Patterns in Data Management

—

A Flipped Textbook

Jens Dittrich

October 19, 2017

Imprint

Notice that the youtube videos produced by Jens Dittrich and pointed to in this book are freely available on http://youtube.com/jensdit and may be used for classrooms at no additional costs. The same holds for the slides that were used to produce these videos in the first place. All slides are freely available under a CC-BY-NC license. Checkout my site http://datenbankenlernen.de and my youtube channel http://youtube.com/jensdit for a full list of videos. Or simply follow the pointers in this book. Those sites also contain about 80 additional introductory videos on database concepts in German.

Disclaimer. All material contained in this book has been researched with great care and the author has used his best efforts in preparing this book. Yet, the author and publisher make no warranty of any kind with regard to the concepts, techniques, methods, programs, and software described in this book. In particular, there is no guarantee with respect to correctness or fitness for a particular purpose. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the concepts, techniques, methods, programs, and software contained in this book. So, again, this is a textbook using abstractions (=we leave away details here and there) and generalizations (=we group different techniques under one umbrella by focussing on commonalities and we may leave away details in the descriptions) to explain general concepts. Often details are crucial in making a particular concept work in a specific scenario and setting. Hence, as in all science and engineering, applying technical concepts blindly to a particular scenario and/or software without considering the particular circumstances is a bad idea. So, you are still reading this boring disclaimer? I am wondering: shouldn't you rather be studying the technical content of this book?

book version: October 19, 2017

The author can be reached at:

Prof. Dr. Jens Dittrich
Campus E1 1
66123 Saarbrücken
Germany

jens.dittrich@cs.uni-saarland.de

http://www.infosys.uni-saarland.de
http://datenbankenlernen.de
http://youtube.com/jensdit

ISBN-13: 978-1523853960 (CreateSpace-Assigned)
ISBN-10: 1523853964
BISAC: Computers / Database Management / General

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); cover design by Christine Tophoven (http://www.tophoven-design.de).

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.

[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.

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

Saarbrücken, February 2016

Prof. Dr. Jens Dittrich

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. Marcel Maltry proofread the book. I would like to thank Christine Tophoven for designing the cover of this book. 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, winter 2014/15, and winter 2015/16 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?

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

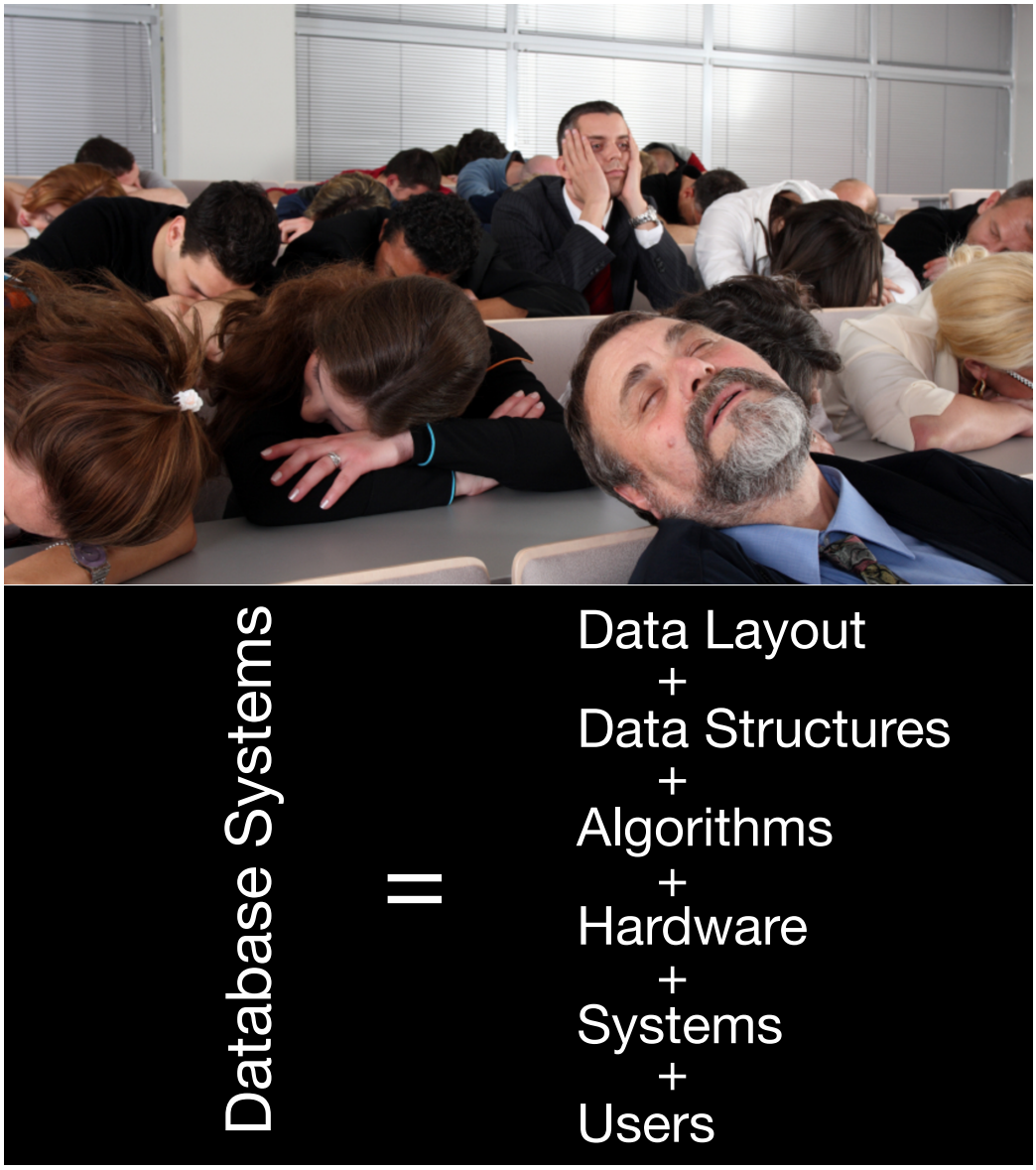


Figure 1: different aspects of database systems

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

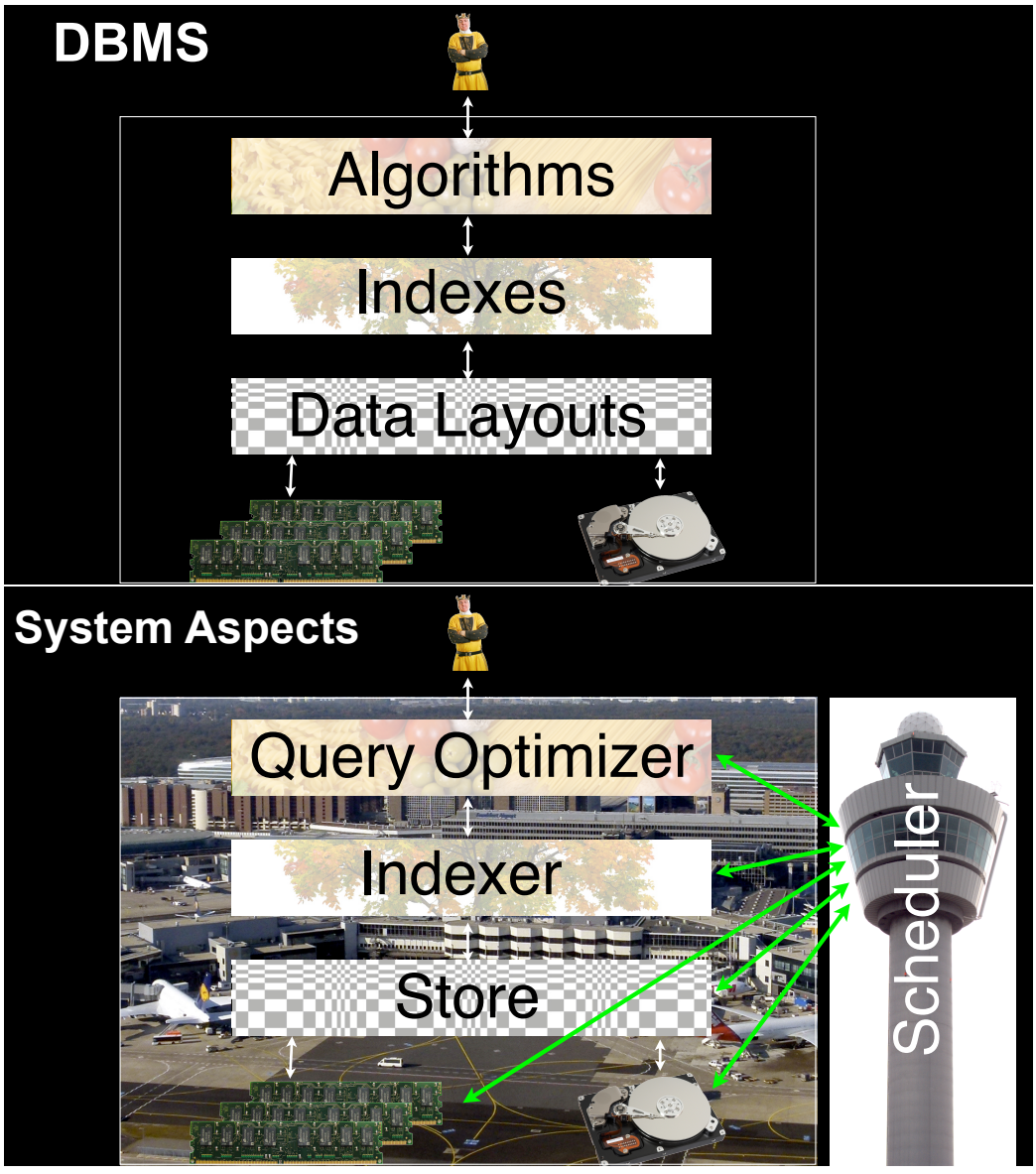


Figure 2: Layers of a database system

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.

Quizzes

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.

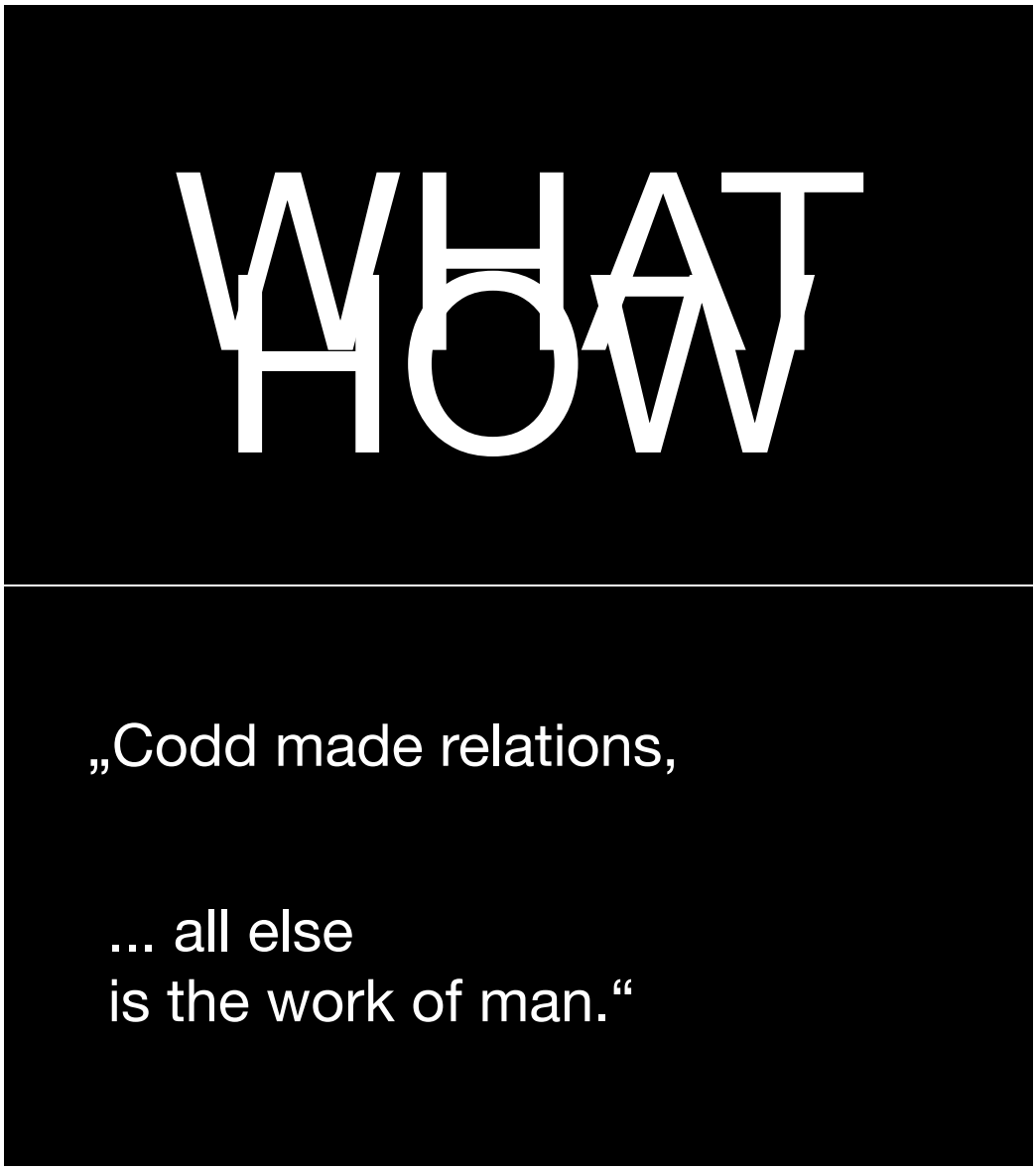


Figure 3: from non-relational to relational database systems

What was a major change introduced by the relational model? **relational model**

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.

What were major developments in database history? **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 techniques 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.

Quizzes

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.

Quizzes

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

What are the properties of ideal computer memory?

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.

What is the core idea of the storage hierarchy?

storage hierarchy

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

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

storage capacity

access time

computing core

The closer a storage layer gets to the computing core, the faster random access, at the same time the smaller the storage capacity.

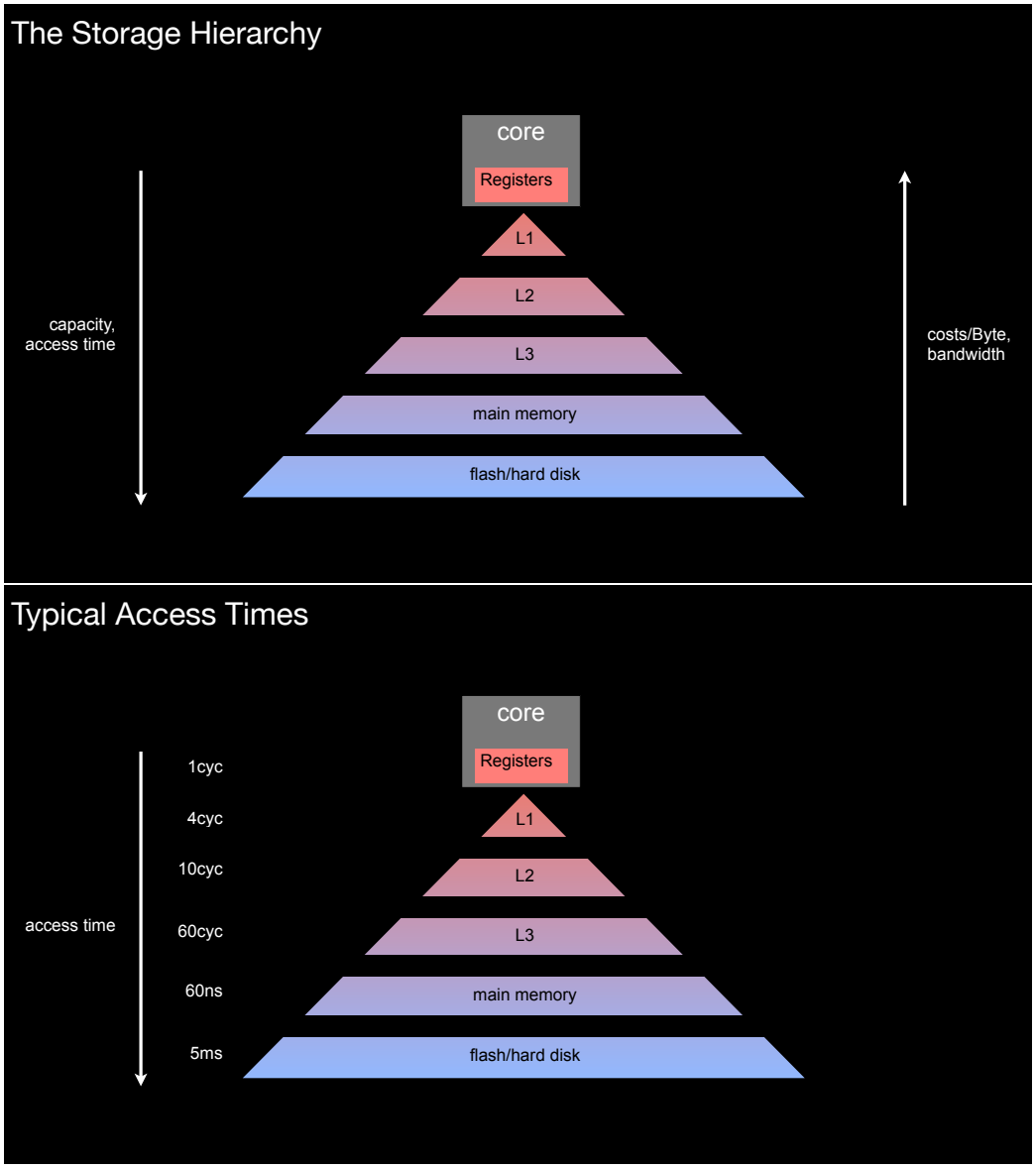


Figure 1.1: A simple storage hierarchy and typical access times

costs

bandwidth

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

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

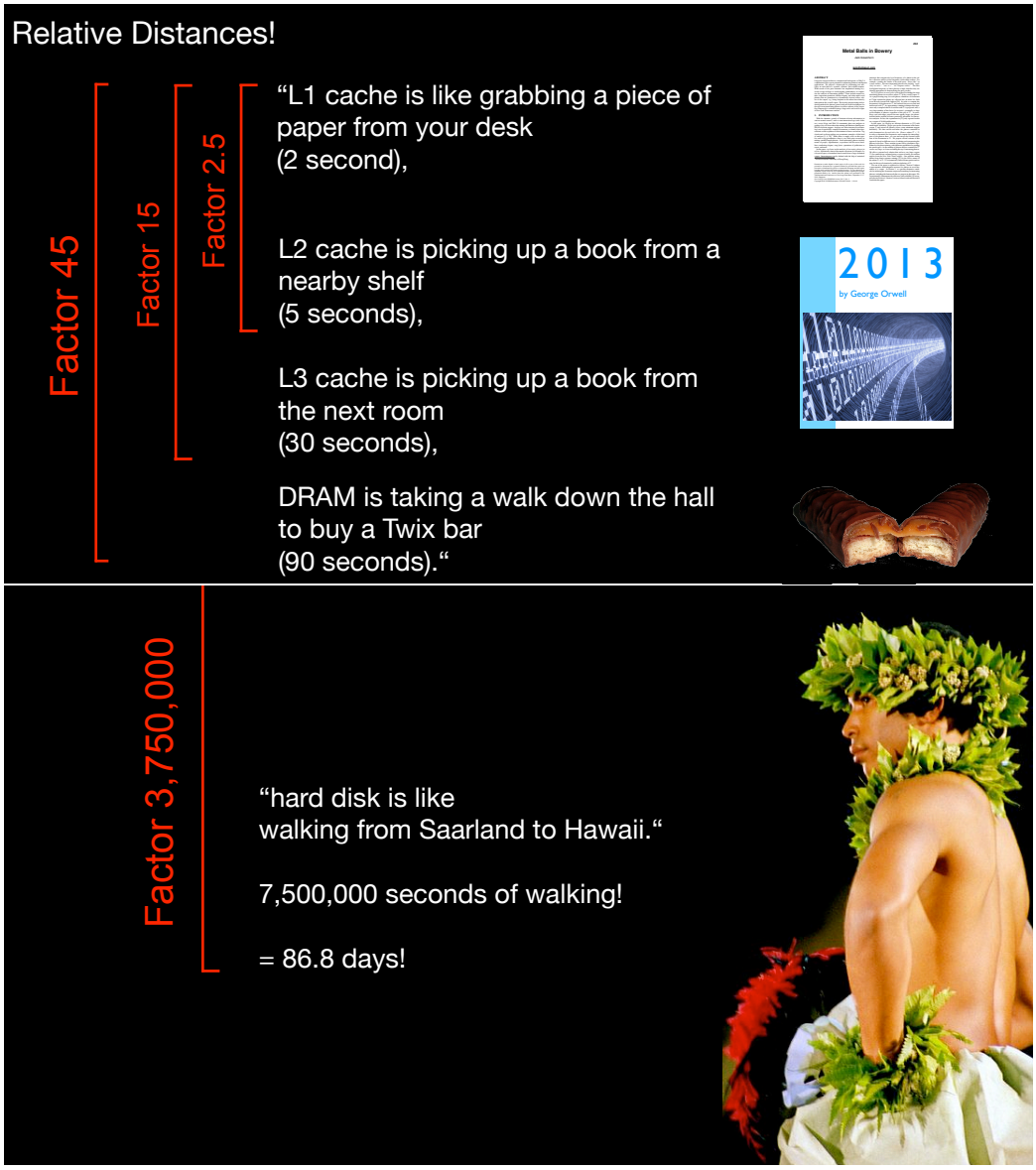


Figure 1.2: Access times translated to a real-life scenario: picking up something from your desk vs walking to Hawaii

RAM has about ∼20ns latency only.

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

storage layer

storage level

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.

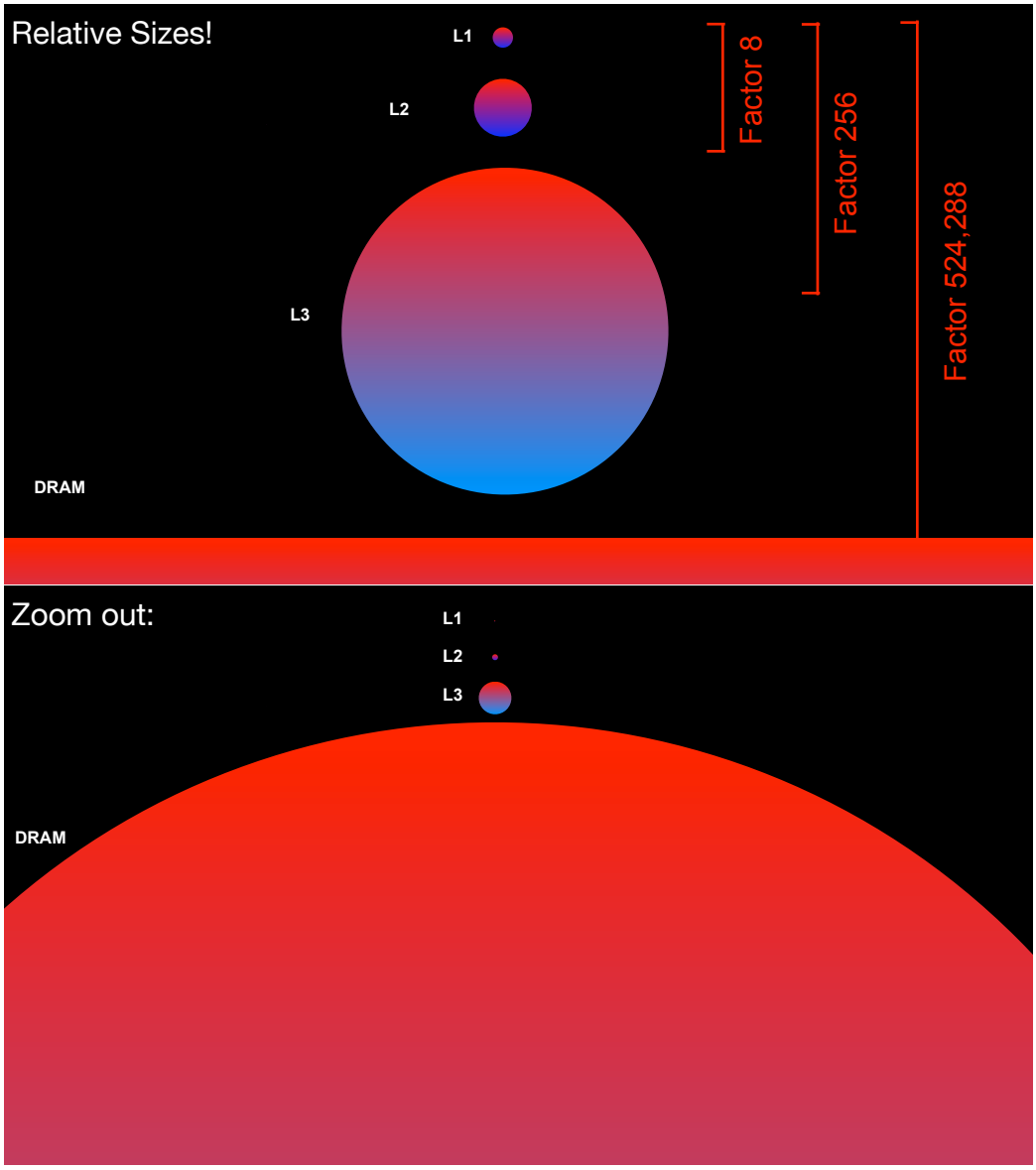


Figure 1.3: Relative Sizes of the different caches in a storage hierarchy: L1–L3 caches vs DRam

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

Localization of data objects, caching of data from lower levels (typically inclusion), implementation 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.

