and filtered. Then we intersect their RID-lists to compute the conjunct (the AND in the WHERE-clause). Only after that we read attribute C (this is the late materialization part) and perform a RID join.

anti-projection

tuple reconstruction join

What is an anti-projection and what is the relationship to tuple reconstruction joins?

An anti-projection is the inverse operation to a projection. Recall that in relational

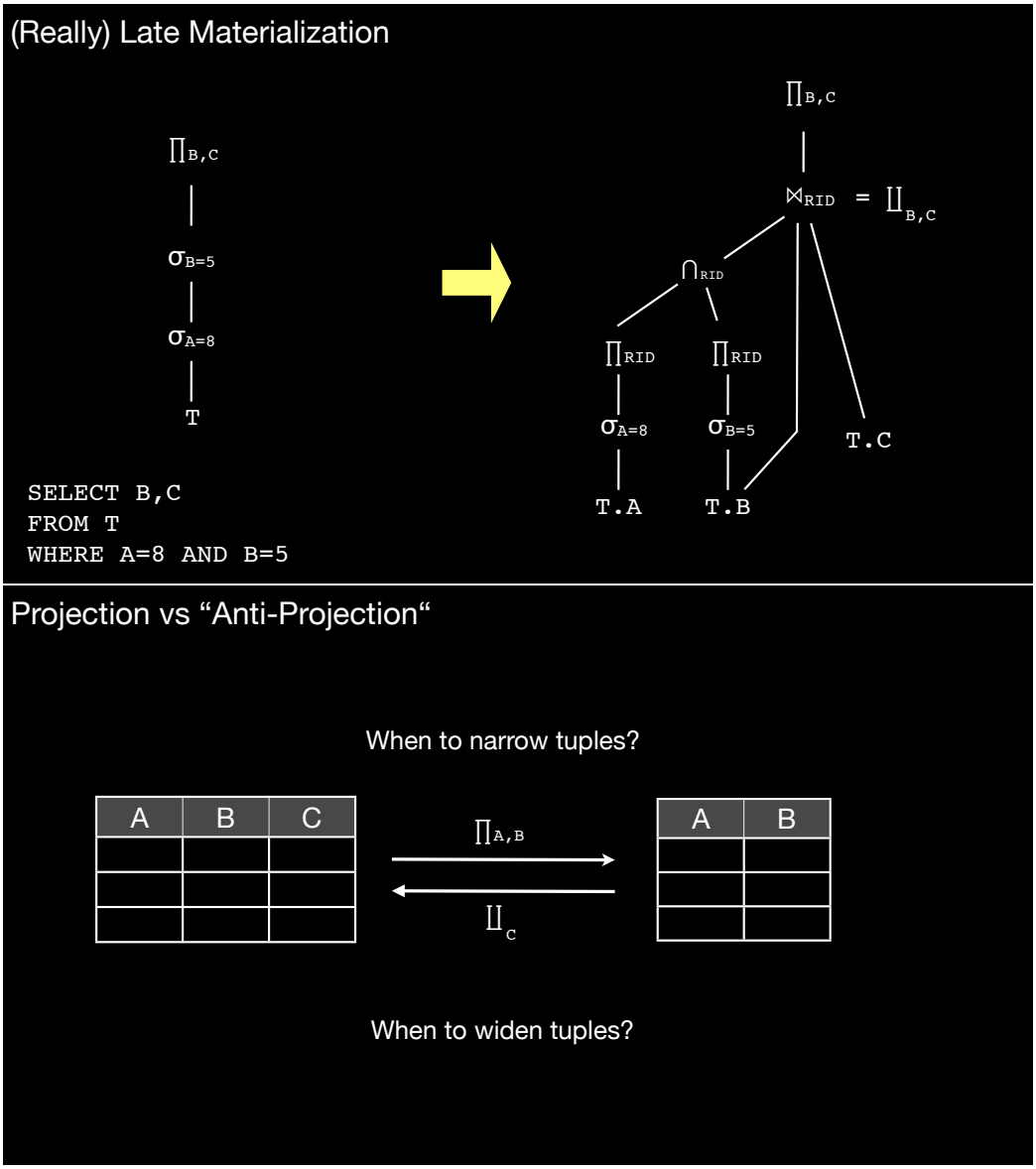


Figure 5.16: Late materialization and the relationship of projection and anti-projection

algebra a projection removes some of the input attributes, i.e. it narrows the schema[1]. In contrast, an anti-projection adds attributes to an input schema, i.e. it widens the schema.

How could we implement early materialization in a column store?

Again, early materialization simply implies that all required attributes are read, joined, and typically converted into a row-layout which corresponds to a materialized view of the input.

What is the impact on query processing?

The strategy used for tuple reconstruction may have considerable impact on the overall runtime of a query. In particular in cases where many attribute values need to be anti-projected, tuple reconstruction costs may overshadow all other costs of the query plan.

[1]Technically, a projection may also keep the input schema as is. In that case the projection does not make much sense though.

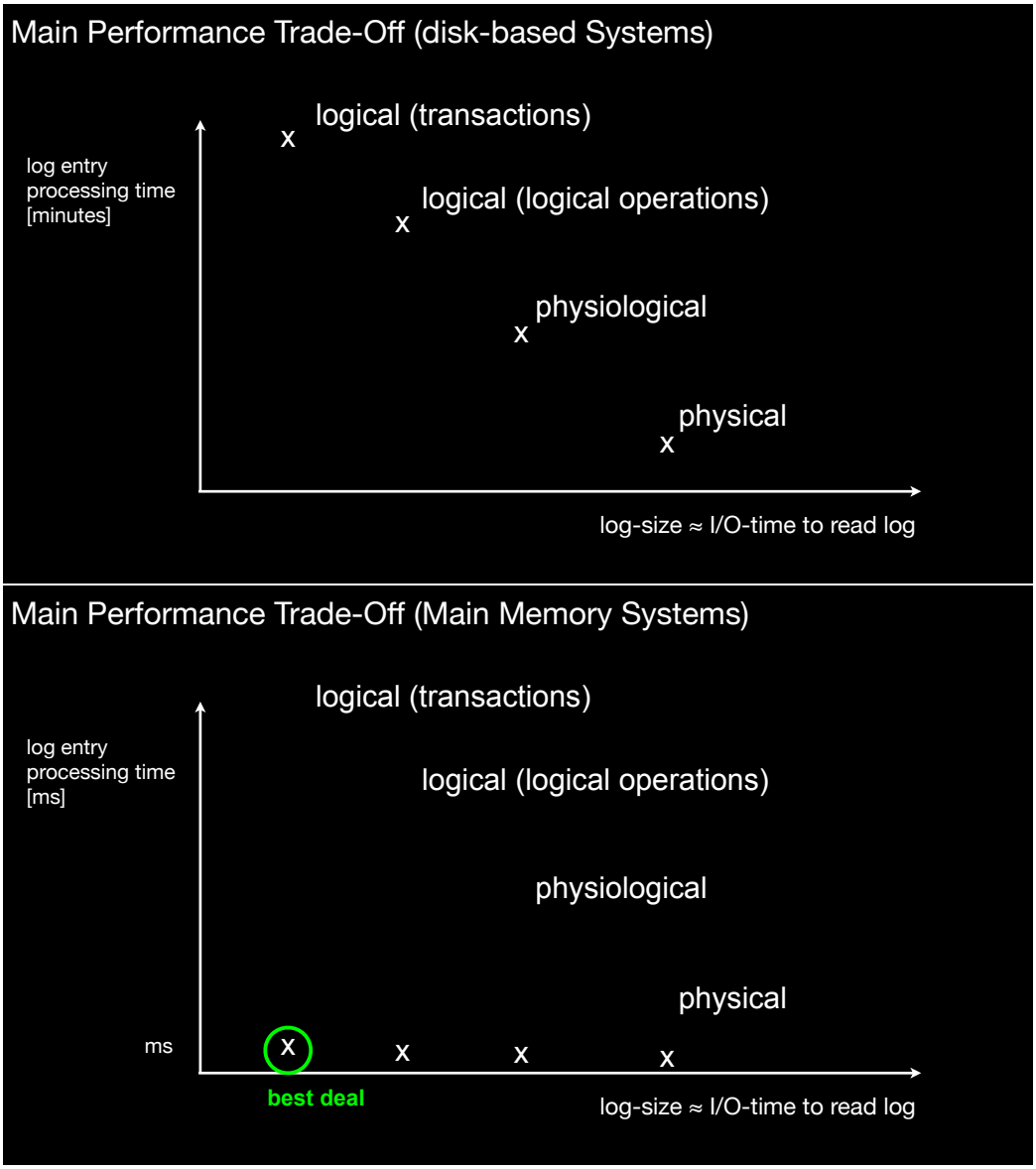


Figure 6.5: Logging trade-offs in different database systems: disks-based vs main-memory systems

where logical undos are translated into physical or physiological log records).