Quizzes

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]

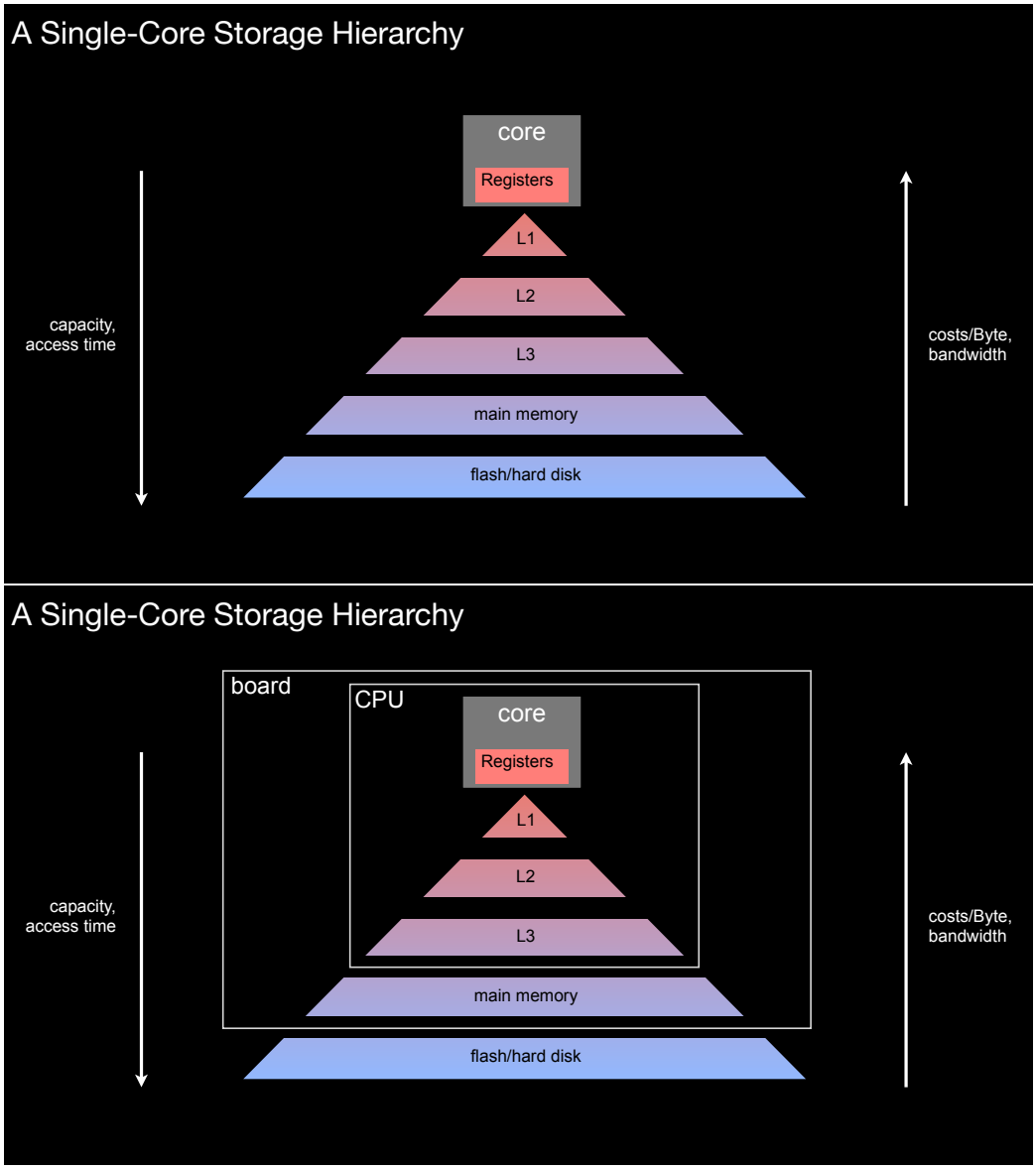


Figure 1.4: A single-core storage hierarchy and how it is mapped to a CPU and the board

Learning Goals and Content Summary

How are the two terms core and CPU related?

core

CPU

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

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

multicore storage hierarchy

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

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

L1

For two major reasons: the access times would go down (for physical reasons) and too

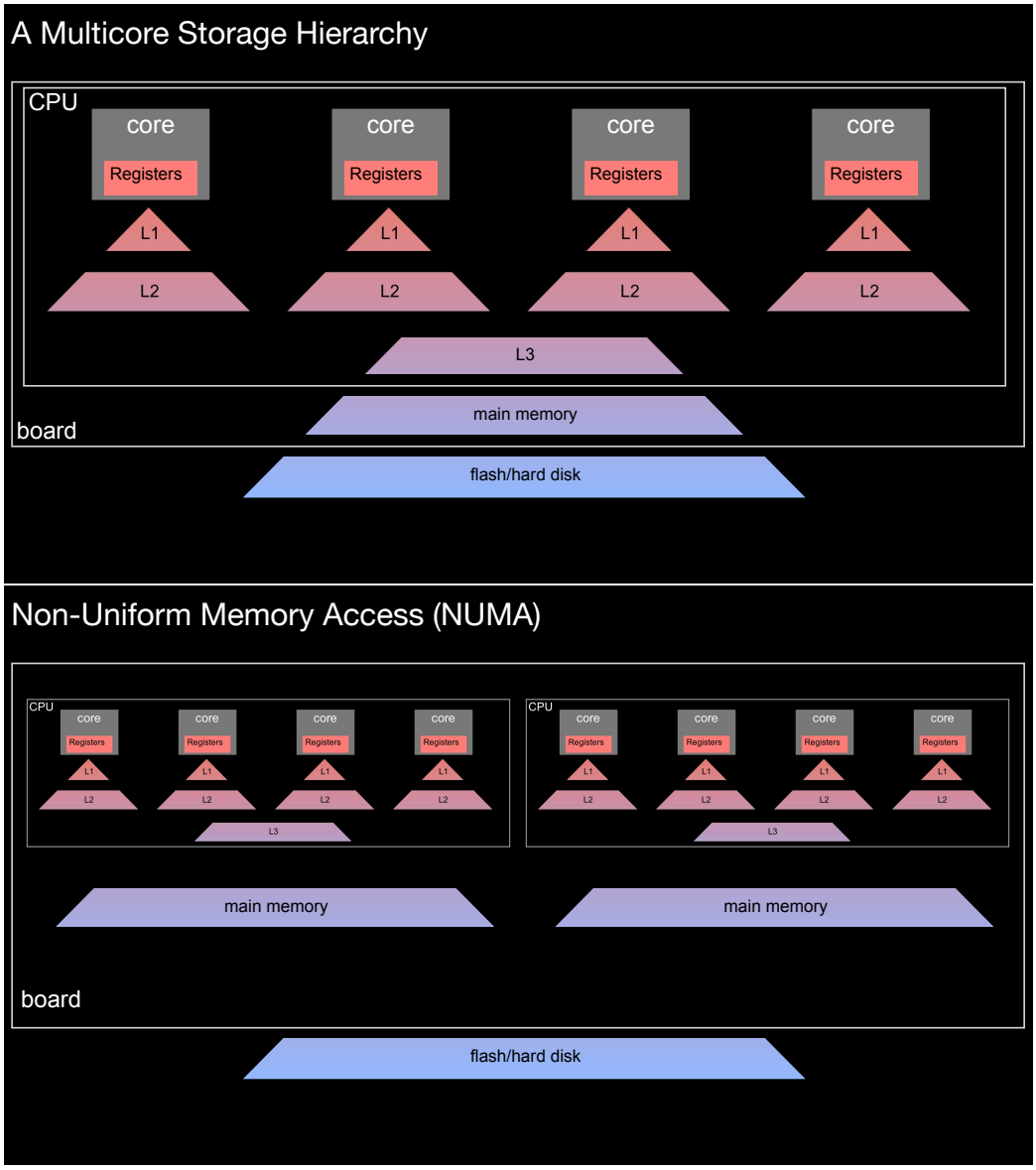


Figure 1.5: A multicore storage hierarchy vs NUMA

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.

Quizzes

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.

Quizzes

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

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.

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

circular sector

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.

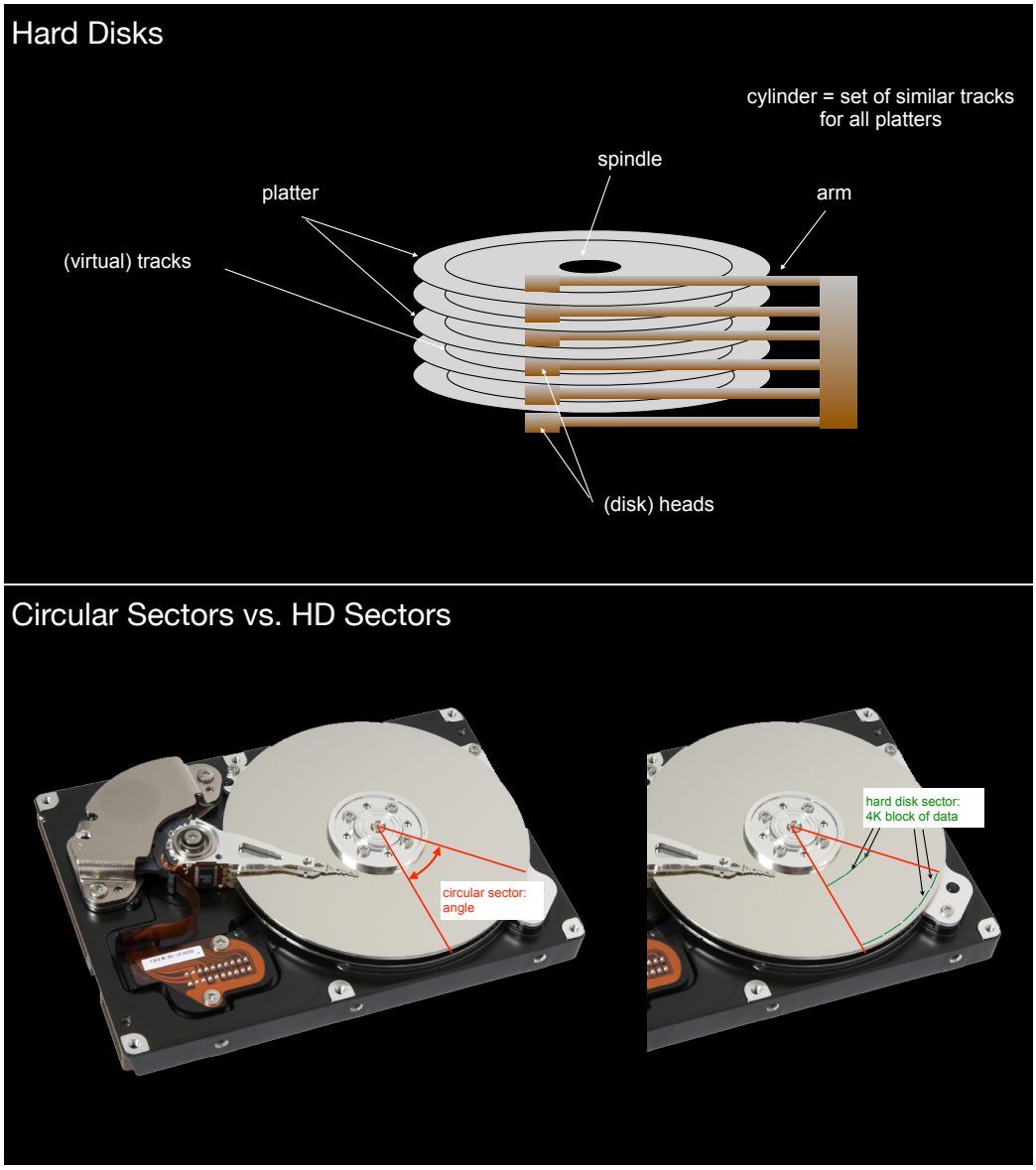


Figure 1.6: The principal architecture of a hard disk; circular vs HD sectors

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 number of HD sectors.

sequential access

What does this imply for 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.

self-correcting block

Where are self-correcting blocks used?

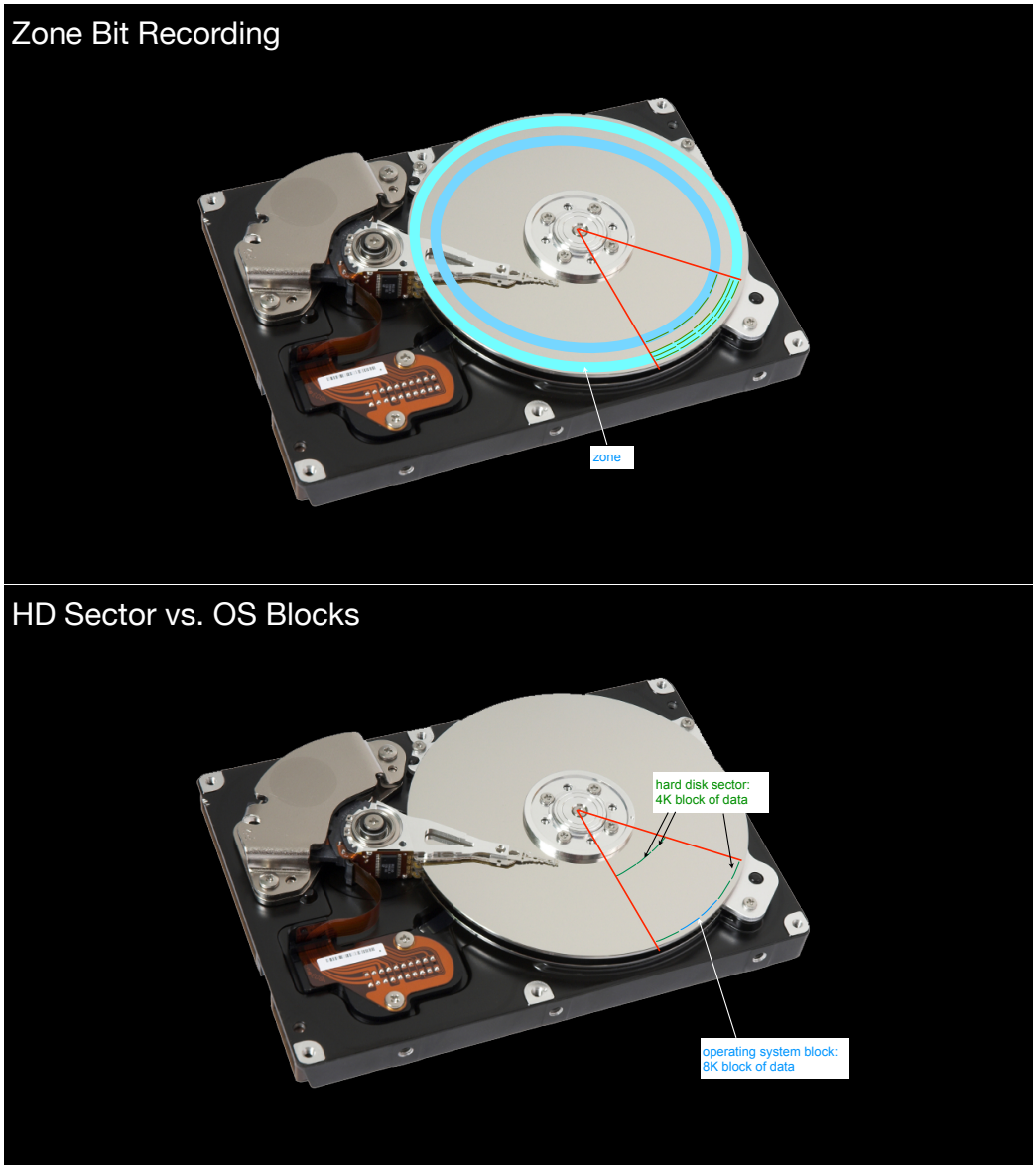


Figure 1.7: Zone bit recording; HD sectors vs operating system blocks

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.

operating systems block

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

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

physical cylinder/head/sector-addressing

What is 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).

Quizzes

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.

1.2.3 Hard Disks: Sequential Versus Random Access

Material

Video:	Original Slides:	Inverted Slides:

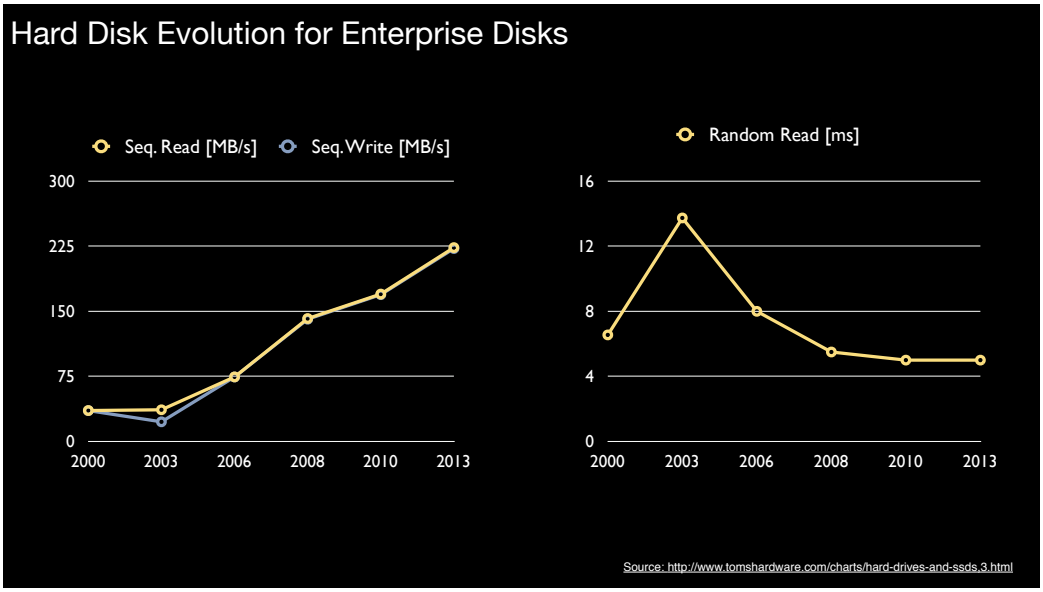


Figure 1.8: Evolution of sequential read and writ bandwidth and random access over time

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).

track skewing

What is 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.

Quizzes

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:

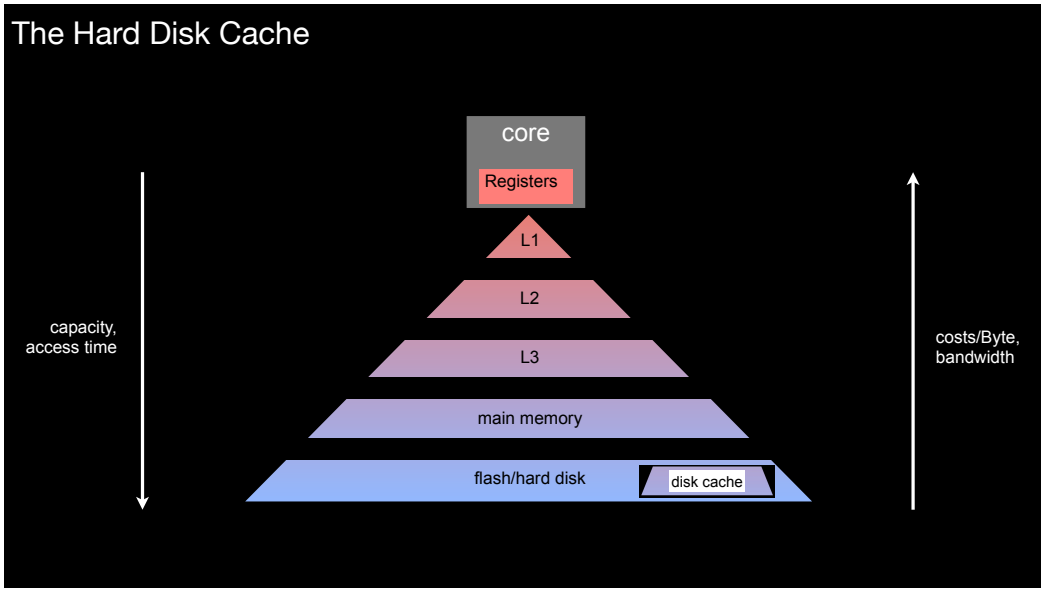


Figure 1.9: The relative position of the hard disk cache within the storage hierarchy

Learning Goals and Content Summary

hard disk cache

What exactly is the hard disk cache?

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

storage hierarchy

How is the hard disk cache related to the 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.

elevator optimization

What is the 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.

Quizzes

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:

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

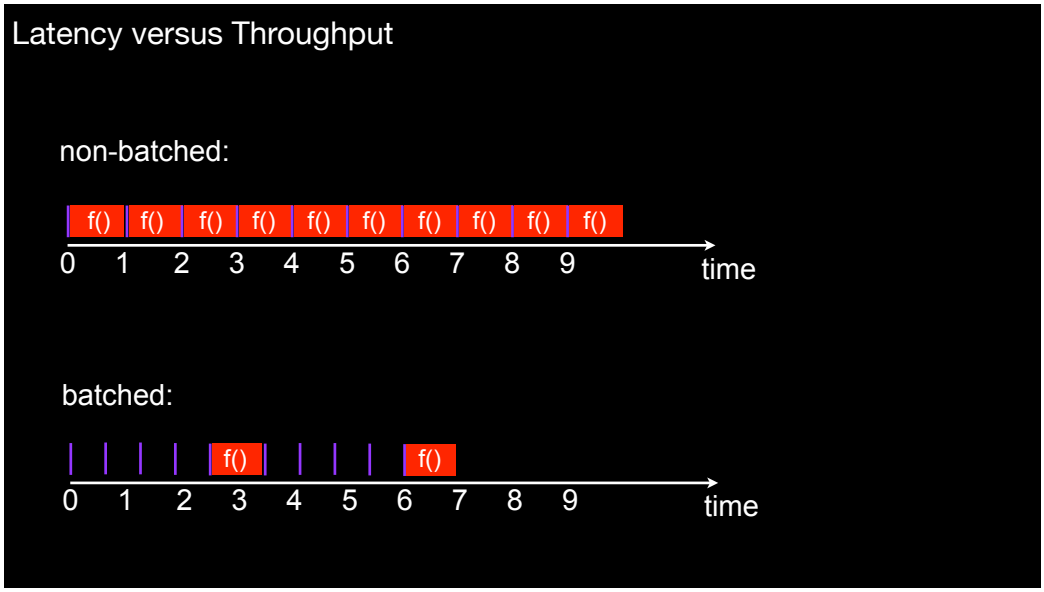


Figure 1.10: Balancing latency and throughput when batching data

"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.

(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.

Quizzes

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
[CLG+94]
[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 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?

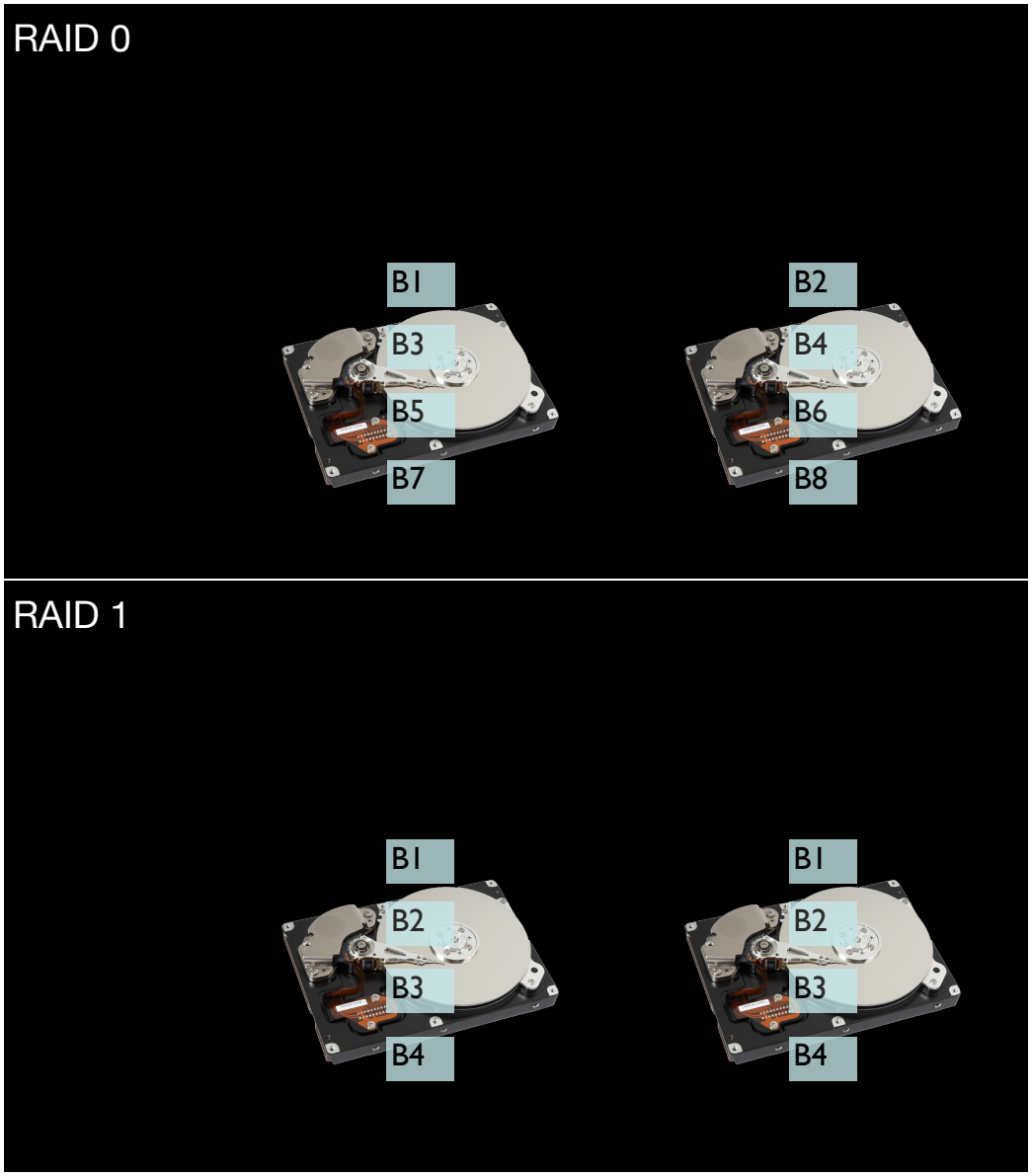


Figure 1.11: RAID 0 vs RAID 1

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 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.

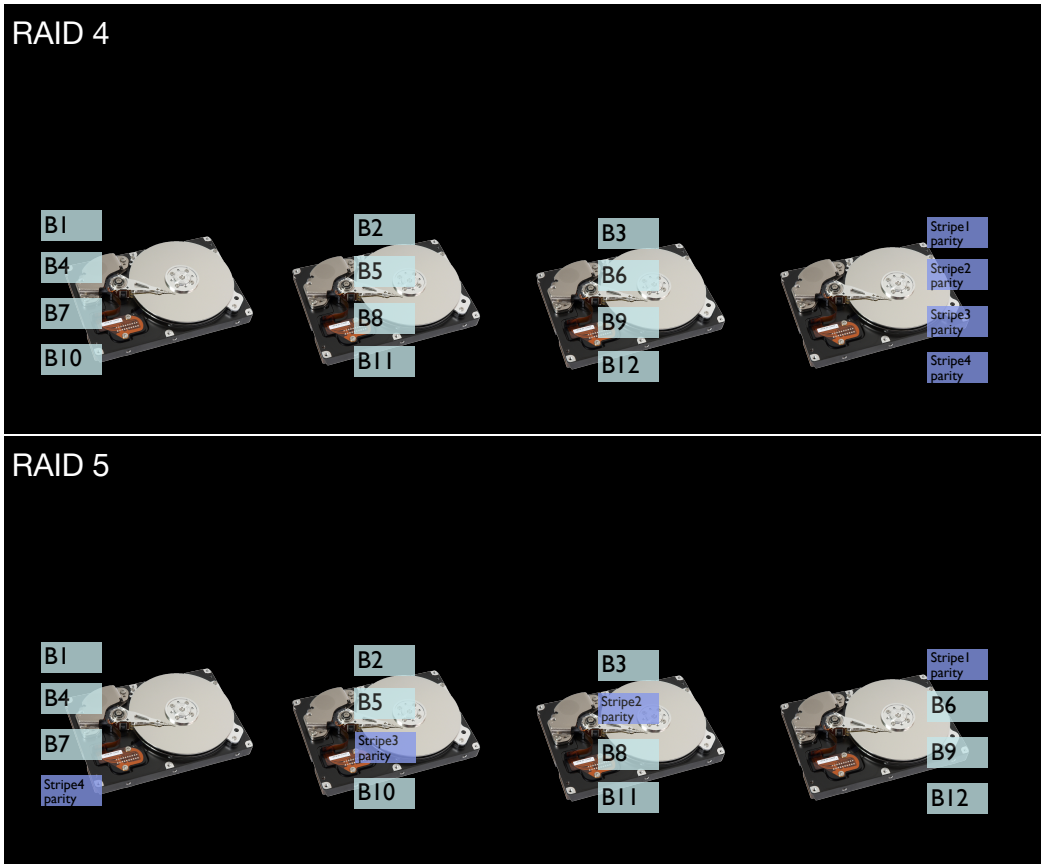


Figure 1.12: RAID 4 vs RAID 5

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.

Quizzes

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

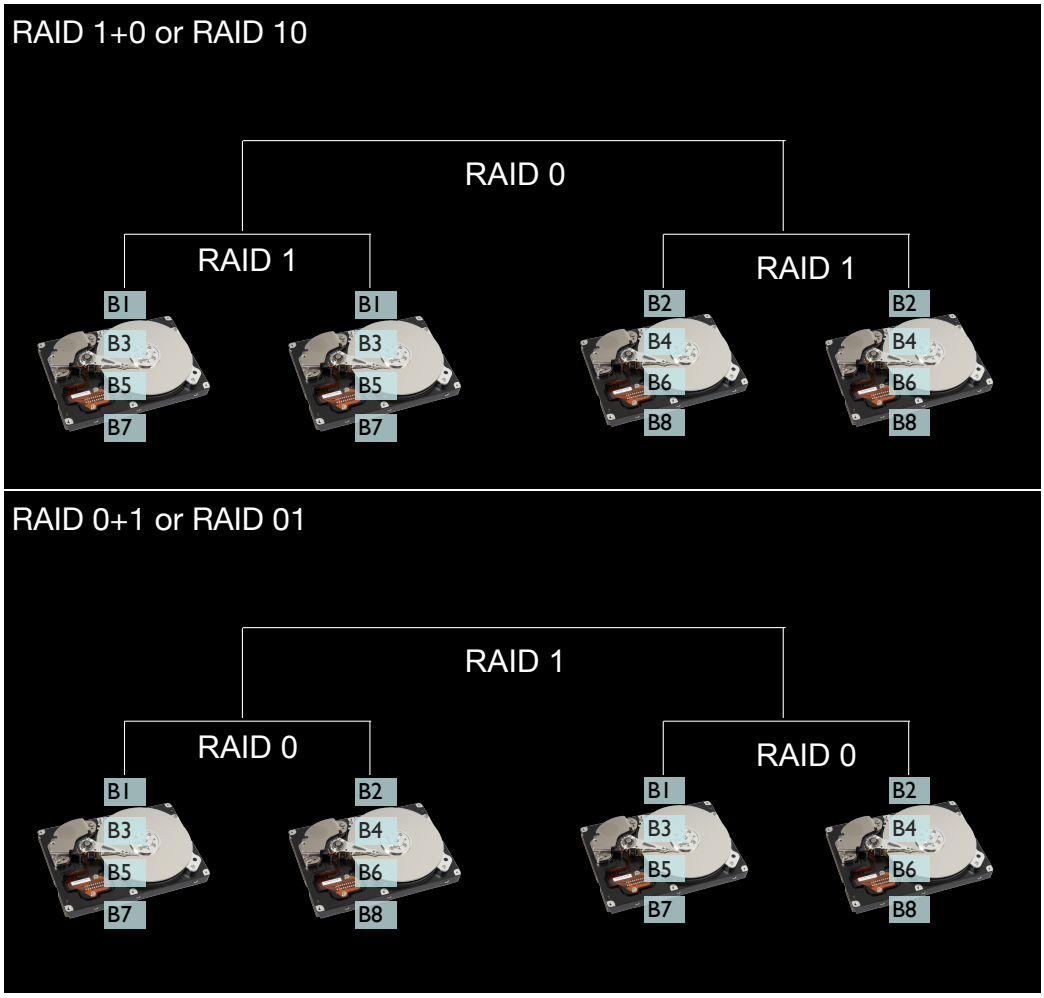


Figure 1.13: RAID 10 (=1+0) vs RAID 01 (=0+1), Note: nested RAID-levels are read from bottom to top

nested RAID

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

Quizzes

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

Quizzes

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

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).

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.

SSD

What is an 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.

hard disk

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

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.

block

In an SSD, what are blocks and superblocks?

superblock

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.

And which random access time? **random access time**

About 0.05 ms

Quizzes

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.

Quizzes

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 simultaneously.

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

spatial locality

And what is spatial locality then?

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.

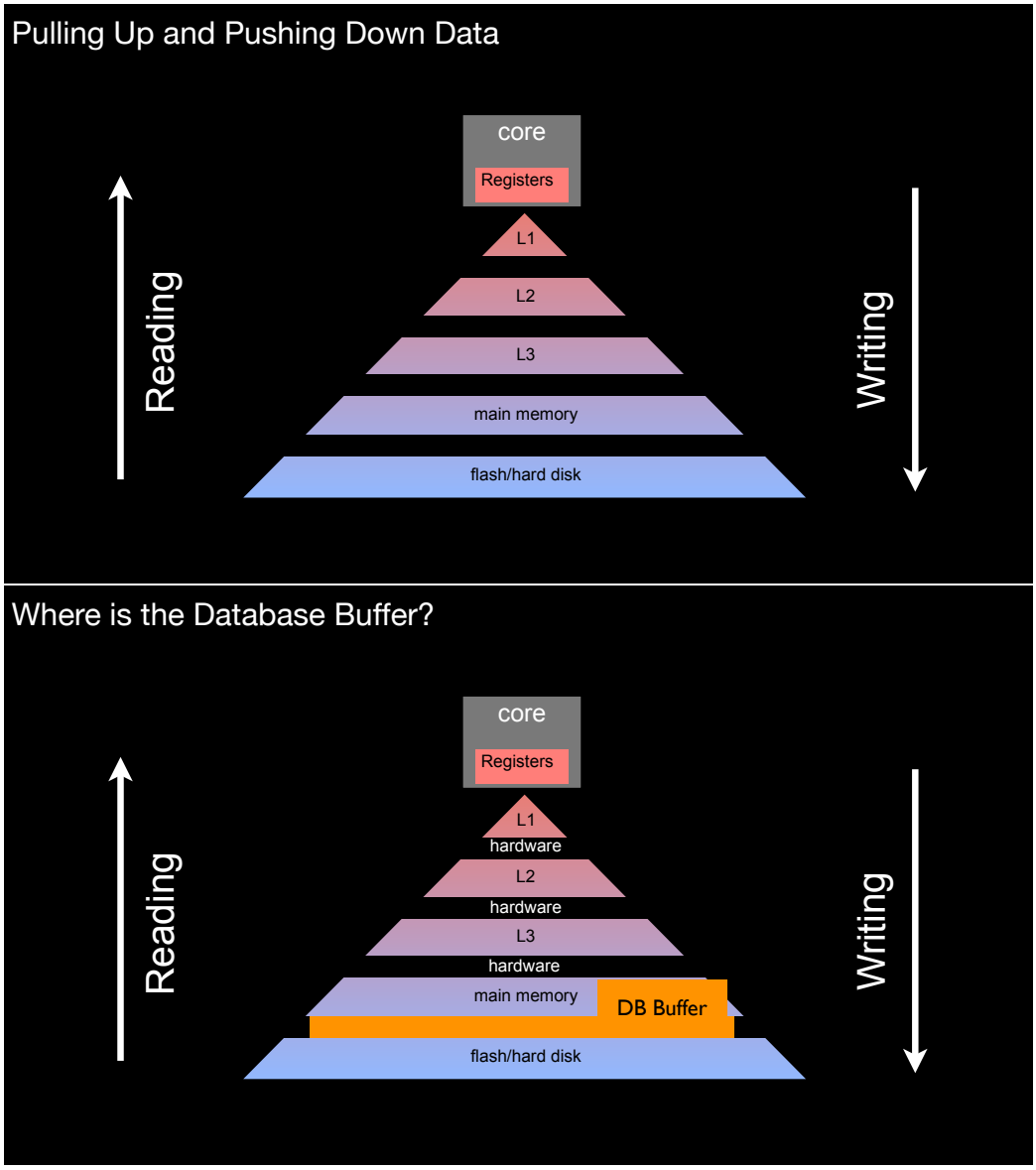


Figure 1.15: Reading data is equivalent to pulling up data in the storage hierarchy. Writing data is equivalent to pushing down data in the storage hierarchy. What is the relative position of the DB-buffer?

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.

Quizzes

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

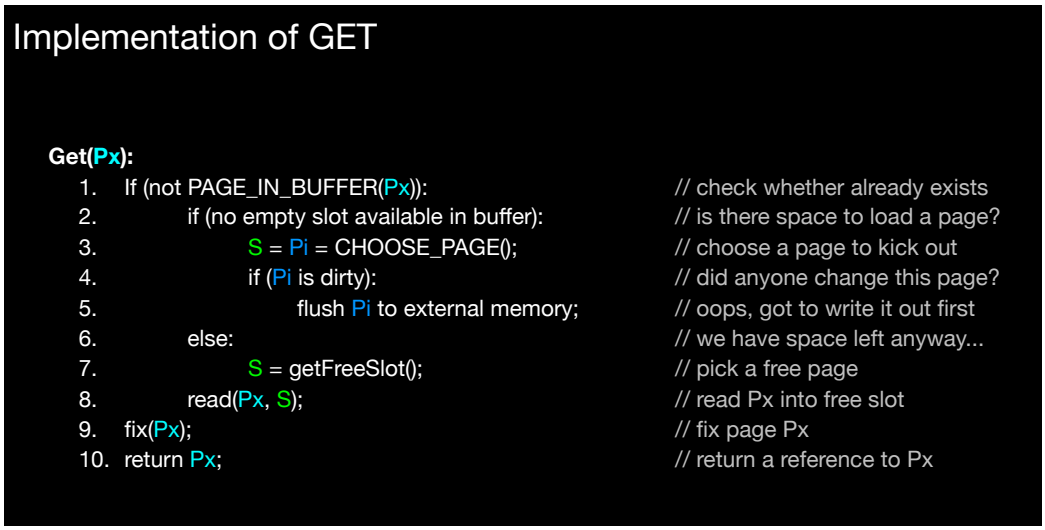


Figure 1.16: The implementation of the get method in the database buffer

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.

Quizzes

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.

Quizzes

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.

Quizzes

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.

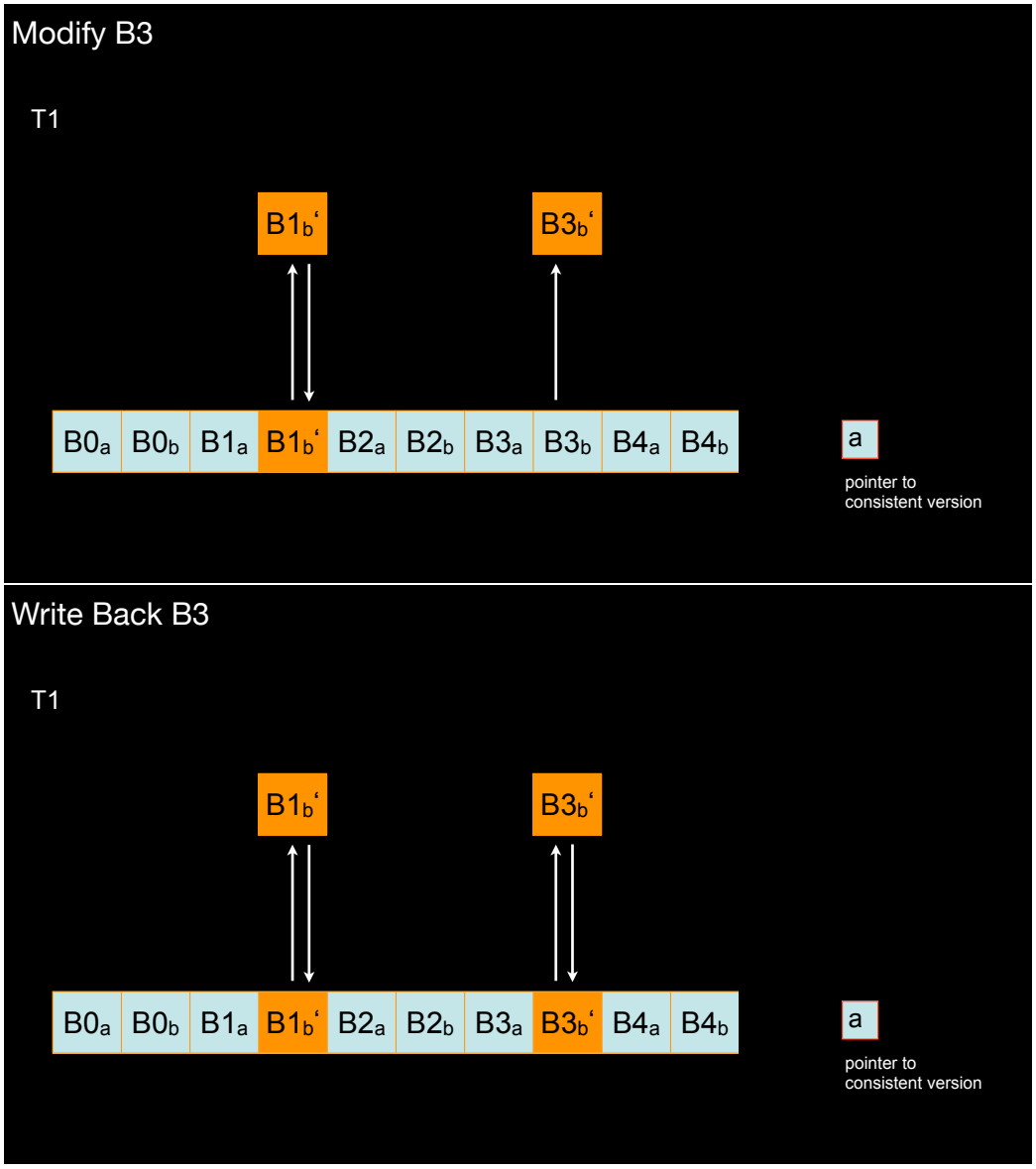


Figure 1.17: Twin block: modifying version b of block B3 to B3' and writing it back to storage.

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

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

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.

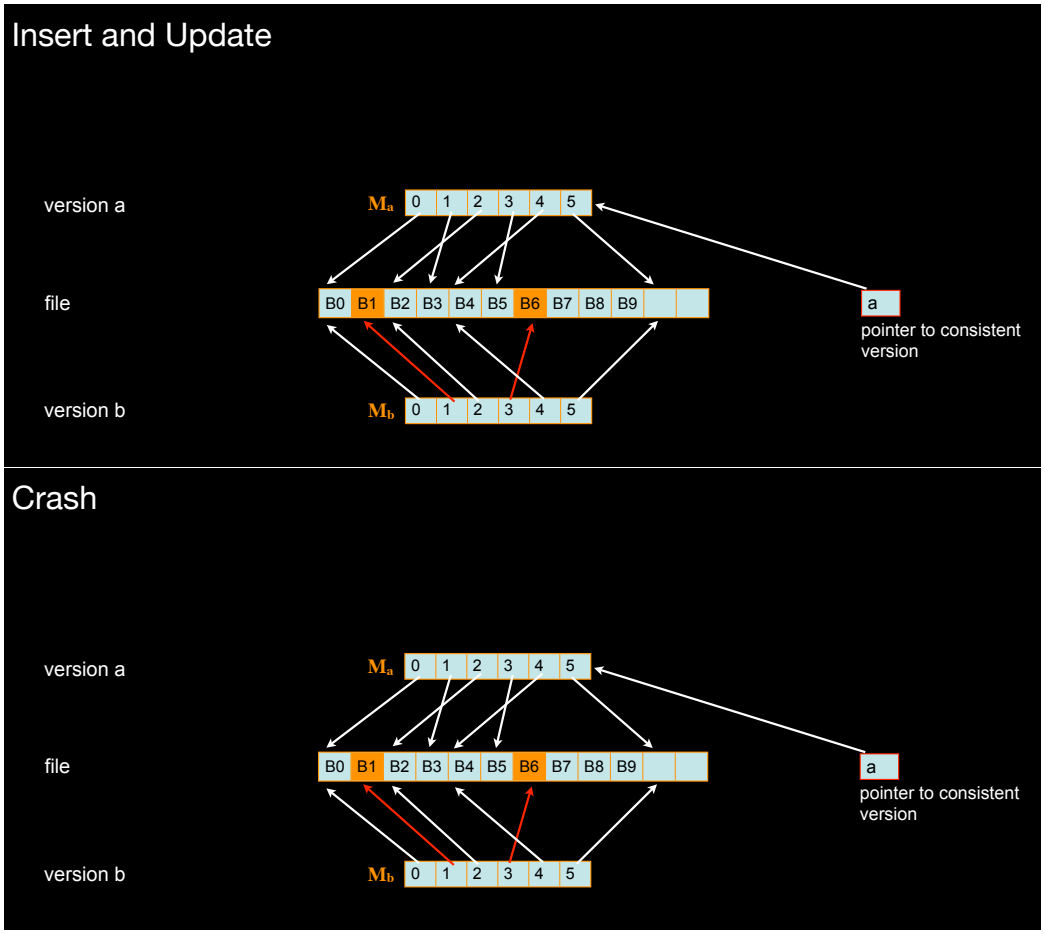


Figure 1.18: Shadow paging: inserting and updating data vs handling a crash

Quizzes

1. In shadow storage we keep
   (a) two versions of each modified block
   (b) two versions of each block
   (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:

Copy On Write Pattern

COW

Learning Goals and Content Summary

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.

Quizzes

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.

Quizzes

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:

Learning Goals and Content Summary

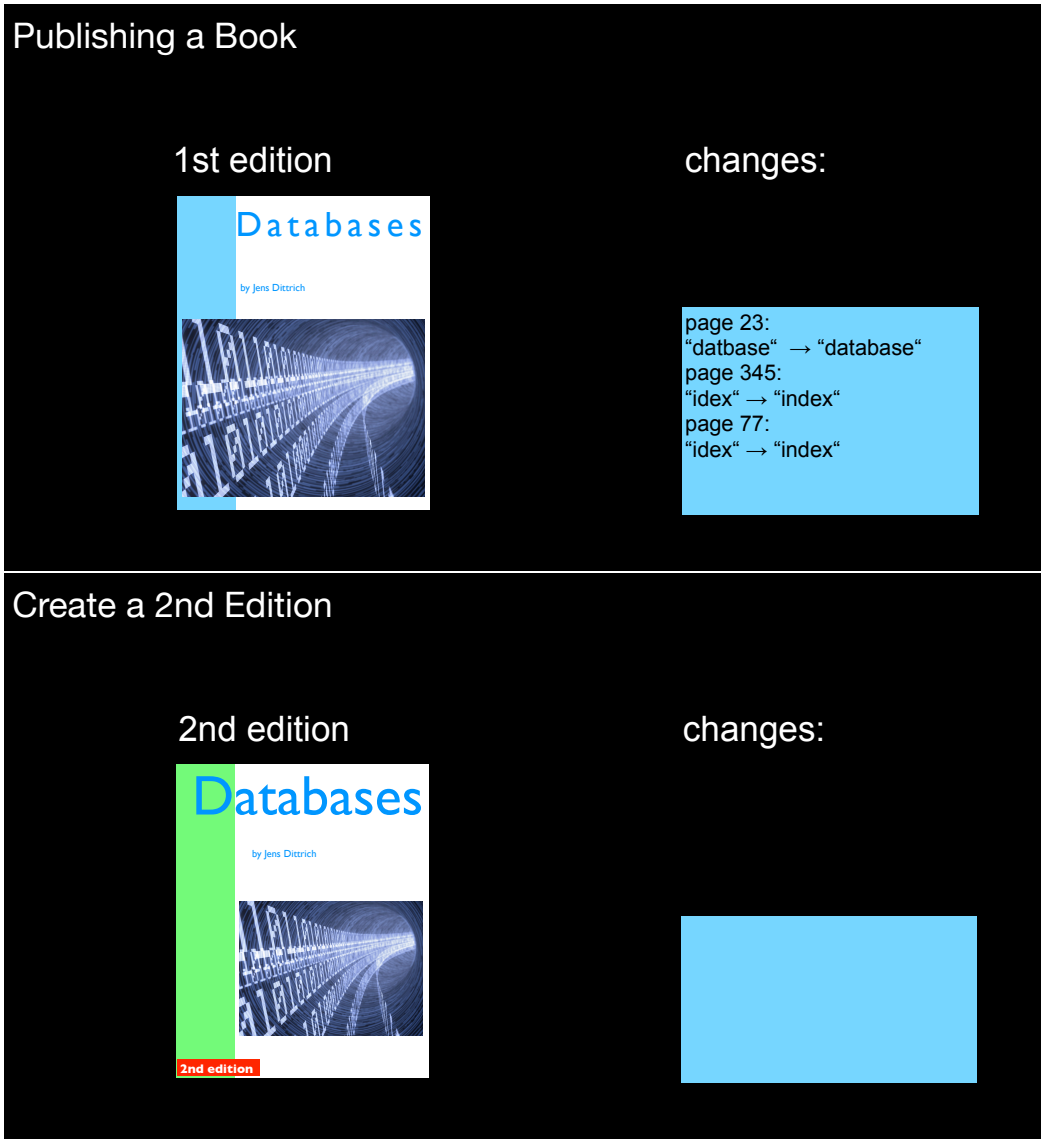


Figure 1.19: Publishing a 1st edition of a book and collecting changes vs eventually creating a second edition

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

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

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*.

To which other patterns does this relate?

pattern

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 file corresponds to an incremental backup. The “editions” (or files) may be considered

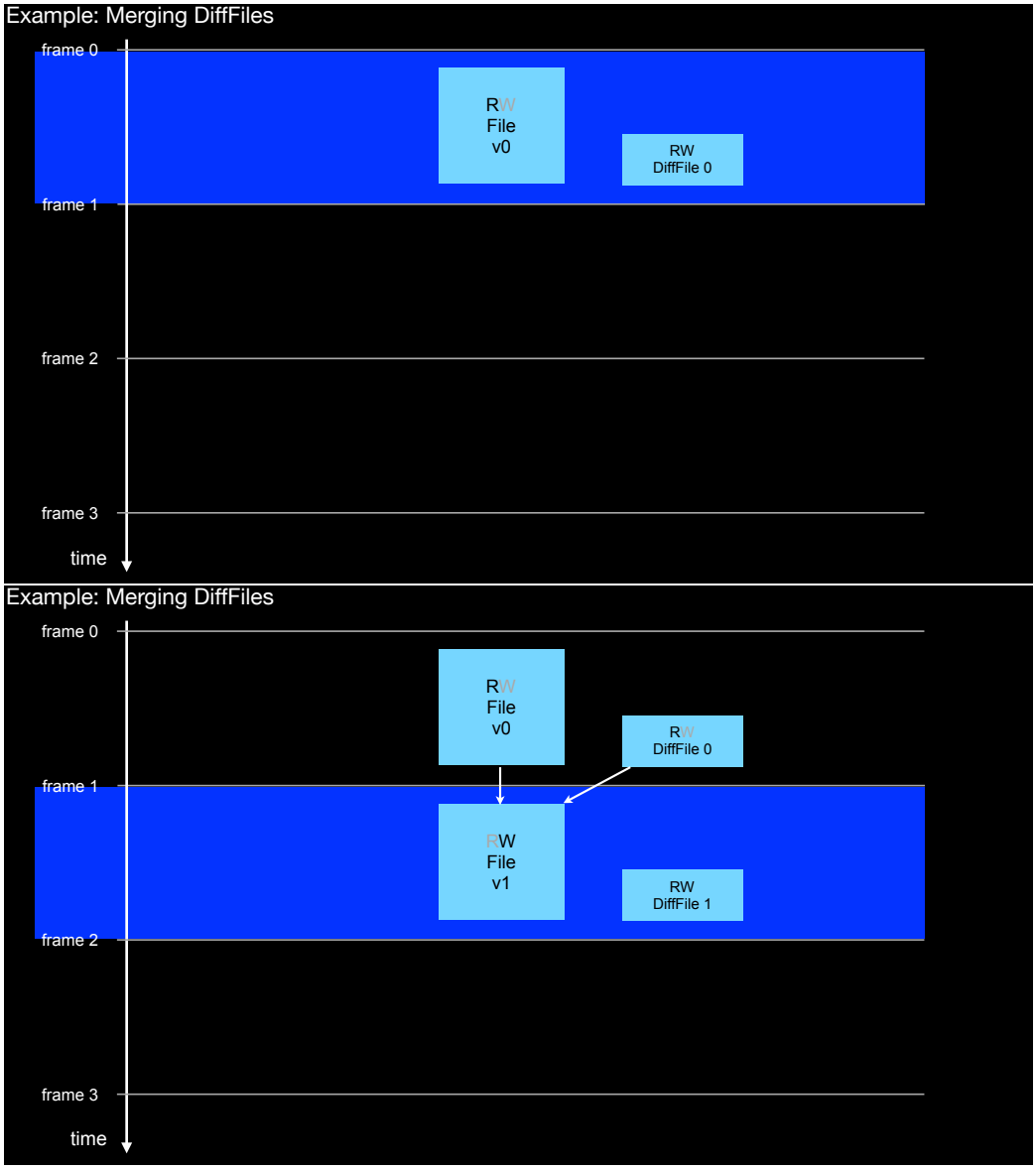


Figure 1.21: Different phases in the merge process: the situation before starting the merge (a read-only file v0 and its DiffFile 0) vs the situation during the merge (the DiffFile 0 is set to read-only as well. Both the read-only file v0 and the DiffFile 0 are merged into a new file v1. Concurrently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

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.

read-only DB

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

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 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.