What are the performance trade-offs for the different logging variants in a main-memory system?

In a main-memory system the processing time for a particular log record is on a completely different scale than it is in a disk-based system: milliseconds rather than seconds/minutes. Therefore, the size of the log and the time it takes to read it, may severely limit the overall recovery time. Therefore, it makes sense to keep log entries as small as possible, e.g. by just storing queryIDs and their invocation parameters, similar to what is done in prepared statements and stored procedures. While those queries are running and considered uncommitted, the system must keep a main-memory resident undo-log, in case that query cannot be committed successfully.

dictionary compression

What is the relationship between logical logging and dictionary compression?

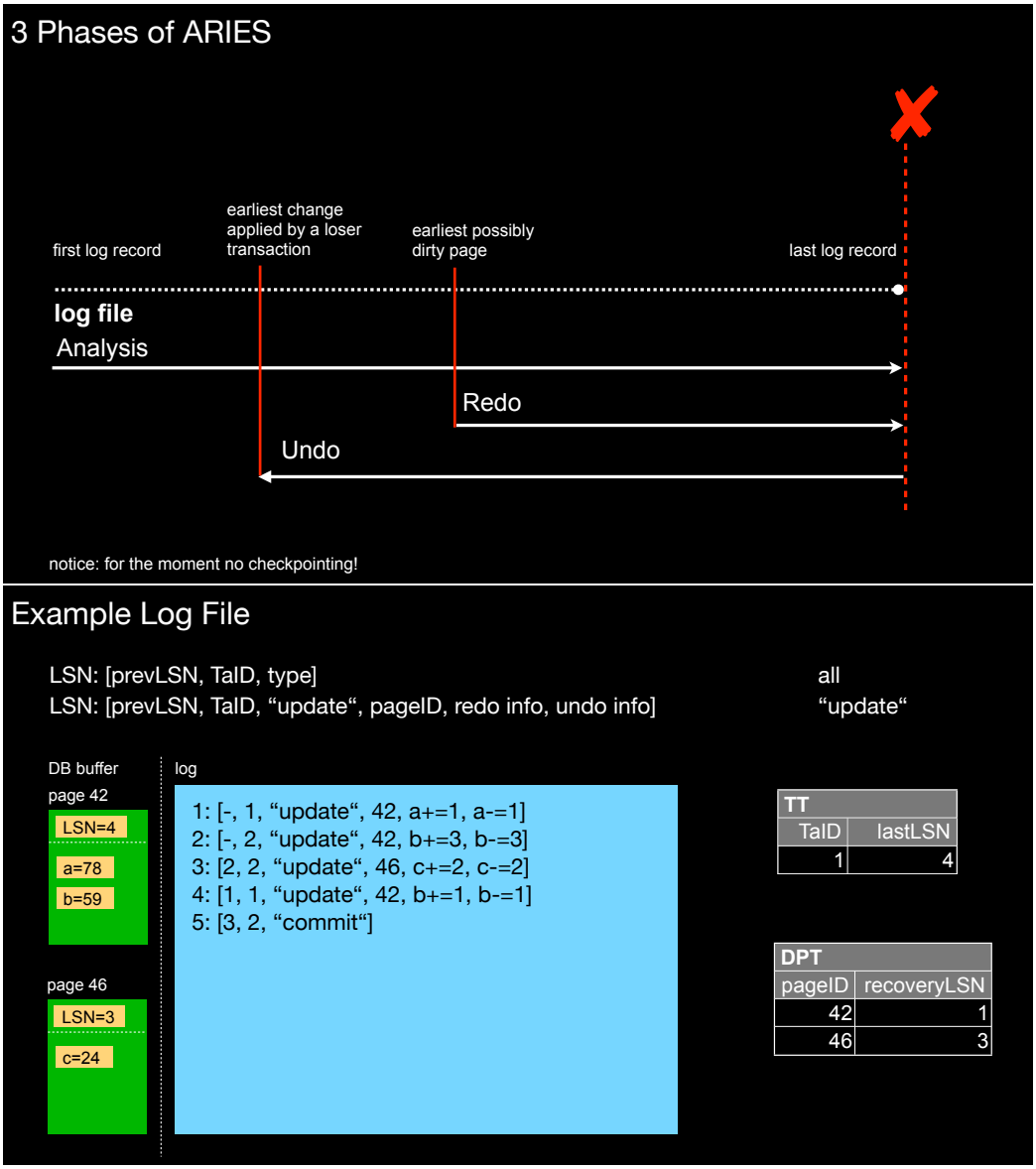


Figure 6.6: The three phases of ARIES and an example logging scenario showing the DB buffer, the log, the transaction table (TT) and the dirty page table (DPT)