Quizzes

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

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 corresponds 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.

Quizzes

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

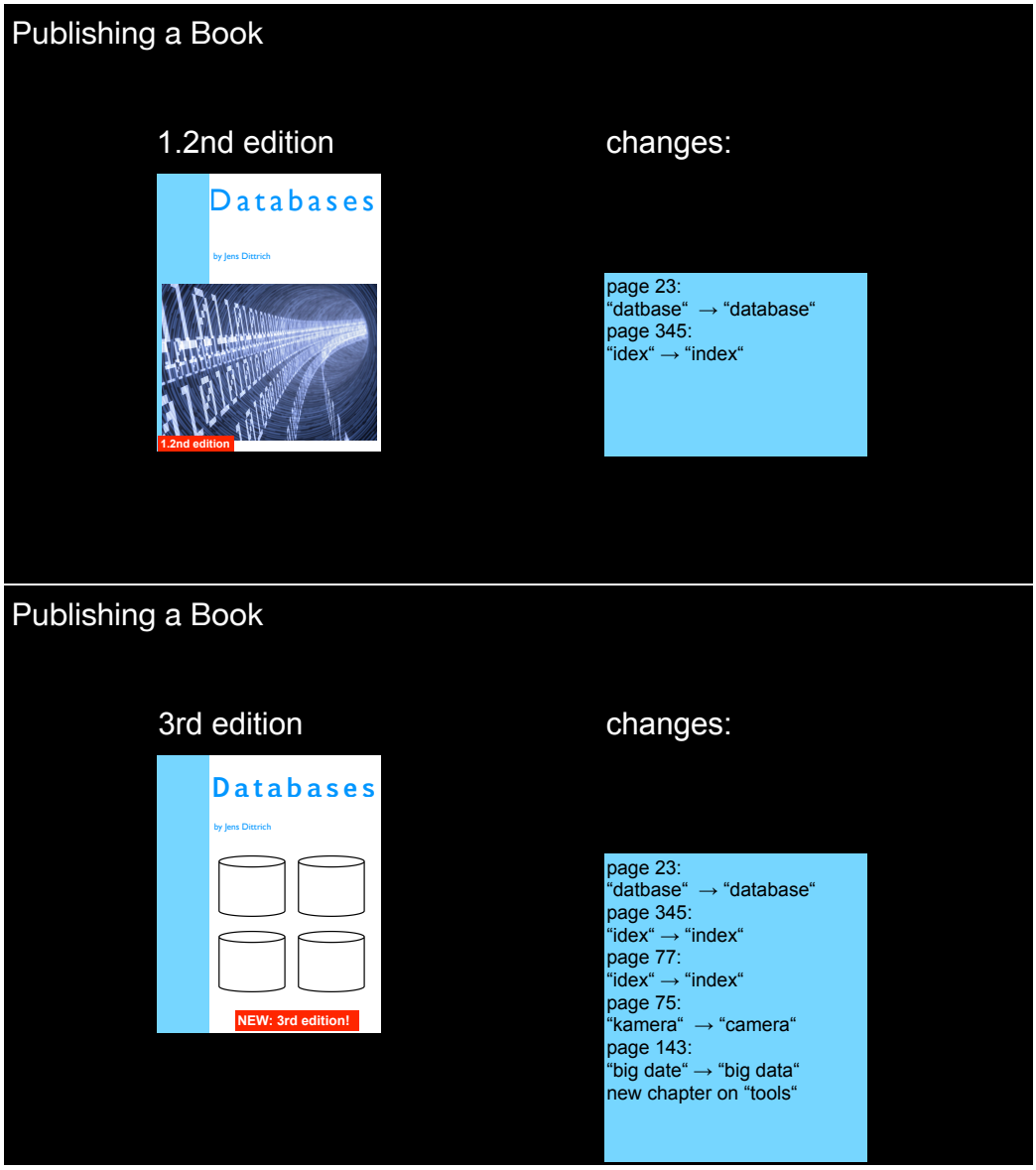


Figure 1.22: Publishing a book using logged writes: changes are not merely collected but every change is also applied to create a new (sub-)edition.

(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:

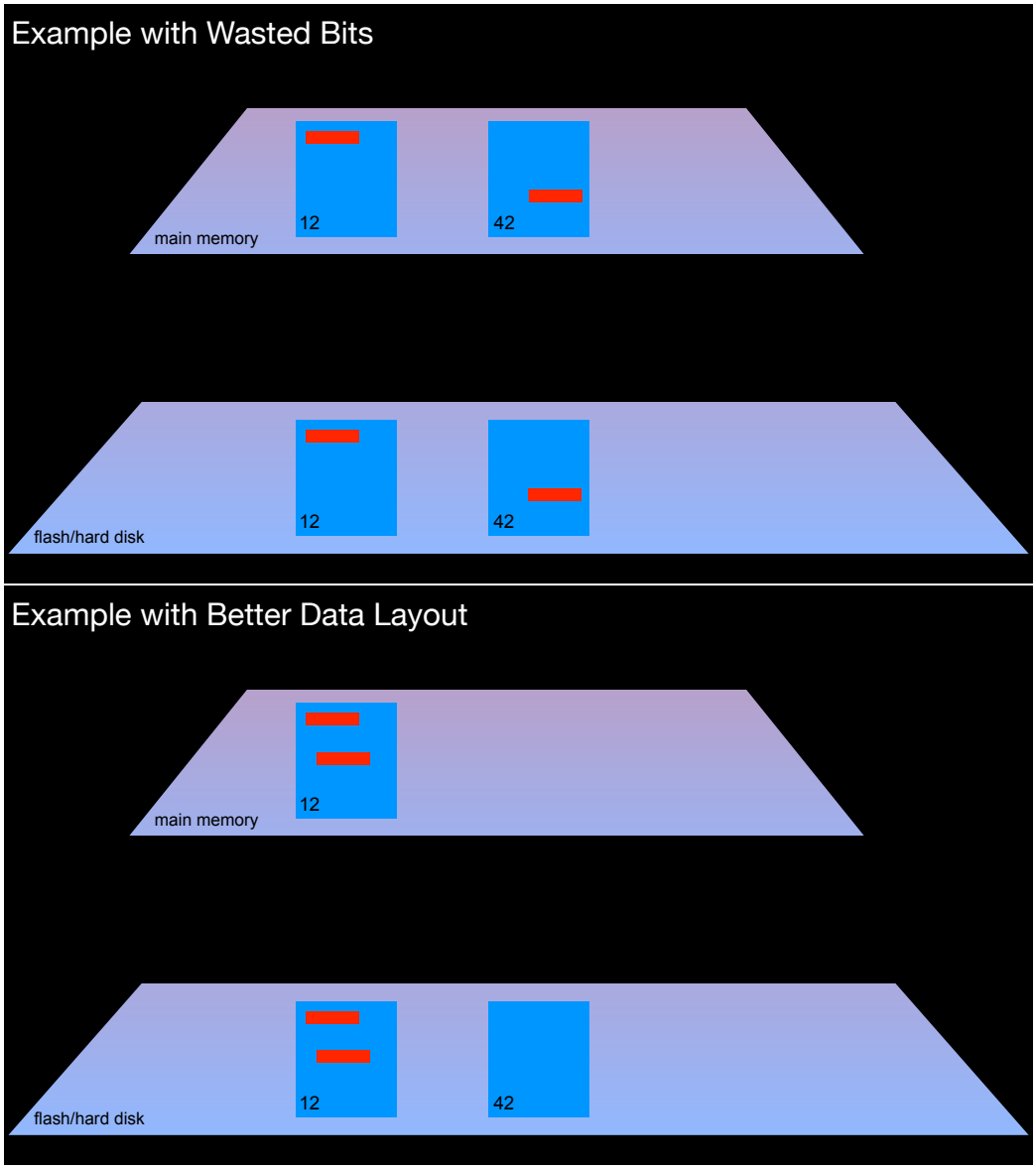


Figure 1.23: Wasting bits by distributing hot data over different pages vs keeping hot data on the same page

Learning Goals and Content Summary

wasting bits

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

Quizzes

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

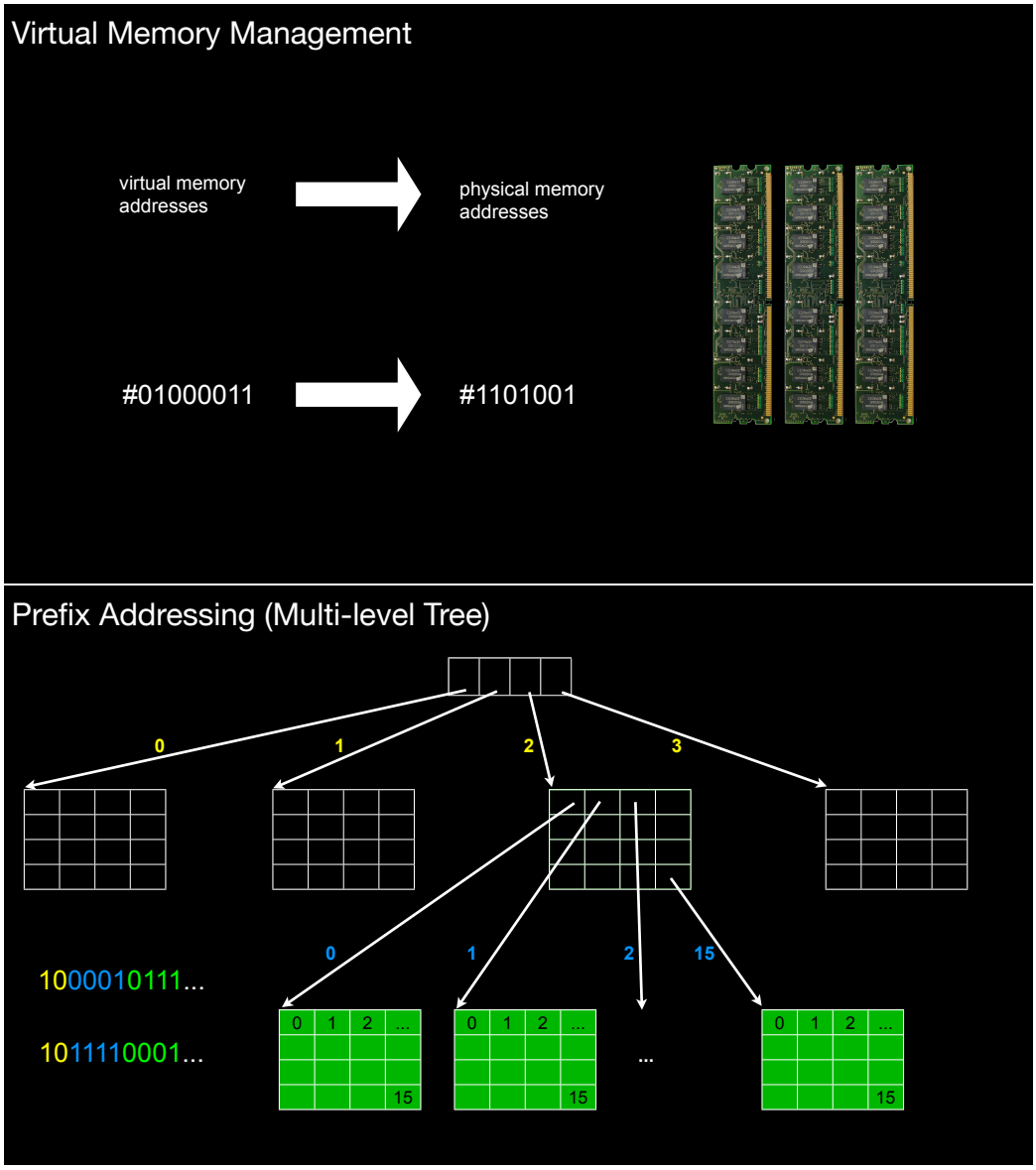


Figure 1.24: Virtual memory management maps virtual address to physical address. This mapping is often implemented using a (multi-level) prefix tree.

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.

What is the role of the page table?

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.

What is prefix addressing?

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

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.

offset

What exactly is an 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.

Quizzes

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?

   (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.

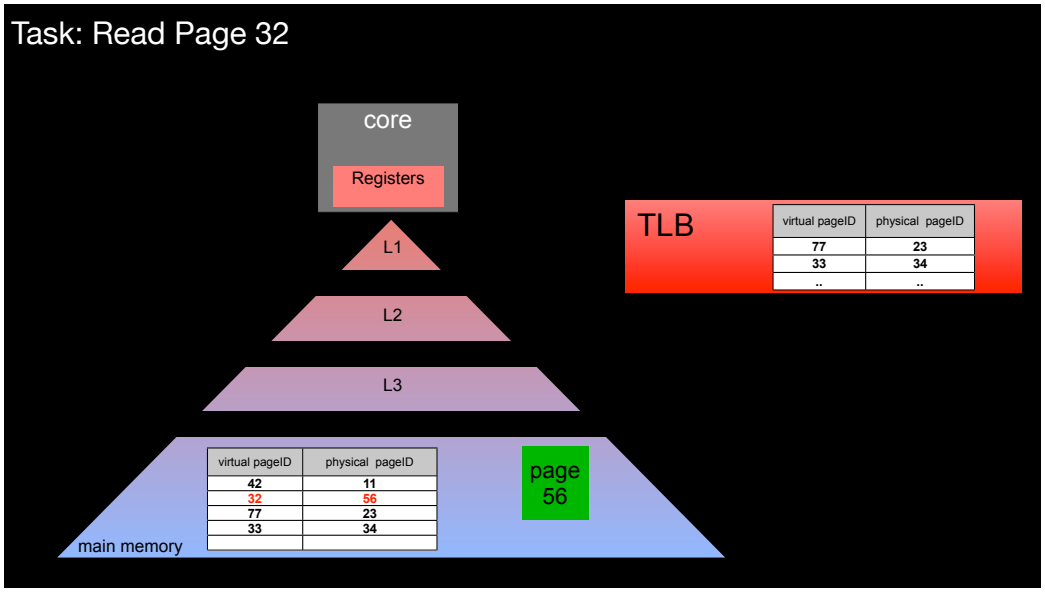


Figure 1.25: Translation lookaside buffer (TLB) and TLB-misses

Quizzes

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?

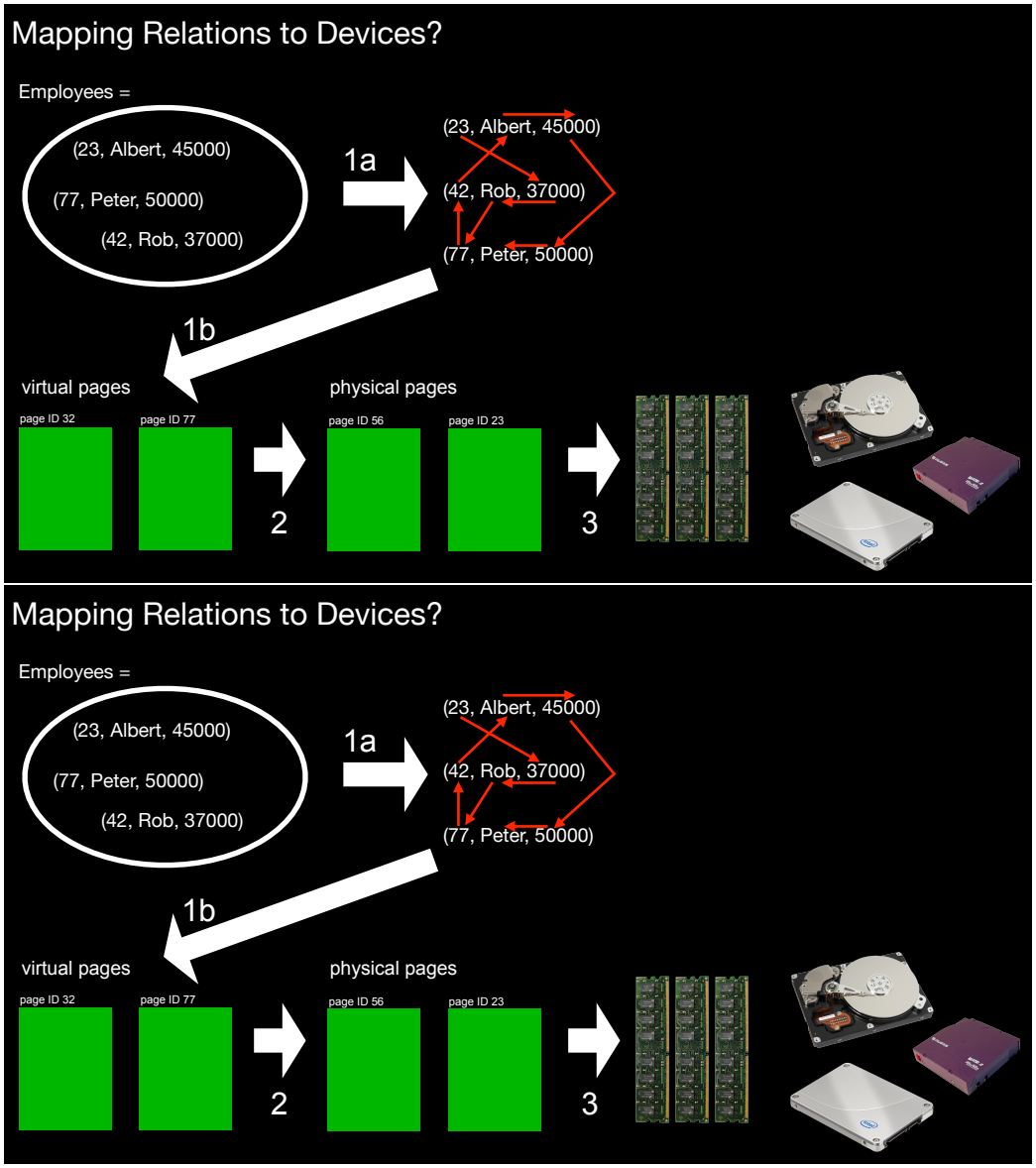


Figure 2.1: The different mapping steps required when mapping two-dimensional relations to storage devices.

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.

Quizzes

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

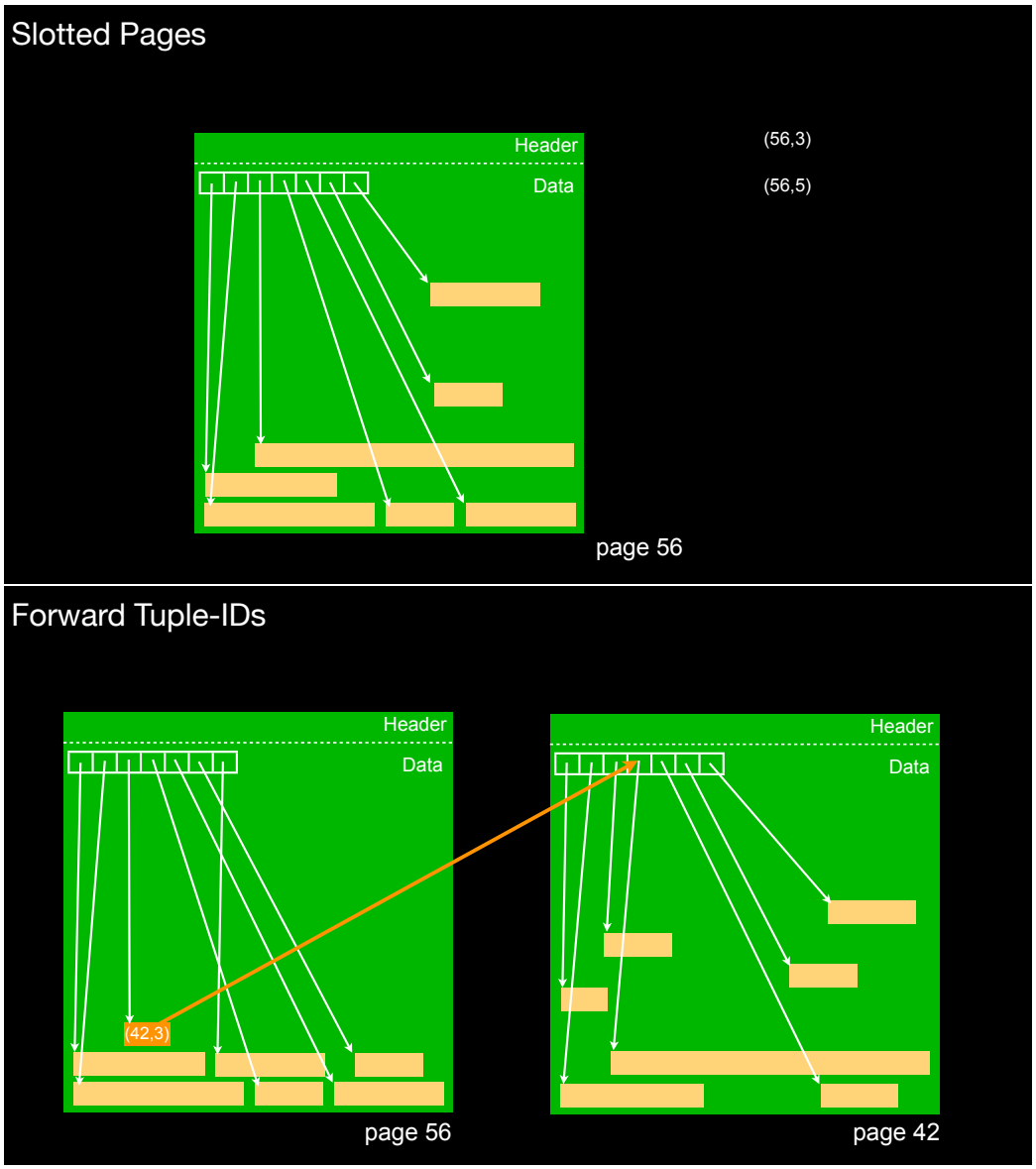


Figure 2.2: Slotted pages and Forward Tuple-IDs

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.

Quizzes

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?

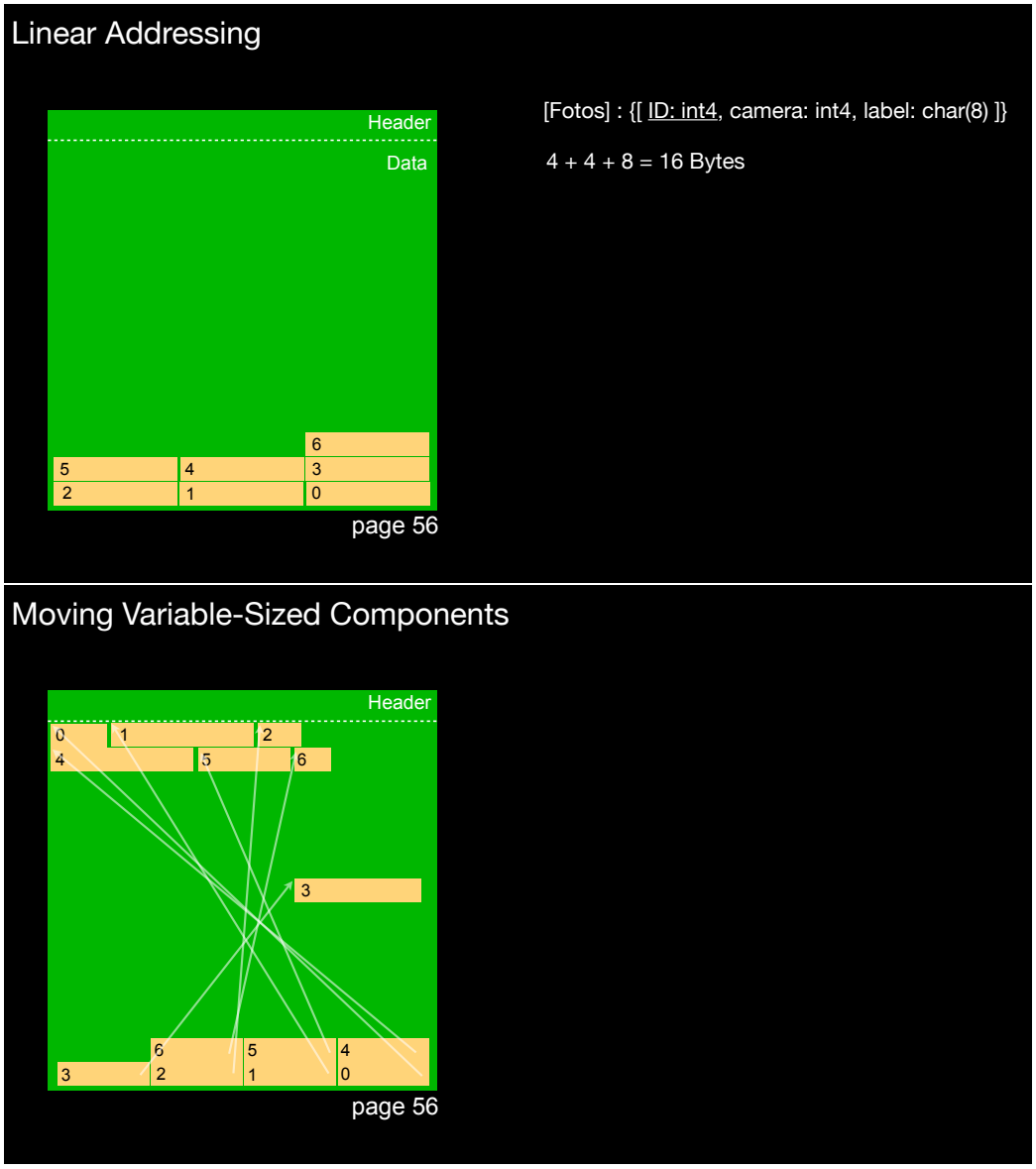


Figure 2.3: Linear Addressing and Variable-Sized Components

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 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.