The transaction table (TT) maintains the list of ongoing (uncommitted) transactions. This is useful during recovery to identify the loser transactions, i.e. the transactions that did not commit and whose changes must be removed from the database store. In the TT, for each transaction we keep the last log record triggered by that transaction. This is called the lastLSN. This implies that if a transaction performs another change triggering a new log record, its entry in the TT must be updated. If a transaction commits, it may be removed from the TT.

lastLSN

What is the purpose of the dirty page table (DPT)?

dirty page table

DPT

The dirty page table (DPT) maintains the list of pages in the database buffer which have not yet been written back to the database store. This is useful during recovery to identify pages that were not persisted in the database store w.r.t. their latest version, i.e. some of the changes applied to those pages in normal operation were not persisted in the database

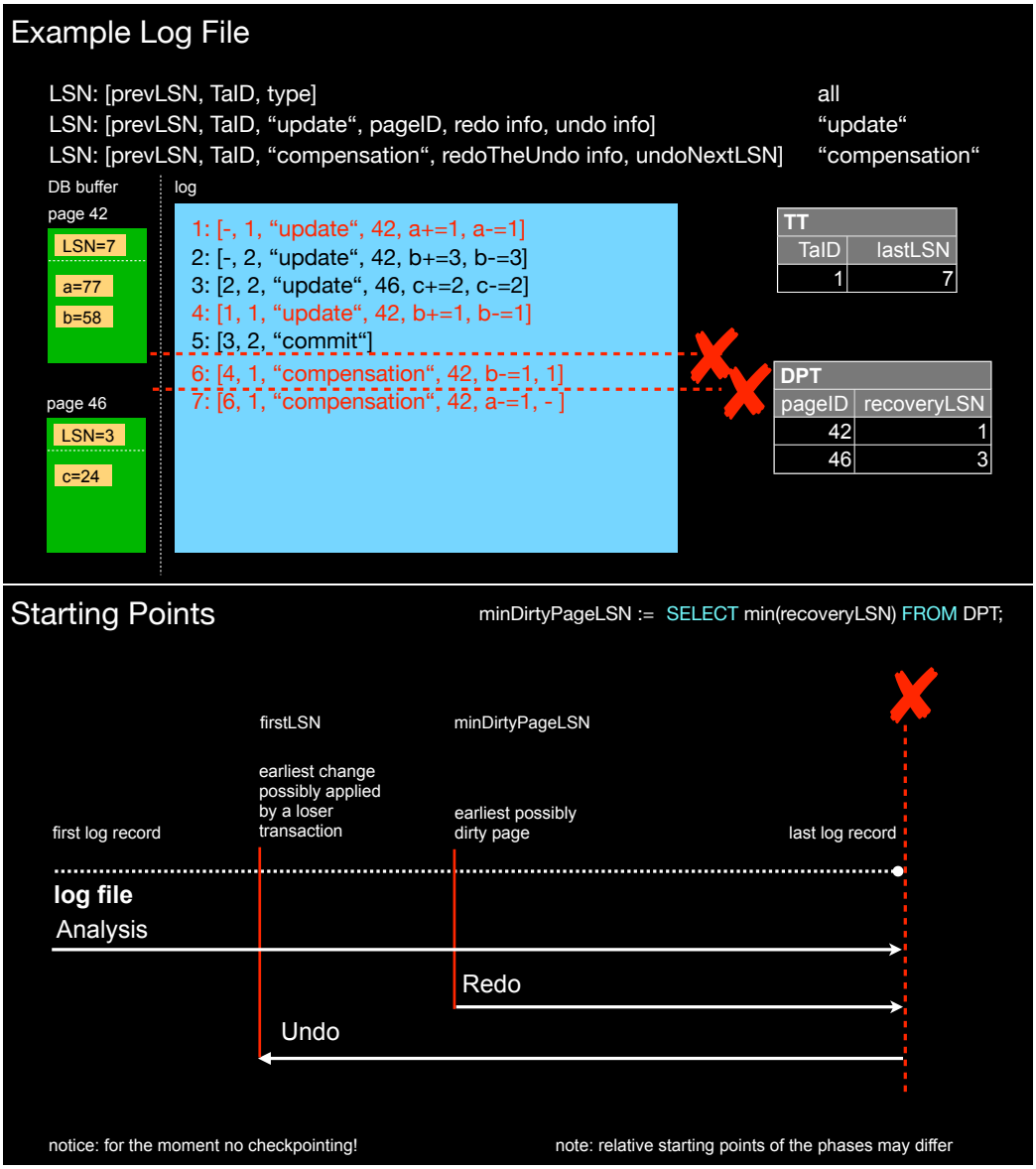


Figure 6.7: The logging scenario under multiple crashes and a schematic view on the different possible starting points for the three different phases in ARIES

store. On those "dirty pages" we may have to redo information during recovery later on. For each page in DPT we keep the *recoveryLSN*. That is the LSN that did the first change to that page making it different from the version on disk, i.e. this was the first change applied to that page converting it from a clean page to a dirty page. This implies that if a transaction performs another change triggering a new log record on a page that already exists in DPT, its recoveryLSN *must not* be updated. The only way to remove entries from DPT is to write dirty pages back to the store, e.g. using a background thread. In that case the page should be removed from DPT as it is then considered a clean page.

Which assumptions do we make in ARIES-recovery?

We assume that pages may contain changes from loser and winner transaction at the same time, i.e. two concurrent transactions may operate on the same page.