Quizzes

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.

free space

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

metadata

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

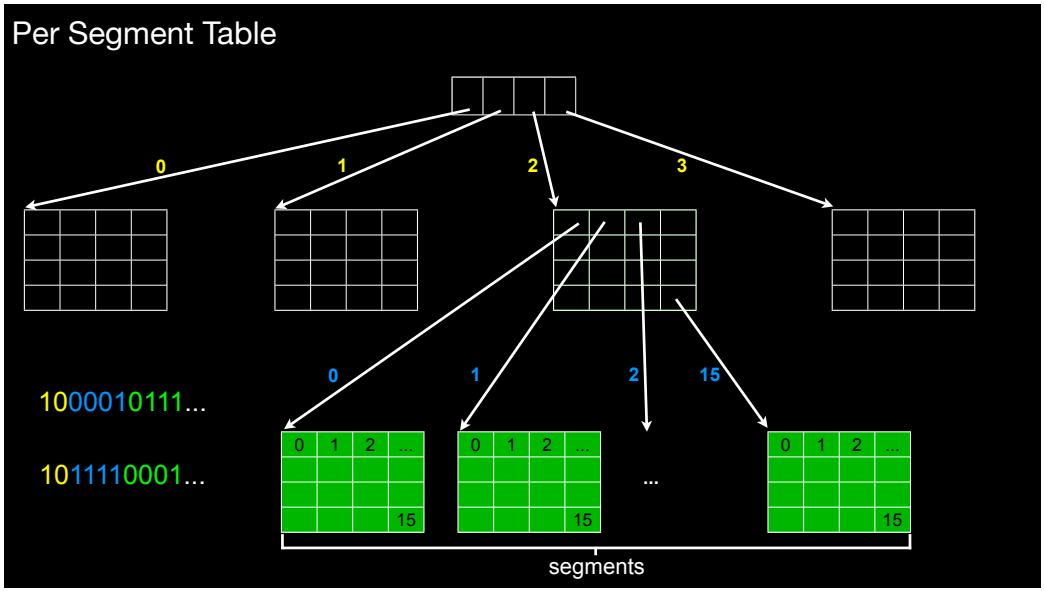


Figure 2.4: Per segment prefix table as a tree

	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

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.

catalog

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

Quizzes

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:

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

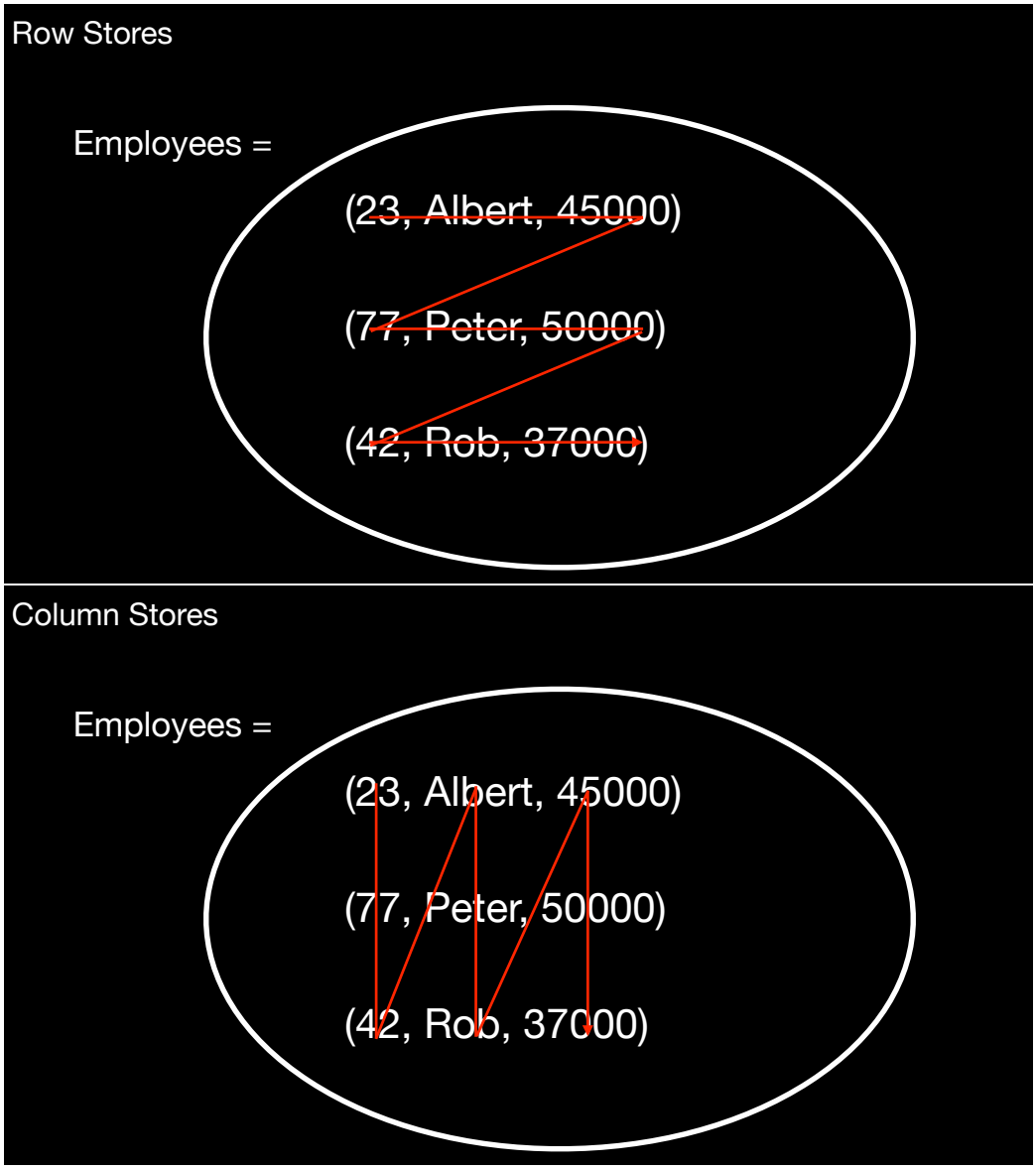


Figure 2.5: Row store vs column store: the major difference is the linearization order of attribute values.

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***.

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.

Quizzes

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

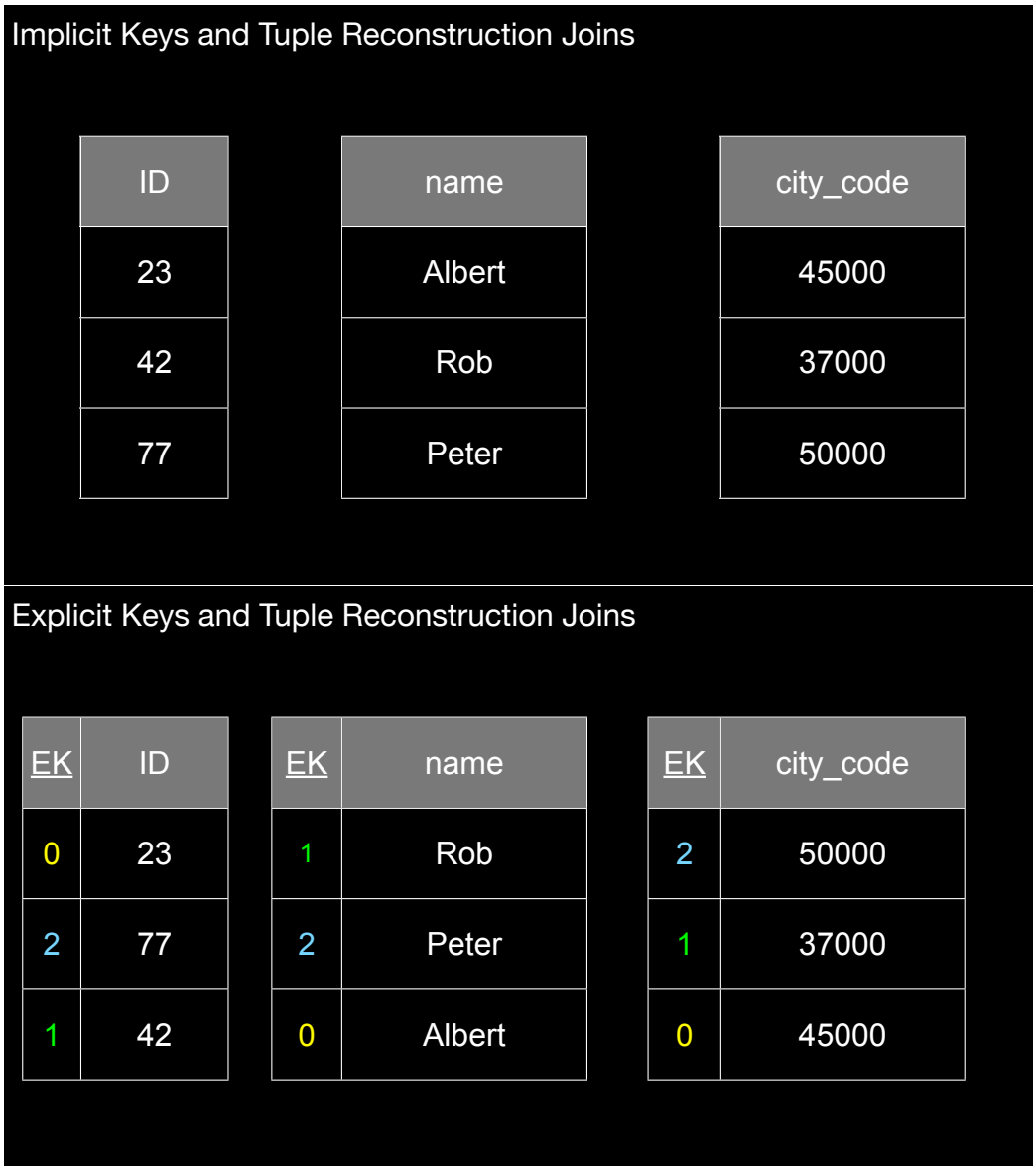


Figure 2.6: Implicit key vs explicit keys in a column store

What are correlated columns? **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.

What is an implicit key? **implicit key**

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).

uncorrelated

Could columns also be uncorrelated?

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

explicit key

What are the consequences of using uncorrelated columns? What is an 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.

Quizzes

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 **redundant**

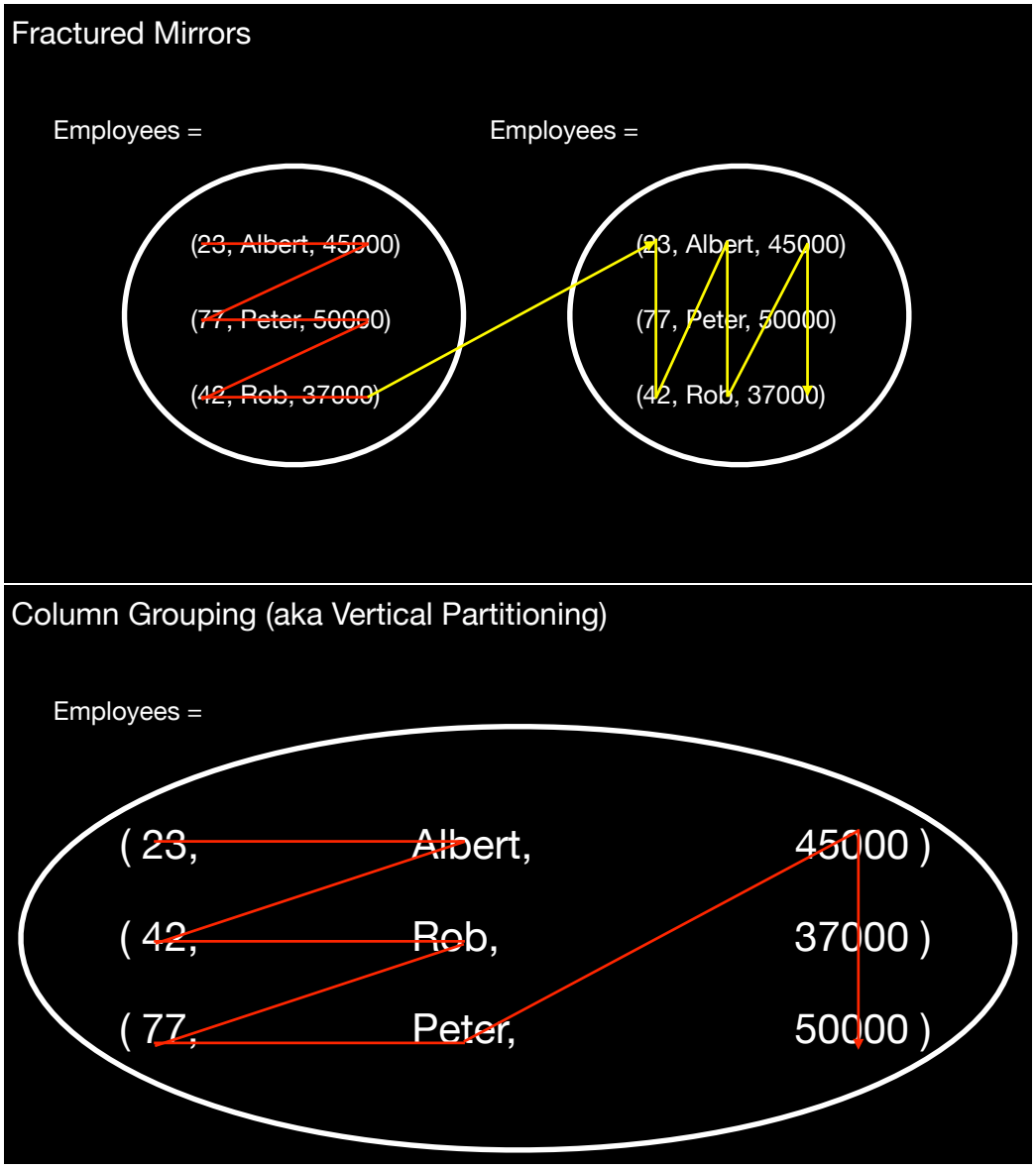


Figure 2.7: Fractured mirrors vs column grouping

mathematical sense as it returns a set of values.

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

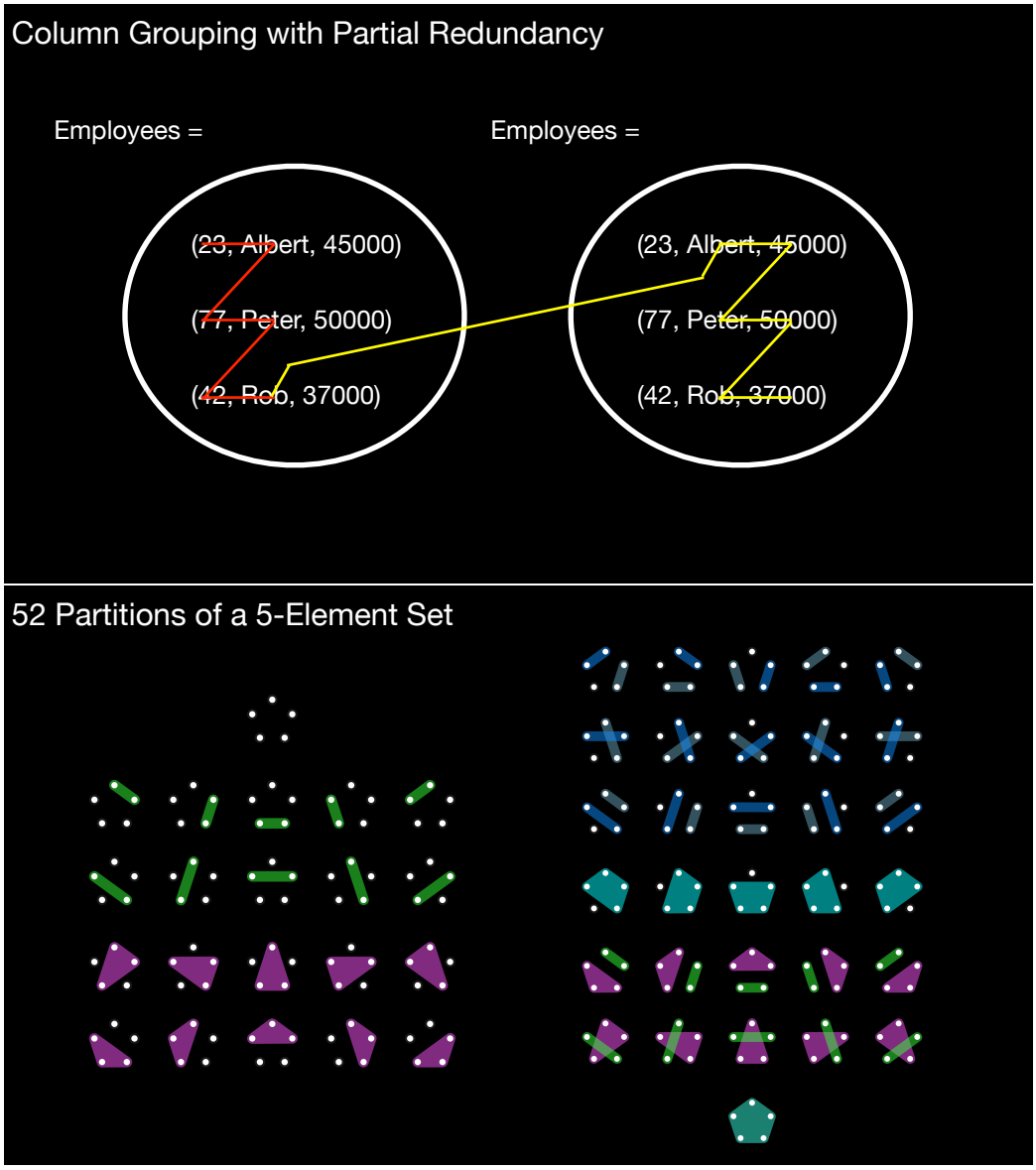


Figure 2.8: Column grouping with partial redundancy and the 52 possible partitionings of a 5-element set

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.

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.

column grouping

What is column grouping and its relationship to vertical partitioning?

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 partitioning 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?

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: **Bell number**

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

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.

Quizzes

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.

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

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

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

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.

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 simplicity 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?

(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

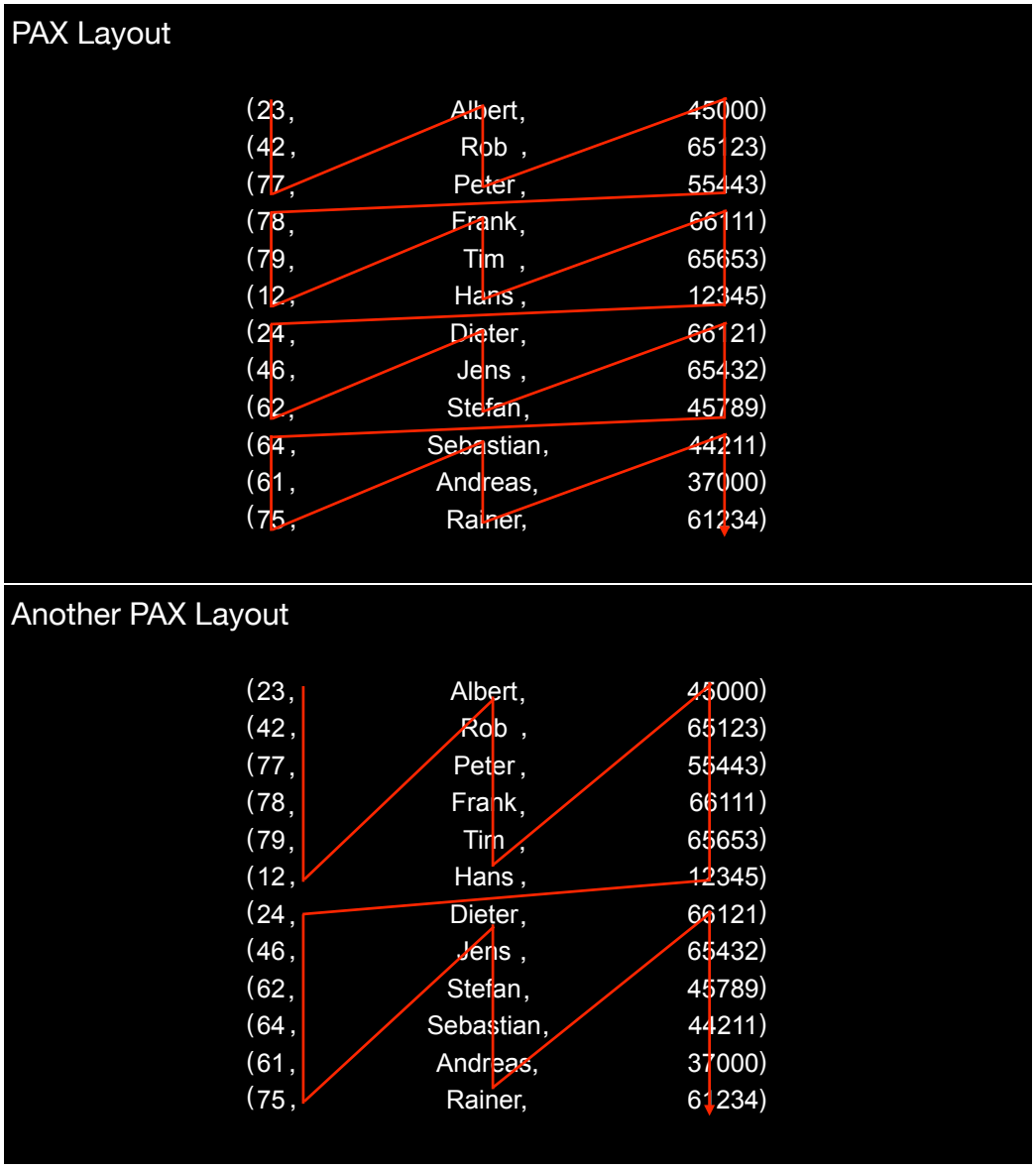


Figure 2.10: Two different variants of PAX layout: three rows per block vs six rows per block

PAX

What is PAX about?

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***.

horizontal partitioning

How does PAX relate to horizontal partitioning?

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

column layout

How does PAX relate to column layouts?

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. tu-

ples. 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.

optimal data layout

How do we get the optimal data layout anyway?

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.

Quizzes

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:

Learning Goals and Content Summary

fractal design

self-similar design

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

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.

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.

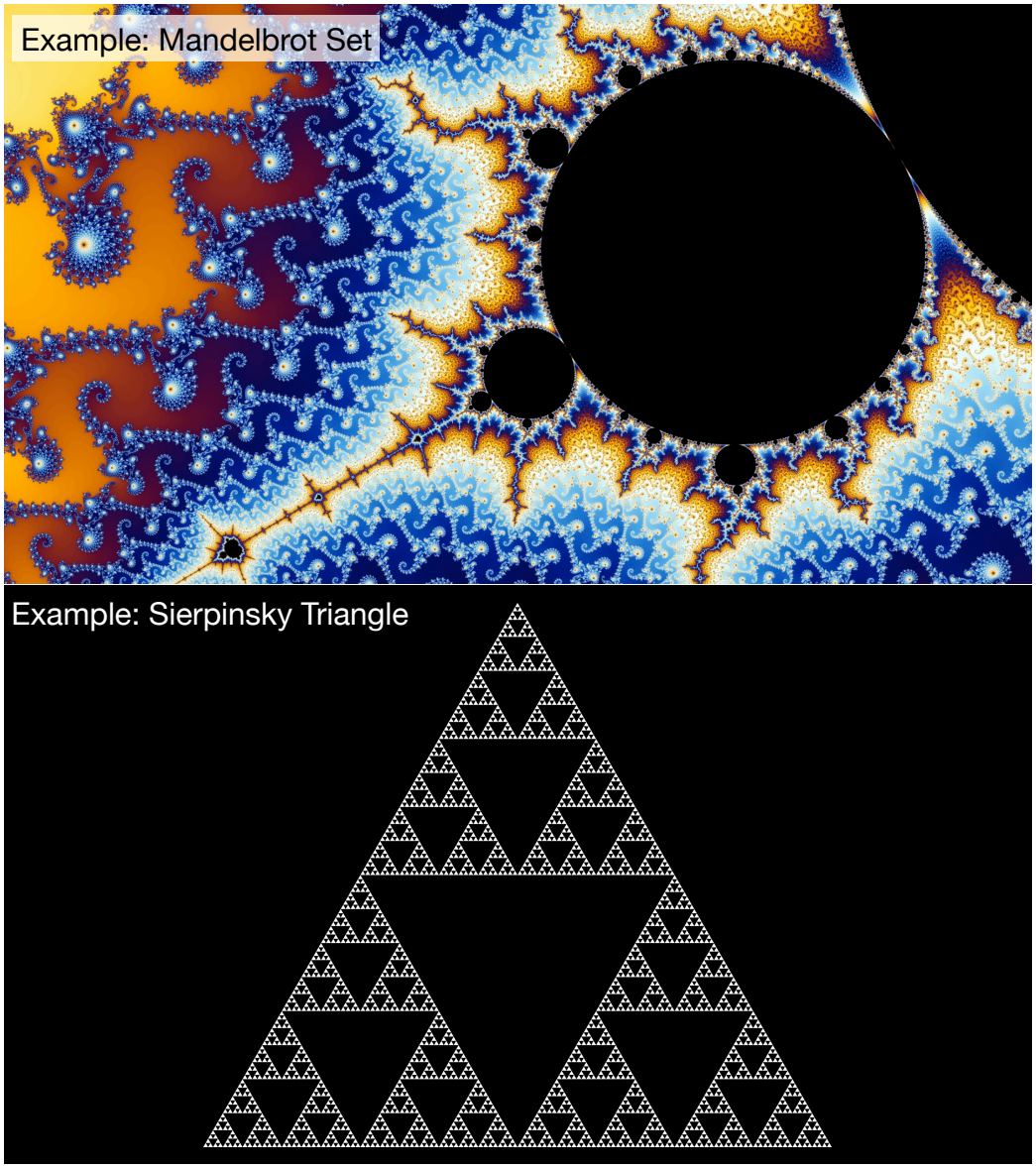


Figure 2.11: Example fractals: the Mandelbrot Set and the Sierpinsky Triangle

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

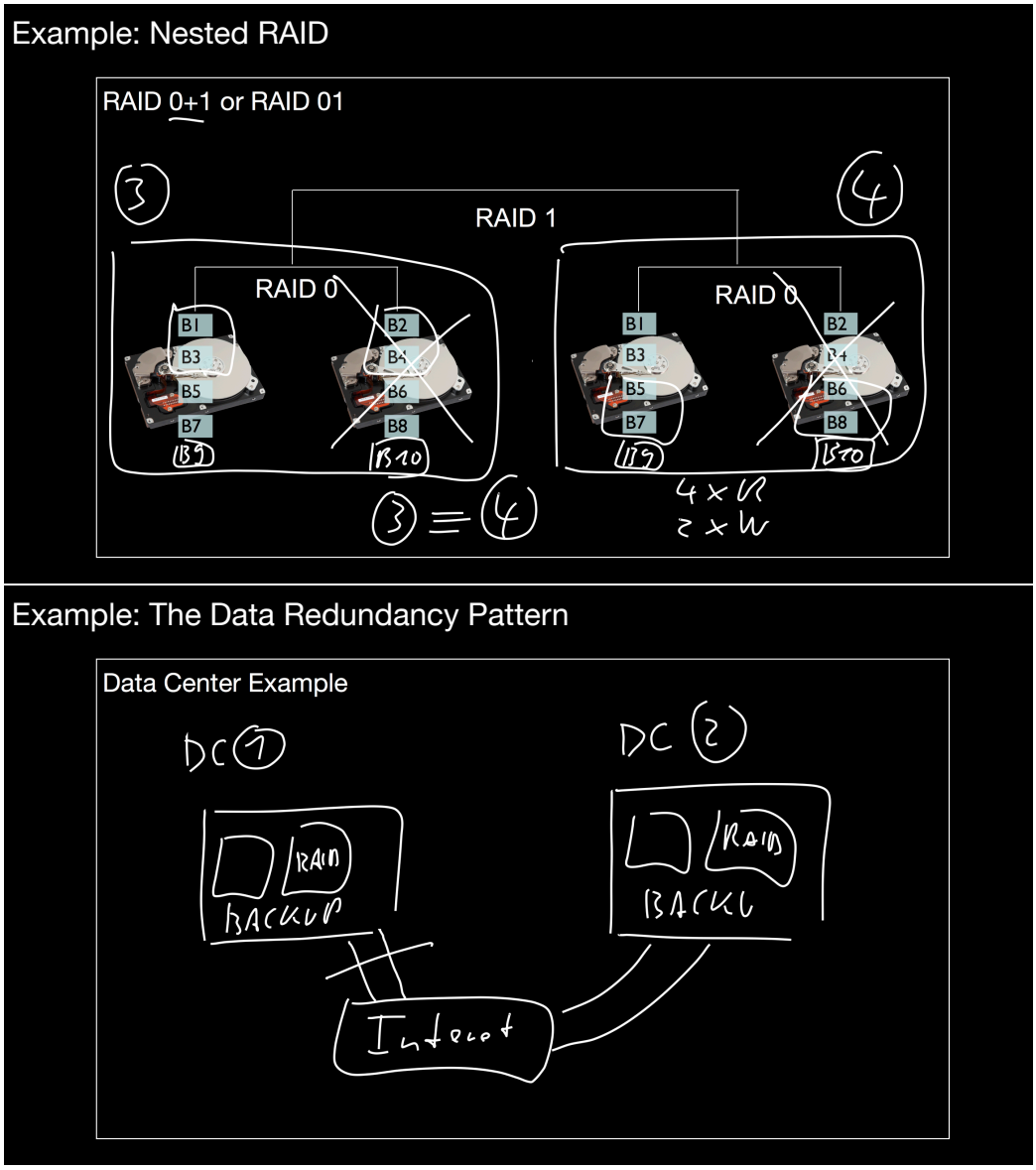


Figure 2.12: Examples for self-similar design in databases: nested Raid and data redundancy

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.

Quizzes

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) _____

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

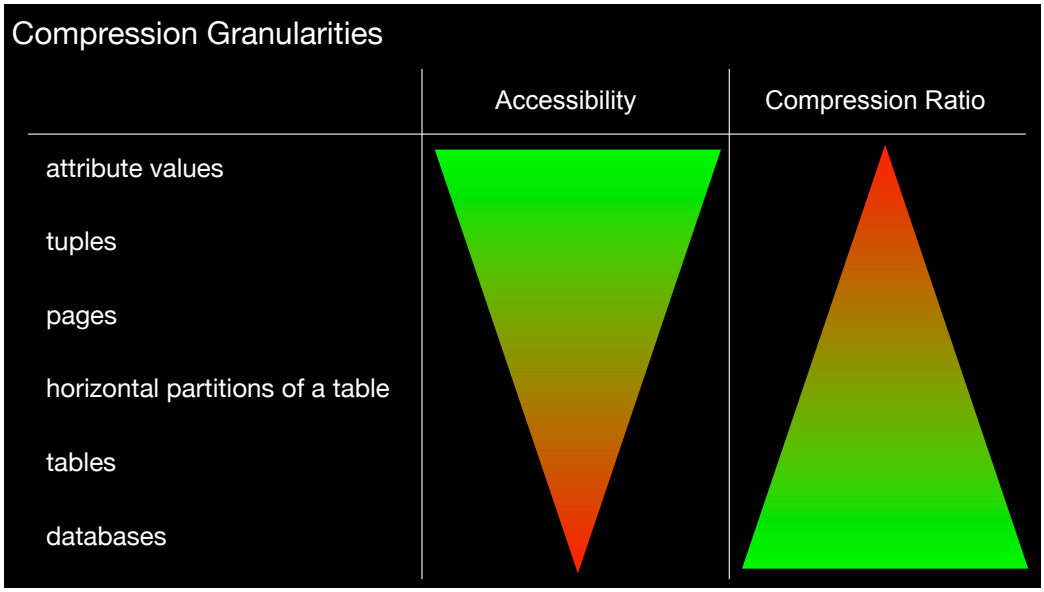


Figure 2.13: Compression granularities and their trade-offs w.r.t. accessibility of data and compression ratio

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) $\mapsto$ 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.

Quizzes

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

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,

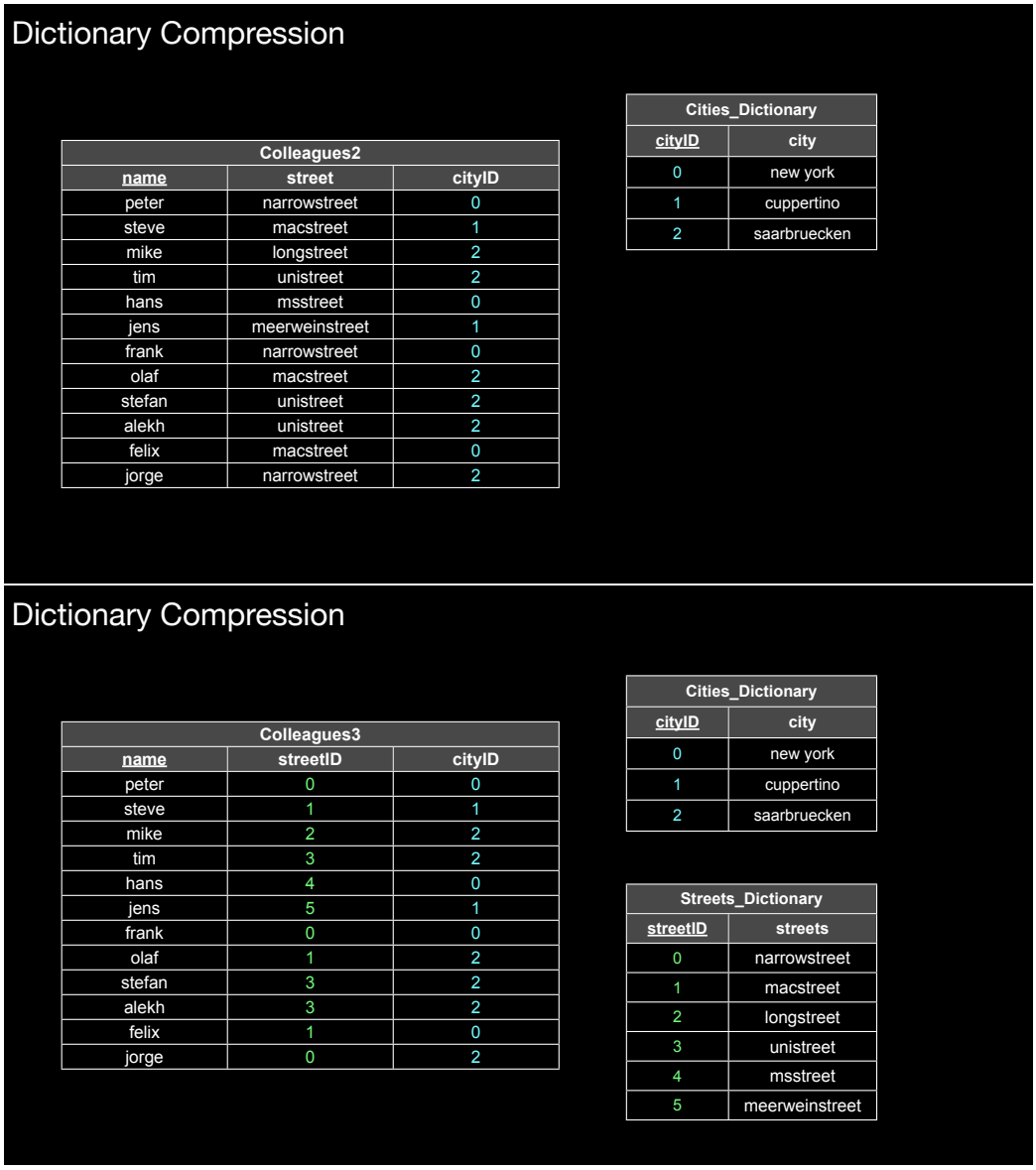


Figure 2.14: Dictionary compression: just column `cityID` vs both `streetID` and `cityID`

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 $\{ 0 \mapsto$ "foo", $1 \mapsto$ "bar", $2 \mapsto$ "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 var-

char(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.

Quizzes

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:

Learning Goals and Content Summary

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

run length encoding

RLE

How does run length encoding (RLE) work?

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.

What is the relationship of RLE to sorting?

sorting

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

What is the relationship of RLE to lexicographical sorting?

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.

Quizzes

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

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

7Bit Encoding

What is the core idea of 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 be more beneficial to use dictionary compression followed by domain encoding (assuming there are duplicate values).

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

storage space

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.

Quizzes

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. 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

filter index

post-filter

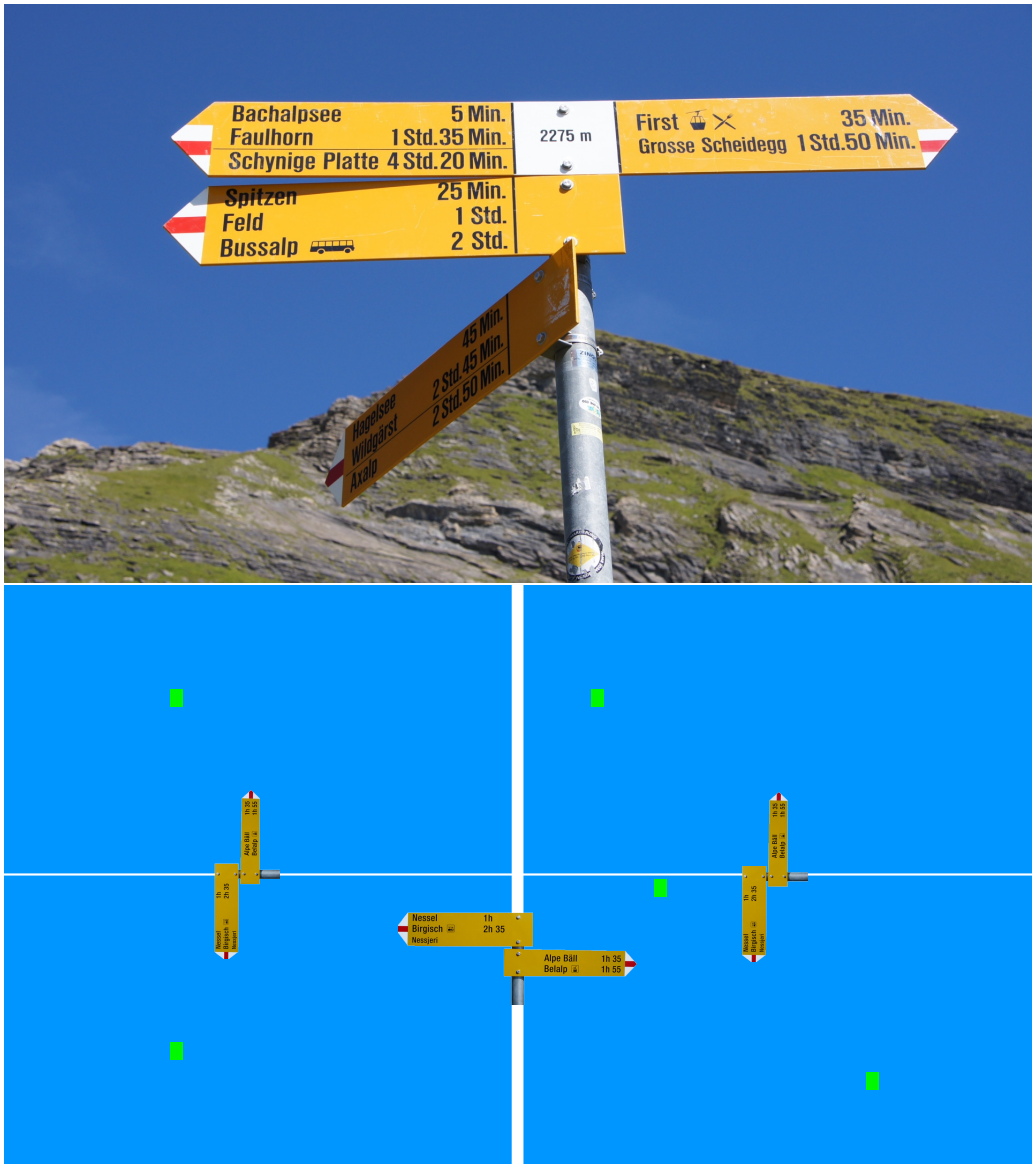


Figure 3.1: Indexes work just like street signs.

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 words: this is the ratio of the number of tuples of relation R where a=42 over the total number of tuples in R.

high selectivity

What is high selectivity?

The term "high selectivity" is counterintuitive: it means that $\frac{|\text{result set}|}{|\text{set}|} \leq 1$ is very small, i.e. close to 0. In other words, only few elements qualify under the selection, most of the elements are discarded.

low selectivity

What is low selectivity?

The term "low selectivity" is counterintuitive as well: it means that $\frac{|\text{result set}|}{|\text{set}|} \leq 1$ is very big, i.e. close to 1. In other words, many elements qualify under the selection, only few elements are discarded.

What type of data managing systems are indexes important for?

Indexing technology is important for almost all types of data managing system. In particular, one can sometimes read statements that main-memory database systems do not use indexes. This is simply not true. Be it that your data resides on disk, in main memory or on any other device. In many situations the right types of indexes allow you to speed up look-ups by several orders of magnitude. However, keep in mind that a simple scan in a main-memory system may be very fast, e.g. on the order of milliseconds for scanning 100 million entries in a column single-threaded. Depending on your runtime requirements this may be fast enough already for your application. Hence, technically you may not need indexes to fulfill your runtime requirements, yet you may still need indexes to reduce energy consumption and overall throughput of your system.

Quizzes

1. What are the basic index structure types mentioned in the video and also used in databases?
   (a) hash maps
   (b) trees
   (c) street signs
2. Which of the following are examples of real-life physical index structures?
   (a) catalogs in libraries
   (b) phone books
   (c) street signs
   (d) advertisements
   (e) cooking books
3. What does it mean for a query q to have a high selectivity?
   (a) The value $sel(q)$ is high.
   (b) The value $sel(q)$ is close to zero.
   (c) Only a few tuples are selected by the query.
   (d) Only a few tuples are excluded from the result.
4. Should you use indexes in main memory?

(a) No. Indexes were invented for disk based systems. In memory it is feasible to perform full scans for all queries.

(b) Yes, if it is more beneficial than doing a full scan.

5. What is the definition of the selectivity of a selection predicate P on a relation R?

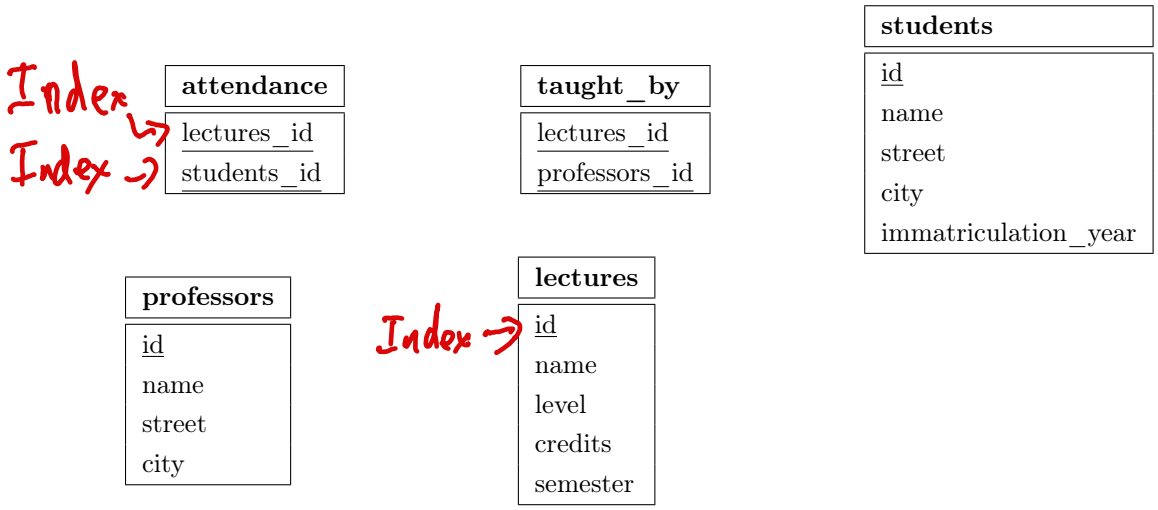
(a) $sel(P) = |\sigma_P(R)|/|R|$

(b) $sel(P) = 1 - |\sigma_P(R)|/|R|$

(c) $sel(P) = |R|/|\sigma_P(R)|$

Exercise

Suppose that the database of a University contains, among others, the following relations (all containing a considerable number of tuples):



Assume that the contents of all tables changes frequently through INSERT, UPDATE, and DELETE operations (more changes happen to **students** and **lectures** in the semester break). In addition, the University's information system triggers SQL-queries retrieving the following data:

Q1: get all the lectures (name and semester) a given student (by students.id) attended

Q2: get the name of all students attending a given lecture (by lectures.id)

Q3: get the name of all students who are in a given semester, e.g. fifth semester

Q4: get the city of a given student (by students.id)

Q5: get the cities for all students in the database

Q6: get all students living in a particular city, e.g. "Saarbrücken"

Q7: get all the lectures (name and semester) taught by a given professor (by professors.id)

Given the above information, for each query, i.e. **treating each query Q1, ... , Q7 independently**:

(a) Which index(es) would you create (provided in SQL syntax)?

(b) What would be a suitable data layout?

Given the above information, **for the database as a whole**:

(c) Which index(es) would you create (provided in SQL syntax)?

(d) What would be a suitable data layout?

3.2 B-trees

3.2.1 Three Reasons for Using B-tree Indexes, Intuition, Properties, find(), ISAM, find_range()

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Indexed Sequential Access Method
[RG03], Section 8.3.2, [RG03], Section 10.1 – 10.4

Learning Goals and Content Summary

The three reasons for using B-trees are ... ?

B-trees

1. B-trees are storage-friendly, i.e. they can be adapted to different layers of the storage hierarchy. See also the All Levels are Equal Pattern in Section 1.1.1 and Fractal Design, Section 2.3.5.

2. B-trees strike a good balance in terms of sequential versus random layout. For instance, consider a binary search tree (BST): it keeps only one pivot element at each node. This means with high probability at every node hop you need to visit a hard to anticipate memory address. The number of expected cache misses is high. And 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 big advantage of BSTs is that their structure may be changed easily in-between

BST

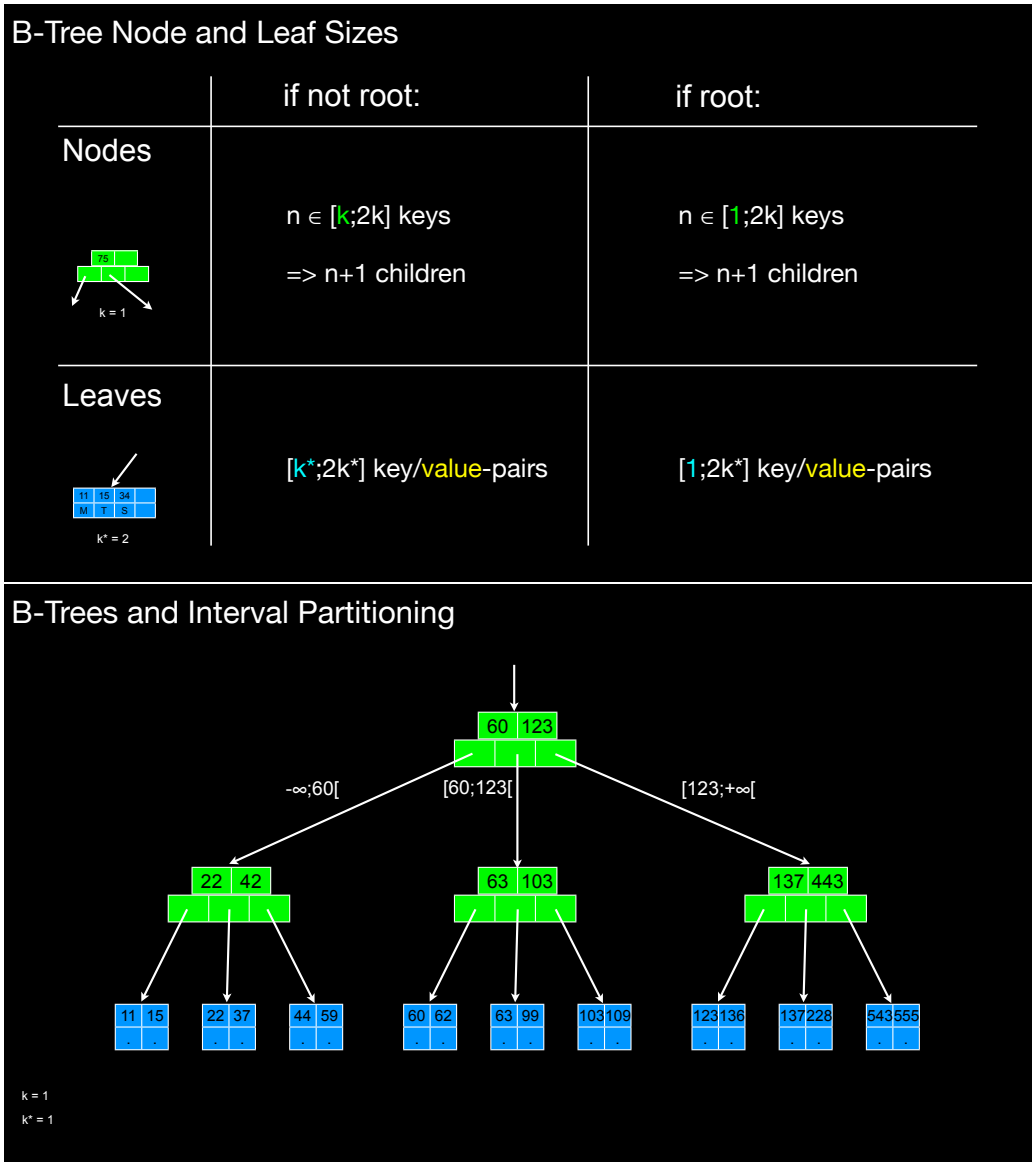


Figure 3.2: B-tree properties and their recursive interval partitioning

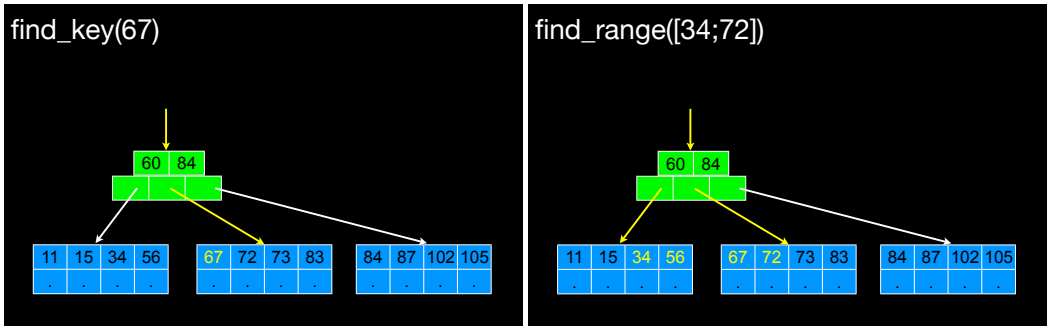


Figure 3.3: Searching keys and intervals in a B-tree

any node (and hence pivot element). Thus, the structure of a BST is extremely flexible. The other extreme in terms of sequential versus random layout is an array. It is simply a compact chunk of sequential memory. All pivot elements representing internal nodes can be precomputed (and or cached automatically, e.g. through changing the layout of the array in clever ways) leading to fewer cache misses. However, it is very hard to insert and or delete elements in/from the array. B-trees are a compromise in-between those two extremes: they inherit the flexibility of BSTs and the compactness of arrays.

3. B-trees are extremely flexible and may me adjusted to support more advanced indexing problems. For instance, in order to index two-dimensional data, B-trees may be extended. The resulting tree is then coined an R-Tree (a rectangle tree). R-Trees inherit all core properties already known from B-trees, but introduce important extensions. Still at a high-level, an R-Tree is a special case of a B-tree, and some software libraries implement it accordingly [dBBD+01]. **R-Tree**

What is the intuition for B-tree indexes?

B-trees connect chunks of data in a tree-structure. In contrast to BSTs, the chunks are much larger. Typically, the chunk corresponds to the granule of some layer in the storage hierarchy, e.g. a disk page or a cache line. Other than that the ideas of BSTs and B-trees are very similar: in both structures the data must be maintained range-partitioned. Otherwise, search operations would not improve much.