update log record

What information is contained in an update log record?

Appendix A

Credits

Figure 1:

©iStock.com: TadejZupancic

Figure 2:

©iStock.com: hidesy, moenez; Rastan; hatman12; mtphoto

CC: Appaloosa http://commons.wikimedia.org/wiki/File:DRAM_DDR2_512.jpg

http://creativecommons.org/licenses/by-sa/3.0/deed.en

Lasse Fuss http://commons.wikimedia.org/wiki/File:Lufthansa_A380_D-AIMA-1.jpg

http://creativecommons.org/licenses/by-sa/3.0/deed.en

Jahoe http://commons.wikimedia.org/wiki/File:Schiphol_Amsterdam_airport_control_tower.png?uselang=de

http://creativecommons.org/licenses/by-sa/3.0/deed.de

also used in other figures

Figure 1.2:

©iStock.com: voyager624

also used in other figures

CC: BY-SA Thomas Tunsch / Hula0081110.jpg (Wikimedia Commons)

http://de.wikipedia.org/w/index.php?title=Datei:Hula0081110.jpg&filetimestamp=20070305150205

http://creativecommons.org/licenses/by-sa/3.0/deed.de

as well as public domain

Twix analogy inspired from:

http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait?

[retrieved Nov 8, 2013] yet: I extended the analogy a bit

Cache latency numbers are based on this article:

Performance Analysis Guide for Intel's CoreTM i7 Processor and Intel's XeonTM 5500 processors?

By Dr David Levinthal PhD. Version 1.0

http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf?[retrie
Nov 8, 2013]?

Figure 1.6:

©iStock.com: mtphoto

also used in other figures

Figure 1.14:

©iStock.com: nico_blue; ludinko

CC: Asim18

http://commons.wikimedia.org/wiki/File:SanDisk_SD_Card_8GB.jpg?uselang=de

http://creativecommons.org/licenses/by-sa/3.0/deed.de

and public domain

Figure 2.1:

©iStock.com: mtphoto

CC: Appaloosa

http://commons.wikimedia.org/wiki/File:DRAM_DDR2_512.jpg

http://creativecommons.org/licenses/by-sa/3.0/deed.en

Intel Free Press

http://www.flickr.com/photos/54450095@N05/6345916908

http://creativecommons.org/licenses/by/2.0/deed.de

Austinmurphy at en.wikipedia

http://en.wikipedia.org/wiki/File:LTO2-cart-purple.jpg

http://creativecommons.org/licenses/by-sa/3.0/deed.en?

Figure 2.11:

CC: Wolfgang Beyer

http://en.wikipedia.org/wiki/File:Mandel_zoom_08_satellite_antenna.jpg

http://creativecommons.org/licenses/by-sa/3.0/

and public domain

Figure 2.12:

©iStock.com: mtphoto

Figure 3.1:

CC: Summi at the German language Wikipedia:

http://commons.wikimedia.org/wiki/File:Wegweiser_Foggenhorn.jpg

http://creativecommons.org/licenses/by-sa/3.0/legalcode

Figure 3.12:

©iStock.com: Freerick_k

Figure 3.13:

CC: Ricardo Liberato

http://de.wikipedia.org/wiki/Datei:All_Gizah_Pyramids.jpg

http://creativecommons.org/licenses/by-sa/2.0/legalcode

other:

http://openclipart.org/image/800px/svg_to_png/26274/Anonymous_Right_Footprint.png

http://openclipart.org/detail/26217/left-footprint-by-anonymous

http://openclipart.org/detail/22012/weather-symbols:-sun-by-nicubunu

Figure 3.14:

iconshock

http://commons.wikimedia.org/wiki/File:Desk_lamp_icon.png

http://creativecommons.org/licenses/by-sa/3.0/legalcode

User Smial on de.wikipedia

http://commons.wikimedia.org/wiki/File:Luefter_y.s.tech_pd1270153b-2f.jpg

http://creativecommons.org/licenses/by-sa/2.0/de/legalcode

Figure 3.15:

public domain:

http://commons.wikimedia.org/wiki/File:The_Blue_Marble.jpg

Appendix B

Youtube Quotes

- very nice refresher on database architecture, and very very very well presented with pictures and examples !! thanks

- These series of videos on databases are really great. Please keep posting more

- Great Video,Very precise.

- very nice video

- Amazing video! really helped understand the concepts! one of the best videos I found online!

- super

- Thanks for making this very useful video. Appreciate it

- Thank you, my book doesn't actually explain well how shadow paging works, but this cleared my mind.

- Awesome!

- love your videos

- Thanks for video Understand whole video.. nice job ... Thanks alot

- Obrigado, este video foi muito instrutivo para mim, vai me ajudar bastante em meu projeto interdisciplinar.

- These videos are really helpful. Thank you so much ..

- Vielen Dank für die tollen Videos Herr Dittrich. Sie schaffen die große Leistung und Kunst, komplizierte Dinge einfach zu erklären und dann noch mit Humor :)

- Thank you for giving this course and sharing this fundamental knowledge in a very clear way. I follow the complete course on Datenbankenlernen.de. Thank you so much!

- best explanation on early and late materialization

- Thanks for uploading these very valuable videos. Really appreciate the time you take explain to database to novices. Bless!

- Ich muss wiederholt sagen: Ich habe noch nie einen Prof. gesehen, der Dinge so klipp und klar rüber bringen kann, wahnsinnig viel Wissen in kurzer Zeit und man behält es

auf Anhieb. Ich wünschte meine Uni würde diesen VL-Stil verfolgen, sehr gut, weiter so!

- Also, ich weiss nicht was ich ohne ihren Kanal machen würde!? Doch.....verzweifeln!!! Großes Dankeschön!

- Excellent video!

- Nice video, Thank You.

German:

- Danke für die hilfreichen Videos.

- Danke!!

- Vielen Dank für die GEILE Vorlesung!! Eine komplette Vorlesung auf akademischem Niveau und das völlig kostenfrei. Ich bin Ihnen wirklich sehr dankbar für diese Hilfestellung!

- Gute Erklärung! Finde es vorallem sehr gut organisiert, sodass man nie Schwierigkeiten hat das nachfolger Video zu finden :)

- Sehr sehr Gut! Danke für die super Erklärungen. Da macht studieren doch wieder Spaß :D!

- super super super! Vielen Dank!

- Warum haben wir nicht so einen Dozenten ;-/ Danke fürs Video!

- Du rettest mich in meinem Studium :D

- deine Beispiele sind echt witzig "Wirtschaftskrise / Sklave Chef :D"

- Vielen, vielen Dank für die Videos, das hilft ungemein!

- Sehr gut erklärt! :) Danke

- ich habe sehr lange nach guten videos über dieses thema gesucht. danke sehr!! sehr gut erklärt und hilfreich!

- sehr gute erklärung !

- Wenn das bei uns in der Vorlesung oder in den Folien auch mal so verständlich gewesen wäre....:D

- Ich finde die gesamte Vorlesungsreihe super interessant, nicht so trocken dargestellt wie manch andere das tun.