What is a node? **node**

A node keeps a set of keys (aka pivots) and a set of pointers to children. The children may be nodes or leaves. A node is sometimes also called an inner node or index node. We will simply call it *node*. **inner node** **index node**

What is a leaf? **leaf**

A leaf keeps a set of key/value-pairs.

What is k? **k**

The parameter k is the minimum number of keys in a node, $2k$ the maximum number of keys in a node. Notice that these parameters may also be defined such that $k/2$ is the minimum number of keys and k is the maximum number of keys in a node. We will stick to the former definition.

What is F? **F**

The fan-out F (aka branching factor) of a node is the number of its children and hence the number of subtrees of this node. For a node having $y \in [k; 2k]$ keys, its fan-out is $F = 2y + 1$. Notice that the term fan-out may be used to either denote the minimum fan-out ($F_{min} = k + 1$), the current fan-out ($F_{current} = 2y + 1$ where $y \in [k; 2k]$) or the maximum fan-out ($F_{max} = 2k + 1$). **fan-out** **branching factor**

What is h? **h**

The height h of a B-tree depends heavily on its fan-out F. **height**

k*

What is k?*

The parameter $2k^*$ defines the number of entries in a node. Notice that this parameter may also be defined such that k is the maximum number of elements in a node and $k/2$ the minimum number of elements.

How many keys does a node have?

In general, a node has $n \in [k, \ldots, 2k]$ keys. This implies that a node has $n+1$ pointers to children. If that node is the root node, it may have $n \in [1, \ldots, 2 \cdot k]$ keys.

How many keys does a leaf have?

In general, a leaf has $n \in [k^*, \ldots, 2k^*]$ key/value-pairs. A leaf does not have children. If that leaf is the root node, it may have $n \in [1, \ldots, 2 \cdot k^*]$ keys.

index sequential access method

ISAM

What is the index sequential access method (ISAM)? How is ISAM related to point and range queries?

The index sequential access method (ISAM) extends a B-tree to additionally implement a doubly linked list among all leaves. This means, each leaf stores two additional pointers: one to its left sibling and one two its right sibling. ISAM allows us to scan multiple leaves in ascending or descending key order without having to visit all visited leaves through their parent nodes. ISAM is useful to implement efficient range queries.

What are the major properties of a B-tree?

1. The path from the root (be it a node or a leaf) to any leaf has always the same length. In other words, a B-tree is always balanced w.r.t. the number of nodes and leaves visited during a search.

2. The parameters k and k^* are implicitly defined by the size of a node. For instance, assume we set the node size to 4KB, a single key (a pivot) requires 4 Bytes, and a pointer to a child requires 8 Bytes. Then a node with k entries occupies $k \cdot (4+8)+8$ Bytes $\Rightarrow k = \lfloor (4096-8)/(4+8) \rfloor = 340$. Similarly, assuming that a single value (of a key/value-pair) occupies 6 Bytes and each leaf contains two pointers to support ISAM. Then a leaf with k^* entries occupies $k^* \cdot (4+6) + 2 \cdot 8$ Bytes $\Rightarrow k^* = \lfloor (4096 - 2 \cdot 8)/(4+6) \rfloor = 408$.

3. The nodes do not contain values (of key/value-pairs). These values are only kept in leaves. The reason for this is that this makes ISAM more efficient.

4. The keys inside nodes and the key/value-pairs inside leaves are sorted by key. This allows for efficient binary search *inside* a node or leaf. However, this sorting is no strict requirement.

5. Notice that B-trees exist in many different variants, e.g. B-tree, B^+-tree, and B^*-tree. The major difference of B-trees and B^+-trees is: B-trees also store values in nodes. In addition, B-trees do not support ISAM. In the following we will only consider B^+-trees, yet to be consistent with database literature, we simply refer to B^+-trees as *B-tree*.

What is the relationship of B-trees to interval partitioning?

interval partitioning

Every node and every leaf implies an interval partitioning. Assume a root node with only two keys 60 and 123 with sub-tress $sub1$, $sub2$, and $sub3$ (see Figure 3.2). Assume that we use the convention that equal keys are represented in the right subtree. For instance, if we search for key 60, we have to continue our search at the right child of 60, i.e. subtree $sub2$. This means this root node partitions the entire tree into three subtrees where each of the three subtrees represents a particular interval of keys. In the example, subtrees $sub1$, $sub2$, and $sub3$ represent intervals $]-\infty; 60[$, $[60; 123[$, and $[123; +\infty[$, respectively. This interval partitioning is then applied recursively. For instance, subtree $sub1$ represents interval $]-\infty; 60[$. In Figure 3.2, the node at the second level with keys 22 and 42 implies a refined partitioning into intervals $]-\infty; 22[$, $[22; 42[$, and $[42; 60[$, respectively.

How do we search for a particular key in a B-tree, i.e. how do we execute a point query?

point query

We start our search at the root. If the root is a node (not a leaf), we have to find the appropriate subtree, i.e. the appropriate child. To support this a node should implement a method `chooseSubtree(Key K)` $\rightarrow$ *Child*. For a given key K this method returns the subtree which may have data for that key. We then continue our search recursively on that subtree. If the root of that subtree is a node, we call `chooseSubtree(Key K)` on that node again. If it is a leaf, we reached the last level of the tree. Inside a leaf, we check whether key K exists (using binary search). If that is the case, we return the value associated to that key. In other words, a point query in a B-tree is similar to a point query in a binary search tree. The major difference is: at every node in a B-tree there are possibly more than two children.

How do we search for a particular interval in a B-tree, i.e. how do we execute a range query?

range query

In order to find all entries in key interval K1,..., K2, first, we execute a point query on K1. Once we found the first entry of K1 in a leaf in that point query (or if an entry for K1 does not exist, the first entry bigger than K1), we simply scan along the leaf-level until we find a value strictly bigger than K2. At that point we stop our search. Notice that this method only works if the B-tree implements ISAM.

Quizzes

1. Which of the following trees do not not store all keys in the leaf nodes?
   (a) B+-trees
   (b) (original) B-trees
2. Assume a B-tree (yes, we mean B+-tree!) where each node/leaf can contain up to 1000 keys is used to index a set of 1,002,001,000 ordered keys. All nodes and leaves of the B-tree are fully occupied. If the root is always kept in main memory, how many disk accesses are needed at most to find any key?
   (a) 20
   (b) 2

(c) 3

(d) 4

3. Which of the following trees keeps its leaf nodes in a double-linked list for easier range queries?

   (a) B+-trees

   (b) (original) B-trees

4. When looking for a key in a B-tree (yes, we mean B+-tree!), binary search may be used inside nodes to guide the search correctly. If the height of the B-tree is h and its branching factor (aka fan-out) is F, what is the number of key comparisons made in a search operation assuming all nodes are fully occupied in the worst case, roughly speaking?

   (a) $F \cdot h$

   (b) h

   (c) $\log_F(F-1) \cdot h$

   (d) $\log_2(F-1) \cdot h$

3.2.2 B-tree insert, split, delete, merge

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], B^+-tree
[RG03], Section 10.5–10.6

Learning Goals and Content Summary

insert

How does a B-tree insert work?

In order to insert a key/value-pair (k, v) into a B-tree, we first execute a point query with key k. Like that we eventually find the position in a leaf L where (k, v) belongs. If L still has room to store (k, v) , we simply insert (k, v) into that leaf (preserving the order of key/value-pairs in L). If L does not have room to store (k, v) , we need to either (1) move some of L's entries to sibling leaves, i.e. we redistribute key/value-pairs such that L has room again to store (k, v). Notice that this operation may have to adjust pivots in parent nodes. Alternatively, (2) we split this leaf and then insert (k, v).

split

Why would we split a leaf or a node?

To locally increase the amount of available storage in the B-tree. This is necessary if a leaf or node does not have room to store a pivot/pointer or key/value-pair, respectively. Notice that a split creates a new leaf or node.

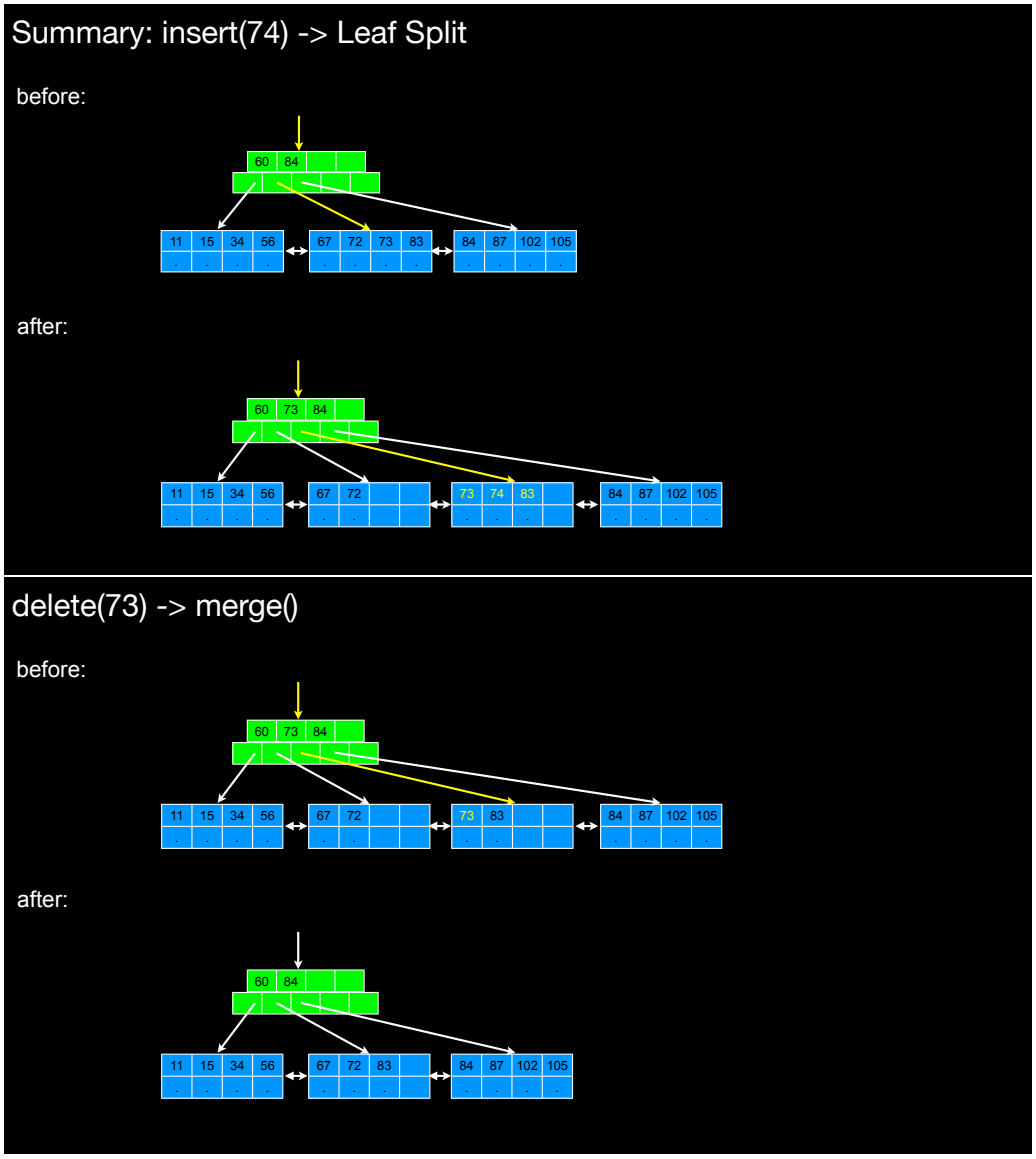


Figure 3.4: split() is the inverse of merge()

And how does this split work in principal?

A split of a leaf L_i simply adds another leaf L_{i+1} as a right sibling to L_i. In addition, a pivot element and a pointer to L_{i+1} needs to be added to the parent P of L_i. he pivot element should be chosen such that the entries are evenly distributed over L_i and L_{i+1}. If P does not have room to store that pivot and pointer, we first have to split P to make room, i.e. the split operation continues recursively up the tree. If in that process we need to split the root of the B-tree, we create a new root node pointing to the old root node and its newly created sibling. This also implies that a split of the root is the only possibility to increase the height of a B-tree. All other split operations make the tree wider but not higher; only a root node split makes the tree higher. A split of a node N_i works similar to a leaf split. We simply add another node N_{i+1} as a right sibling to N_i. In addition,

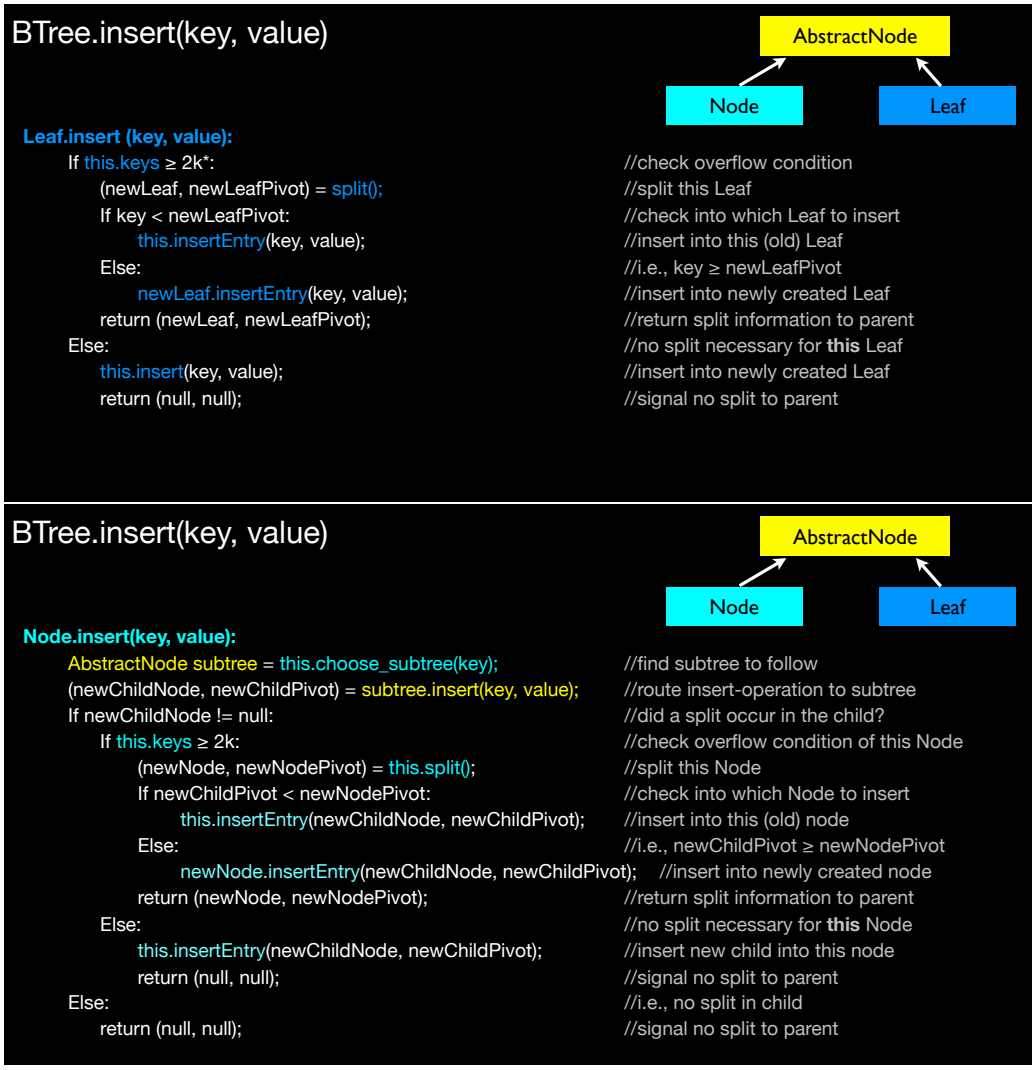


Figure 3.5: inserting into a leaf vs inserting into a node

a pivot element and a pointer to N_{i+1} needs to be added to the parent P of N_i. If P overflows, we have to split it recursively.

How do we implement the split operation in an object-oriented programming language?

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?

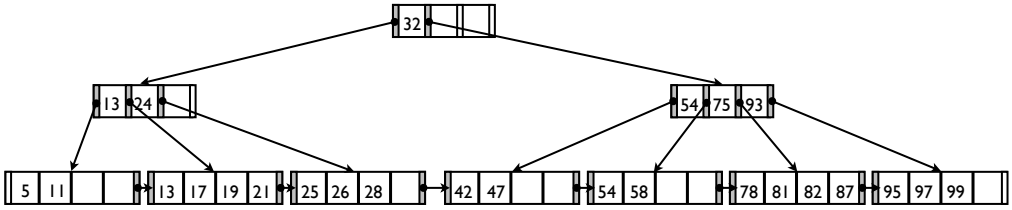


Figure 3.6: Initial B-tree.

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 merging continues recursively up the tree until we reach the root. If we try to remove the penultimate pointer from the root, we even remove the root. So again, and inverse to adding a level during insert, we can only remove a level when merging.

merge

How are merge() and split() related?

merge() is the inverse of split().

Quizzes

1. When a node n has to be split in a B-tree due to an insertion operation, which of the keys of n becomes the pivot?
   (a) The smallest key found in n
   (b) The median of the keys found in n
   (c) The largest of the keys found in n
2. Which of the following paradigms describes best the insertion operation in a B-tree?
   (a) Top-down
   (b) Bottom-up

Exercise

For each (a)–(f) consider the B^+-tree of Figure 3.6 as the initial state to answer the questions below. Perform the operations specified and draw the resulting B^+-tree.

(a) Insert a data entry with key 49.

(b) Insert a data entry with key 23. How many page reads and page writes does the insertion require assuming that at the beginning of the operation no page is available in the DB-buffer?

(c) Insert a data entry with key 90. How many page reads and page writes does the insertion require assuming that at the beginning of the operation no page is available in the DB-buffer?

(d) Delete the data entry with key 54, assuming that the left sibling is checked for a possible node merge.

(e) Insert 31 into the initial tree and then remove, from the resulting tree, the data entry with key 42. Assume again that the left sibling is checked first for a possible node merge.

(f) Successively delete the data entries with key 5 and 11.

3.2.3 Bulk-loading B-trees or other Tree-structured Indexes

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	Further Reading:
[RG03], Section 10.8.2	[dBS01]

Learning Goals and Content Summary

bulkload

How do we bulkload a B-tree?

There are many algorithms for bulkloading tree-structured indexes and in particular B-trees. An intuition for a simple and very efficient method is to: (1) sort the data w.r.t. the key attribute(s) to use for the B-tree. (2) Construct a B-tree from left to right and bottom to top.

In more detail, insert entries of the sorted data into a newly created leaf. When that leaf is full, create a second leaf and insert pointers to both leaves into a newly created parent node. Keep on adding leaves until the parent node is full. When it is full, add a new parent node, i.e. a new root on the same level as that parent. In addition, create a new node pointing to both the old and the new parent node. And so forth.

root

This method gives you the intuition, but it does not always create a valid B-tree. Whether the resulting tree is a valid tree depends on the amount of data inserted. In particular, this method may create a forest of trees or even underfull nodes and leaves. Fixing this is, however, rather simple and can be done in several ways. For the details, see the exercises.

What is bulkloading useful for?

Basically, in order to create a new B-tree on an already existing table, we have three options: (1) initialize an empty tree, insert tuples one by one (aka tuple-wise insert),

tuple-wise insert

(2a) bulkload the index as described above or (2b) sort data w.r.t. the key attribute(s) to use for the B-tree and then continue with option (1).

Option (1) is relatively expensive as every insert of a tuple into the tree performs an entire walk down the tree, possibly followed by one (or more) split(s). Option (2a) is cheaper as the tree is built from left to right. Still the tree grows by splitting nodes and leaves. In addition, as the input data is sorted, leaves and nodes will only be half full (except the last nodes and leaves in each level). Option (2b) is the cheapest method as no splits are performed. In addition, the entire tree may be written out in a single (streamed) sequential write operation. There is no need to perform random I/O-operations.

What should be kept in mind for the free space in nodes and leaves when bulkloading?

We can freely adjust how much free space we keep in nodes and leaves. Leaving some room in the nodes and leaves is useful in order to allow for efficient future inserts. If we keep too much free space, we waste space, but decrease the likelihood of splits. If we do not keep any free space at all, we do not waste space, but increase the likelihood of splits. So we need to balance space with insert performance.

Quizzes

1. Given a dataset with 20 tuples and a B-tree that can store three keys in the nodes and five keys per leaf. What do you need to do in order to bulk-load an index with those tuples?

   (a) always sort the data on the key

   (b) always sort on the very first attribute

   (c) create four leaves

   (d) create seven leaves

   (e) create a single node as the root node

   (f) create a root node together with two nodes

Exercise

Assume a B-tree residing entirely in main memory. You are allowed to use a node size of up to 64Bytes, i.e. the size of a cache line. Assume that the type of the key as well as the value is a 4Byte int each and the CPU architecture is 32 Bits. Further assume that the number of entries to store in the tree is N=39304 key/value-mappings. Assume that the tree was bulkloaded.

(a) What is the fan-out F of the internal nodes?

(b) What is the height h of the tree?

(c) How many cache misses CM do you expect in the worst case to fetch a value?

(d) What is the size S of the tree in Bytes?

(e) **Bonus:** so the entire tree fits into main memory. Can you come up with an alternative B-tree that improves ALL four variables F, h, CM, and S?

Exercise

Assume you have a set of entries with the following key values: 4, 9, 5, 8, 13, 16, 10, 27, 21, 1, 3, 43, 39, 15, 2, 23, 42, 40. Explain how to bulk-load this set of entries into a new B^+-tree. Each index-level page can hold up to 2 key values and each leaf page can hold up to 3 records. Draw the resulting B^+-tree for each of the steps, i.e. each leaf/inner node added, of the bulk-loading process.

Exercise

Assume you are getting a sorted stream of data pages (with unique keys). You have to build a sparse B^+-Tree with at most 4 keys per inner node and 4 keys per leaf node. None of the inner nodes is allowed to have less than two keys — except the root node.

(a) Assume you insert 6 data pages one at a time into an empty tree. What does the tree look like in the end? Draw the structure of the tree.

(b) What is the height of the tree if you simply insert n data pages one at a time?

(c) Provide an algorithm to create a compact B^+-tree, where compact means that you create a tree of minimal height. Your algorithm should denote when a node is flushed to disk. You should minimize the number of inner nodes you keep in memory and flush inner nodes as early as possible to disk. On the other hand you should never overwrite an already flushed inner node. Implement two methods: `void insert(DataPage d)` and `void close()` using Pseudo-Code or Java. When close() is called, then you must make sure that a valid B-tree is persisted (flushed) on disk.

3.2.4 Clustered, Unclustered, Dense, Sparse, Coarse-Granular Index

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	
[LÖ09],	Dense Index
[LÖ09],	Sparse Index
[RG03], Sections 8.2.1 and 8.5.2	

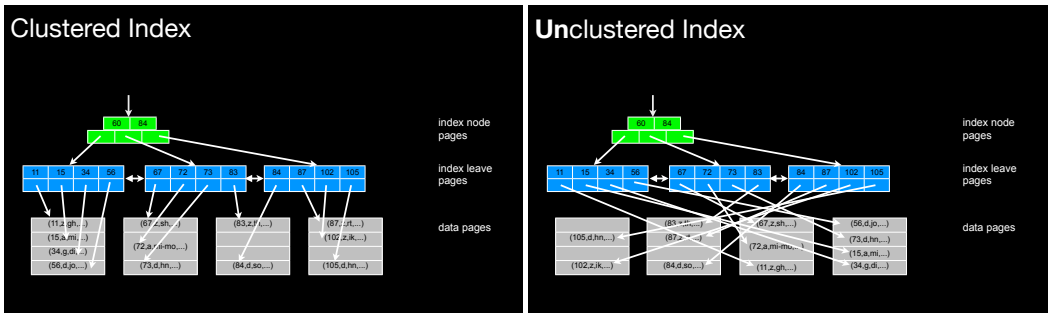


Figure 3.7: Clustered vs unclustered index

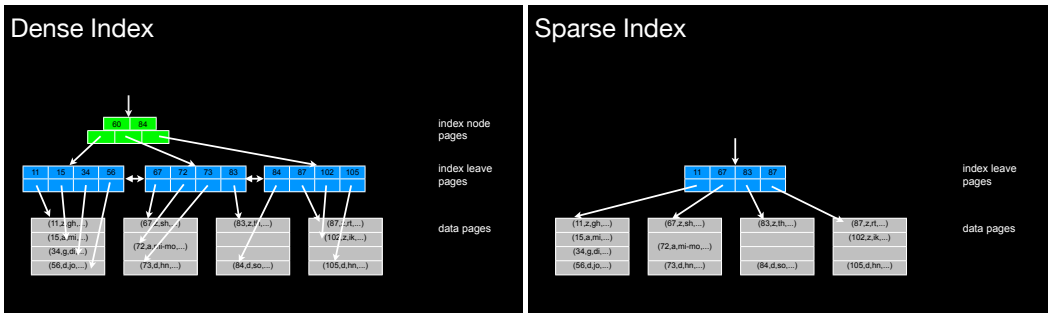


Figure 3.8: Dense vs sparse index

Figure 3.9: Coarse-granular vs no index

Learning Goals and Content Summary

clustered index

What is a clustered index?

In a clustered index the sort order of keys in the leaves corresponds to the sort order of tuples on the data pages. This implies that a range query can be answered by: (1) execute a point query, (2) visit the data page pointed to, and (3) continue scanning along the data pages (rather than the leaves). In other words, the index sequential access is done on the data pages rather than on the leaves (compare Section 3.2.1). Still ISAM on the leaf-level may be useful to execute EXISTS queries, i.e. to compute the keys that exist in a particular interval.

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.

unclustered index

What is an 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 sequential access is done on the leaves of the B-tree and every entry triggers a (possibly random) lookup to a page in the data store. Similar to clustered indexes, ISAM on the leaf-level may still be useful to execute EXISTS queries, i.e. to compute the keys that exist in a particular interval.

How many unclustered indexes are possible for a table?

You may create as many unclustered indexes as you want. But remember that all indexes have to be maintained whenever the underlying table is changed through inserts, updates, and deletes. Thus, the more indexes you create on a table the higher are the costs for inserts, updates, and deletes. There are two extremes in indexing: on read-only data the number of indexes is just limited by storage space. In contrast, on write-intensive data, the number of indexes is also limited by the index maintenance costs. To get optimal performance, it is important to find the right balance.

dense index

What is a dense index?