- Tolle Videos mit sehr guten Erklärungen!

Bibliography

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[AMDM07] D.J. Abadi, D.S. Myers, D.J. DeWitt, and S.R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, 2007.

[APR+98] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable Sweeping-Based Spatial Join. In *VLDB*, pages 570–581, 1998.

[ASDR14] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. Main Memory Adaptive Indexing for Multi-core Systems. In *DaMoN*, June 23, 2014.

[Bac73] Charles W. Bachman. The Programmer As Navigator. *Commun. ACM*, 16(11):653–658, November 1973.

ACM version  ,

UDS campus version  .

[BBD+01] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB*, pages 39–48, 2001.

[Cod82] E. F. Codd. Relational Database: A Practical Foundation for Productivity. *Commun. ACM*, 25(2):109–117, February 1982.

ACM version  ,

UDS campus version  .

[CSRL09] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[dBBD+01] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB*, 2001.

[dBS01] Jochen Van den Bercken and Bernhard Seeger. An Evaluation of Generic Bulk Loading Techniques. In *VLDB*, pages 461–470, 2001.

[Fra97] Michael J. Franklin. Concurrency Control and Recovery. In *The Computer Science and Engineering Handbook*, pages 1058–1077. 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, pages 518–529, 1999.

[Gra06] Goetz Graefe. Implementing Sorting in Database Systems. *ACM Comput. Surv.*, 38(3), September 2006.

[HRSD07] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*, pages 389–400, 2007.

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, pages 297–308, 2009.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[JFJT11] Chao Jin, Dan Feng, Hong Jiang, and Lei Tian. A Comprehensive Study on RAID-6 Codes: Horizontal vs. Vertical. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 102–111, July 2011.

IEEE version  ,

UDS campus version  .

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[KSL13] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. Experimental Evaluation of NUMA Effects on Database Management Systems. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 185–204, 2013.

UDS campus version  .

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

UDS campus version  .

[MBNK04] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-conscious Radix-decluster Projections. In *VLDB*, pages 684–695, 2004.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[Moh99] C. Mohan. Repeating History Beyond ARIES. In *VLDB*, pages 1–17, 1999.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *VLDB*, pages 314–325, 1990.

[PH12] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[RAD15] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. In *accepted at PVLDB 9, September 2015*, 2015.

[RDS02] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A Case for Fractured Mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 430–441. VLDB Endowment, 2002.

VLDB version  ,

UDS campus version  .

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3rd edition, 2003.

[SG07] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, 2007.

ACM version  .

[SS16] Jens Dittrich Stefan Schuh, Xiao Chen. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD, to appear*, 2016.

[Vig13] Stratis D. Viglas. Just-in-time Compilation for SQL Query Processing. *Proc. VLDB Endow.*, 6(11), August 2013.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.*, 29(3):55–67, 2000.

[WLO+85] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit Transposed Files. In *VLDB*, pages 448–457, 1985.

[WOS06] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing Bitmap Indices with Efficient Compression. *TODS*, 31(1):1–38, 2006.

Index

CV Jens Dittrich

Jens Dittrich is a Full Professor of Computer Science in the area of Databases, Data Management, and Big Data at Saarland University, Germany. Previous affiliations include U Marburg, SAP AG, and ETH Zurich. He is also associated to CISPA (Center for IT-Security, Privacy and Accountability). He received an Outrageous Ideas and Vision Paper Award at CIDR 2011; a BMBF VIP Grant in 2011; a best paper award at VLDB 2014 (the first ever given to an E&A paper); two CS teaching awards in 2011 and 2013; several presentation awards including a qualification for the interdisciplinary German science slam finals in 2012; and three presentation awards at CIDR (2011, 2013, and 2015). He has been a PC member of prestigious international database conferences such as PVLDB/VLDB, SIGMOD, and ICDE. In addition, he has been an area chair at PVLDB, a group leader at SIGMOD, and an associate editor at VLDBJ.

His research focuses on fast access to big data including in particular: data analytics on large datasets, main-memory databases, database indexing, and reproducability (see https://github.com/uds-datalab/PDBF).

Since 2013 he has been teaching some of his classes on data management as flipped classrooms (aka inverted classrooms). See:

http://datenbankenlernen.de  ,

http://youtube.com/jensdit  .

for a list of freely available videos on database technology in German and English (about 80 videos in German and 80 in English so far).