A dense index has one entry on the leaf-level for each row in the table.

sparse index

What is a sparse index?

In contrast to a dense index, a sparse index does not have an entry for every row in the table. Typically, a sparse index only has one entry on the leaf-level for each data page in the table. This implies that leaves cannot (always) answer EXISTS queries anymore, i.e. even though a key is not available in the leaf, it may still exist on a data page. In addition, in a sparse index, leaves play a similar role as nodes, as entries in a leaf mark key ranges in data pages. For instance, if the leaf contains two keys, say 67 and 83 (see Figure 3.8), the value pointed to from key 67 points to a data page containing keys in range $[67; 83[$.

What are the major advantages of a sparse index?

A sparse index has fewer entries. Therefore it needs less space. In addition, it is likely to have fewer levels. It is also easier to maintain, i.e. only if the range of the data kept in a data page is changed, the sparse index has to be adjusted accordingly.

coarse-granular index

How is a coarse-granular index related to a sparse index?

"Sparsity" may be defined in several different ways. As already stated above, typically,

a sparse index only has one entry on the leaf-level for each data page in the table. But what happens if we use even fewer index entries, say one entry for every two data pages? Then we evolve the index towards a coarse-granular index. Assume we use one index entry for every x pages. An index with $x = 1$ is called sparse, an index with $x > 1$ is called a coarse granular index. So x is a parameter defining the "sparsity" or "coarseness" of an index. Both sparse and coarse-granular indexes are both variants of what is called a filter index. A filter index is an index that returns a superset of the actual result. That superset then needs to be post-filtered by inspecting the database.

filter index

What are the major advantages of a coarse-granular index?

A coarse-granular index has even fewer entries than a sparse index. A major advantage of a coarse-granular index is that it is even cheaper to maintain than a sparse index. A drawback of a coarse-granular index is that (typically) a larger portion of data has to be scanned to answer a query. In general, sparse and coarse-granular indexes are beneficial to index data that is frequently changed. For these cases a dense index may be to expensive to maintain, but a sparse or coarse-granular index may be affordable.

How is a sparse index related to having no index at all?

Assume again we use one index entry for every x pages. Assume that y is the number of data pages that we want to index. Thus, the number of index entries is $\lceil y/x \rceil$. This implies that for $x \geq y$ the number of index entries is exactly one, i.e. the granule is so big that all data pages belong to the same partition. This case has a similar effect as not indexing the data in the first place; it actually has some overhead over having no index in the sense that we pay the extra costs for traversing the index and then scanning all data anyway.

Quizzes

1. In general, which kind of index is likely to have better performance in practice when doing range queries on the indexed attribute?
   (a) Dense unclustered index
   (b) Dense clustered index
   (c) Sparse unclustered index
2. Which of the following is a benefit of sparse indexes?
   (a) Index size is small
   (b) The system can directly tell whether a key exists using the index.
   (c) The system can directly tell whether a value exists using the index.
   (d) If the attribute of a tuple which is used as the key in the index is changed, we sometimes do not have to change the index at all.
3. When the selectivity of a range query is very low (say 10 percent of the elements qualify), which option is likely to have better performance in practice for an unclustered index?

(a) A coarse-granular index with a granule of two pages (assuming you have thousands of pages)

(b) A simple scan of the underlying table being indexed

(c) A sparse index

(d) A dense 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

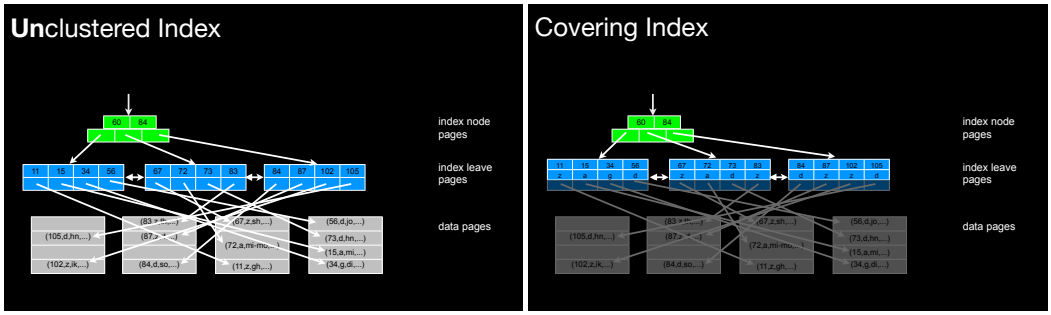


Figure 3.10: Non-covering vs covering index

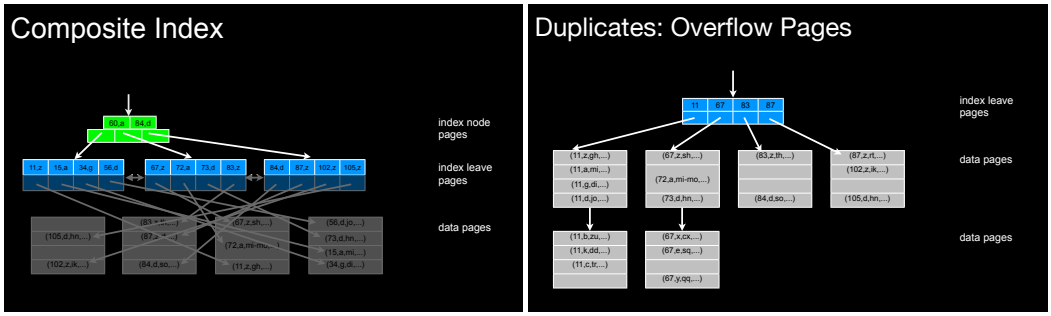


Figure 3.11: Composite (key) index vs overflow pages

What is a covering index? **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 qualifying entry we have to lookup the data page (following the rowID) to lookup B's **index only plan**

value, see also Figure 3.10. The disadvantage of covering is that attribute B is replicated: its values are stored both in the leaves and the data store. Covering indexes are also related to column layouts and vertical partitioning in general, as a scan along the leaf-level has similar advantages: rather than scanning large portions of a table in row-layout, we have to (potentially) scan less data. This may even be exploited in cases where the index does not help in evaluating the filter predicate, yet it may be exploited for a column scan on the attribute values that are covered in the index.

How are covering indexes related to clustered indexes?

A covering index has similar effects on the covered attributes as the clustered index has on the entire data pages pointed to. In other words, in a covering index the covered attributes are clustered w.r.t. the key used for indexing. In a clustered index, data pages are clustered w.r.t. the key used for indexing. So, on a high level, a covering index is a compromise in-between an unclustered and a clustered index.

How many covering indexes can we create on a single table?

In theory, as many as we want. The price we pay is additional storage space and update costs for the replicated attributes. Notice that the exact same answer can be given to determine how many clustered indexes to create.

composite index

What is a composite index?

In a composite index, we define the key to be a composite of multiple attributes, see also Figure 3.11. Like that all attributes pertaining to the key are implicitly covered by the index.

How is a composite index related to a covering index?

We may use a composite index in order to mimic a covering index, compare Figures 3.10 and 3.11. This is useful when a covering index is not supported by the database system. However, notice that a composite index for each key keeps the entire key in nodes as the pivot element[1]. For a fixed page size, this may potentially decrease the number of entries that can be kept in a node (parameter k). This reduces the fan-out of the nodes which in turn may lead to a tree with more levels and therefore may lead to slower queries.

duplicates

How do we handle duplicates in a B-tree?

There are many ways to handle duplicates: (1) introduce overflow pages, i.e. allow leaf-entries to point to a list of data pages rather than a single data page, (2) use a composite key to convert an attribute with duplicates into a duplicate-free column (not necessarily a key), (3) implement an integrated tree with three different node-types: nodes, leaves, and data pages (see discussion in the video for details).

Quizzes

1. Assume the following query is performed: SELECT C FROM Table WHERE A = a2 AND b1 <= B <= b2. The selectivity of the query is 20 percent, data is stored on HDD, and the size of attribute C is rather small. What kind of index is most

[1] Which in turn could be optimized by just storing a prefix of the key in the nodes as a pivot. This is done in prefix B-trees which are a general method to shorten keys in nodes.

beneficial for answering this query?

(a) Unclustered index on A

(b) Unclustered index on B

(c) Covering index on A that covers B and C

(d) Composite index on A and B

(e) Composite index on A and B that covers C as well

2. Assume an index is built using a composite key on a non-key attribute together with the primary key. How should this index handle duplicates?

(a) Store all duplicates in overflow pages.

(b) Not at all.

3. You would like to handle duplicates in a good manner in a B-tree, but unfortunately you have no access to the code of the B-tree. What method do you use then to handle duplicates?

(a) Overflow pages

(b) 3-node type B-tree

(c) Composite key containing a key attribute as well

(d) this is not possible to fix without having access to the source code

4. Assume the ratio of duplicates to the total number of keys is very bad, i.e., there are a lot of duplicates in the data. Which of the following (B-tree) indexes has the smallest memory footprint? By index size we mean only the size of the B-tree (internal nodes and the leaves)

(a) Using overflow pages

(b) Using composite keys

(c) 3-node type

3.3 Performance Measurements in Computer Science

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[Jai91], Sections 2 and 3
[LÖ09], I/O Model of Computation

Further Reading:	
[LÖ09], Performance Monitoring Tools	
W	Big O Notation
W	Simulation

Learning Goals and Content Summary

How to determine which algorithm, index, or whatever is better?

This is an easy question, however, it is not easy to answer. It depends on what is meant by "better"? Is "faster" the same as "better"? And then is it wall-clock time you are interested in? Or CPU time? Or is less memory consumption better? Less I/O? So before investigating how much "better" a particular method is, make sure you understand what "better" means to you. The same confusion exists with the term "performance".

What are the three ways to measure the performance of a computer program?

In general there are three approaches to measure performance (or other properties) of a computer program.

1. Analytical Modeling: we use a mathematical model of the algorithm or system we want to examine. Notice that a model can be created in many different ways. Models should always be built having a particular goal in mind: what do you want to do with that model? Which feature do we want to model? A "good" model is at the same time simple (it leaves away things we are not interested in in the analysis) and powerful (it allows us to realistically quantify performance). Once the model has been built, performance is determined by arguing along the model. Be careful not to confuse the model with the underlying reality. This mistake is, unfortunately, done frequently.

2. Simulation: we perform experiments with an analytical model of the algorithm or system. Hence, this method is a blend of analytical modeling and experiment. In contrast to analytical modeling, the focus is on executing the model rather than arguing along the model mathematically. In contrast to an experiment, we execute the model rather than the real system or algorithm.

3. Experiment: we run the actual algorithm and system.

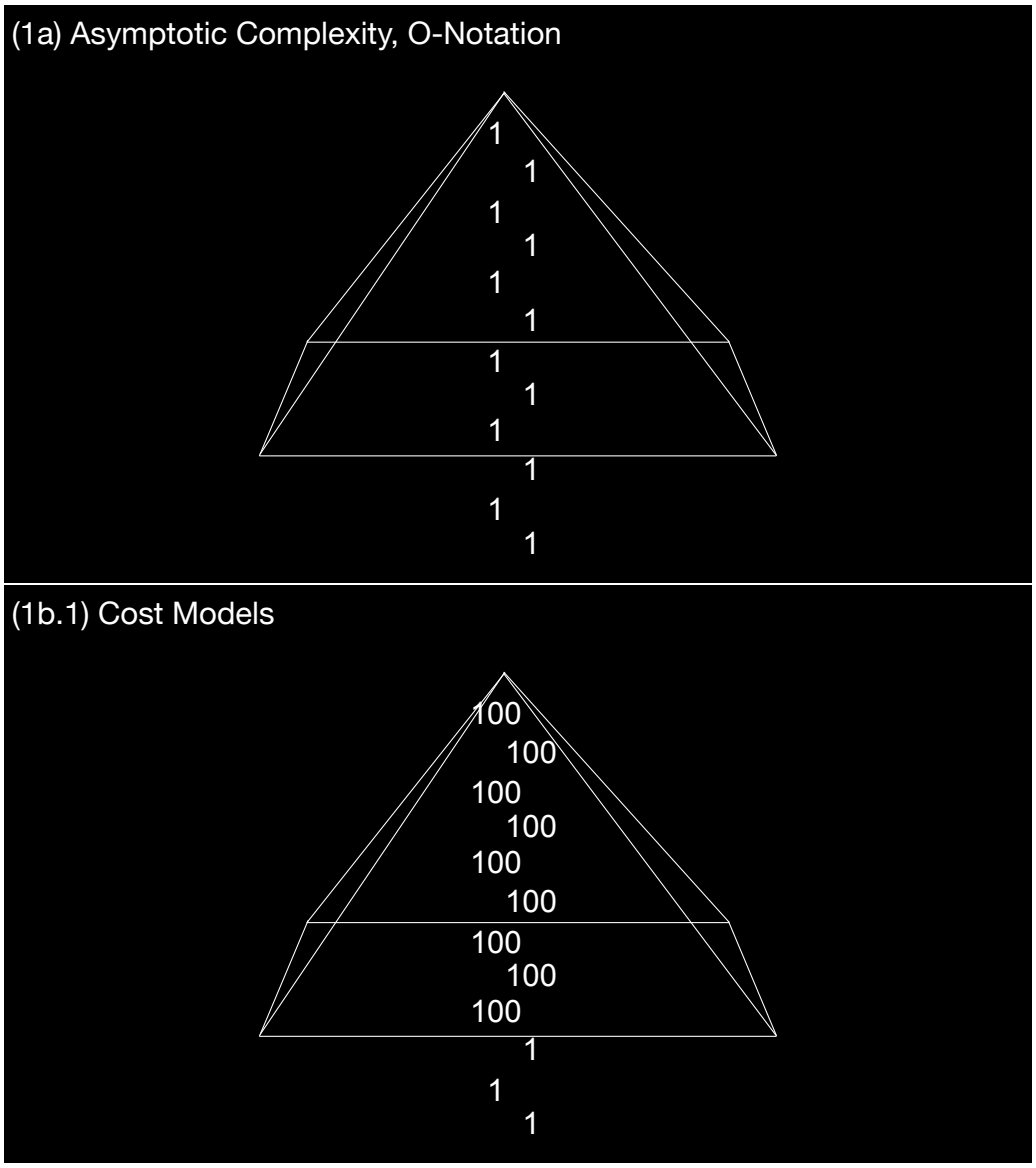


Figure 3.12: Asymptotic complexity vs cost models

Notice that the boundaries among the different methods are fuzzy. For instance, parts of a system may be simulated others may be executed.

O-Notation

O-Notation is good enough, right?

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

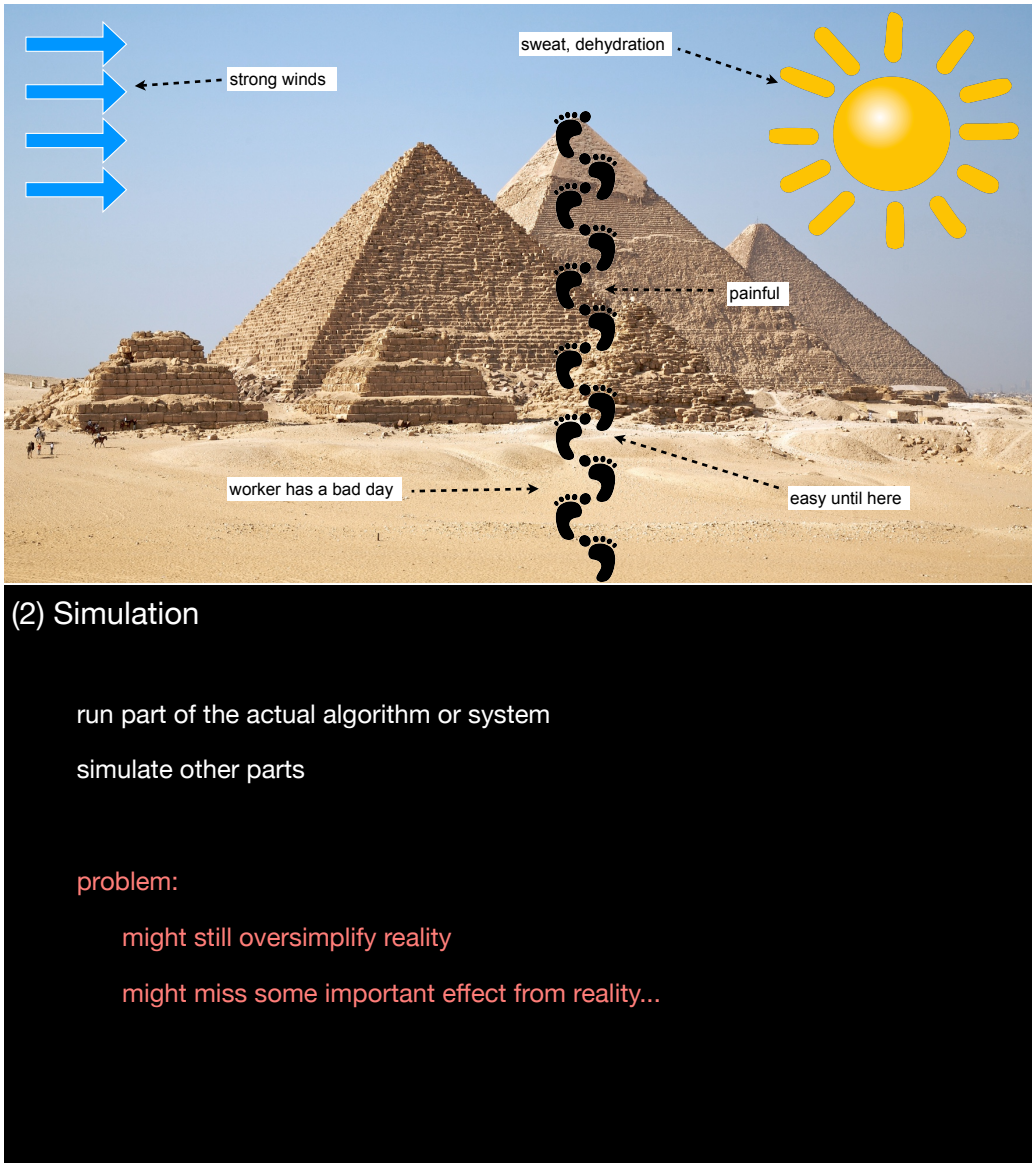


Figure 3.13: Simulating a pyramid

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 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 $O(n \log n)$,

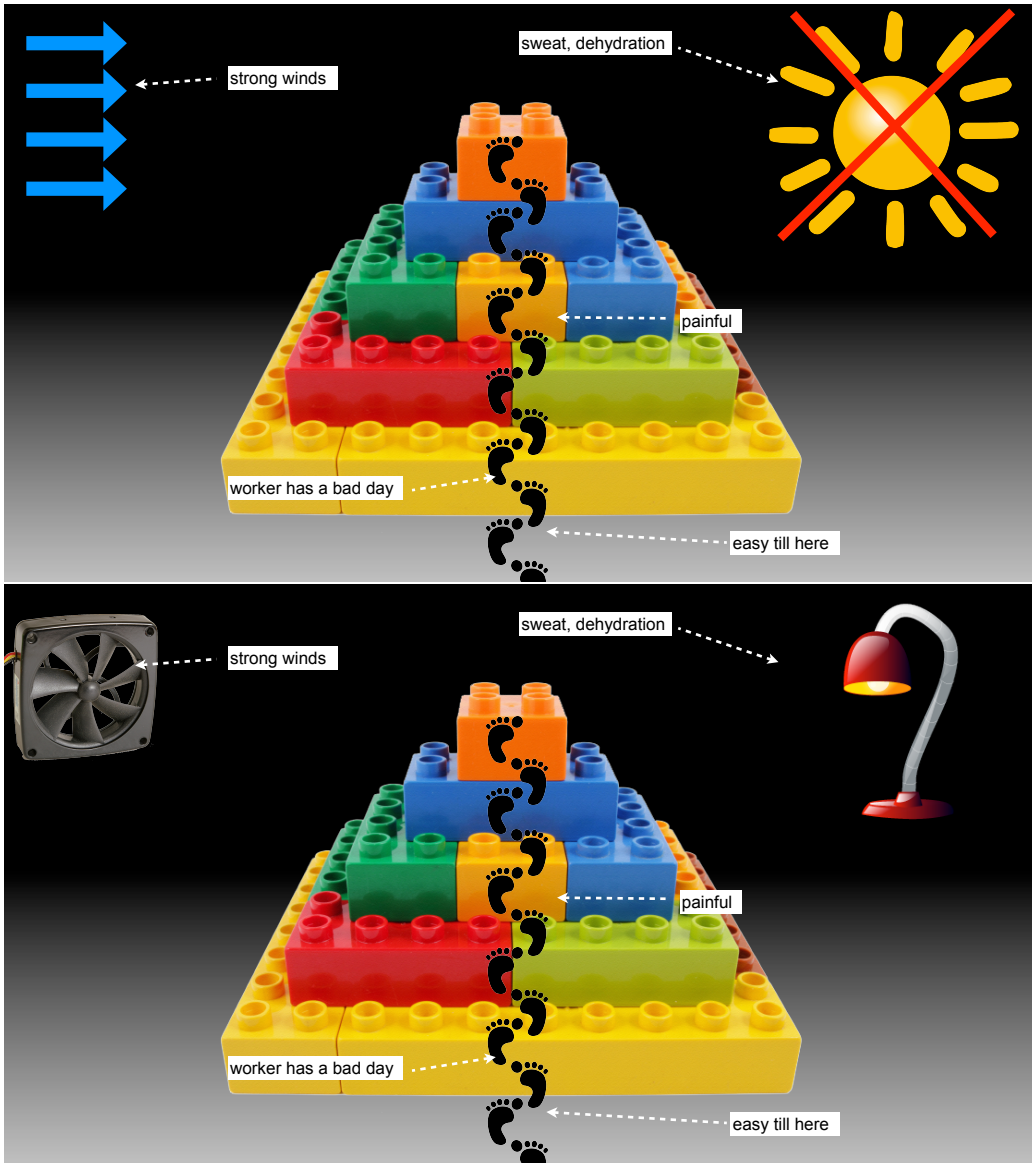


Figure 3.14: Additional influence factors when simulating a pyramid

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.

What is a simulation?

simulation

In a simulation we execute a model of the system or algorithm. This is useful if an analytical model of the system is too complex to be argued upon mathematically. It is

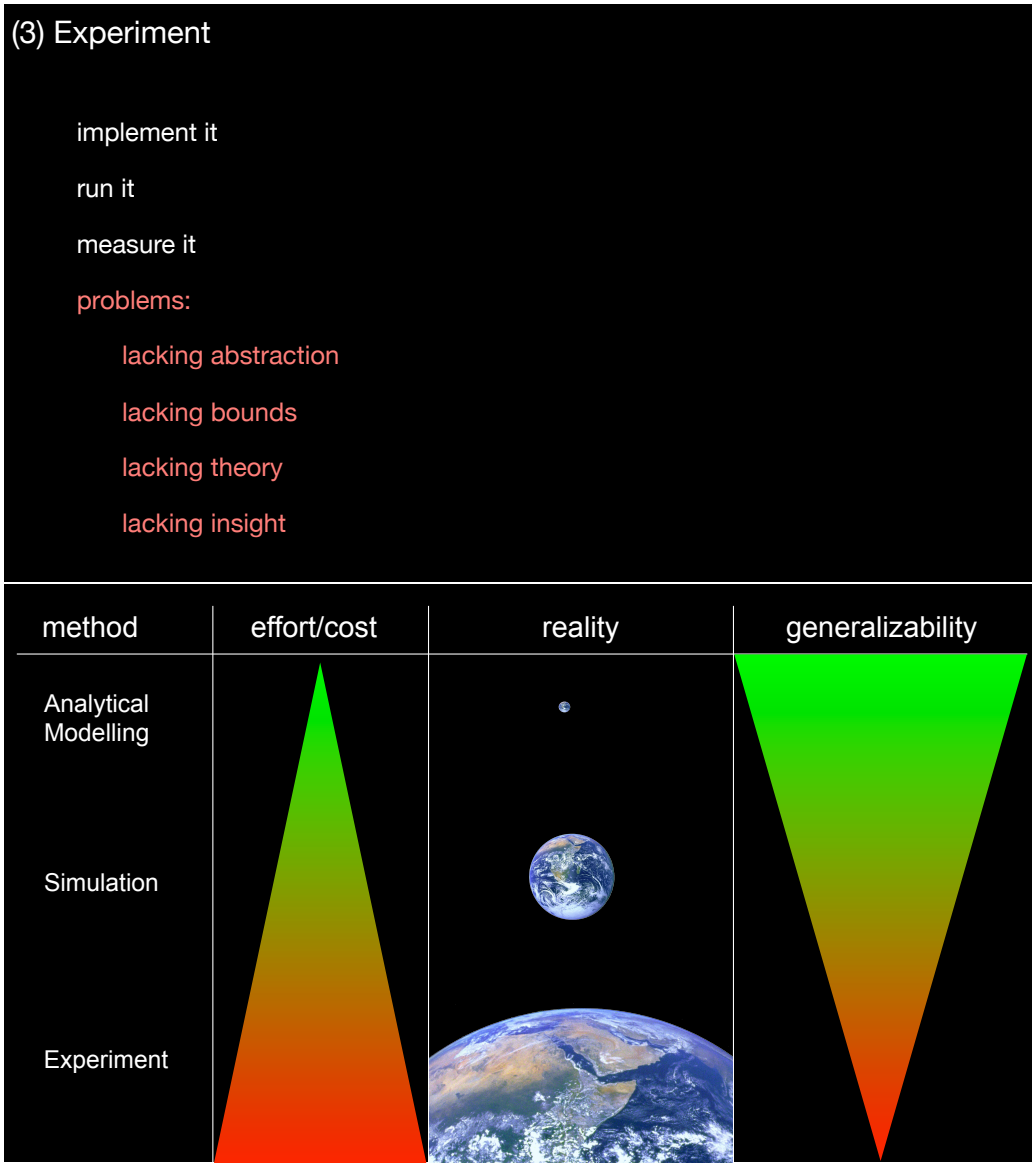


Figure 3.15: Experiments and the different trade-offs of modeling, simulation, and experiment

also useful if (parts of) the algorithm/system do not exists (yet). In these situations we can also use a blend of a simulation and an experiment where some parts of the system are simulated (by executing a model) and others are actually run (by executing the real system or algorithm).

What might be the problem of all of the former methods?

Analytical modeling, cost models, and simulations may be considerably oversimplifying reality. They may still imply useful bounds and predictions. However, these predictions may still be too far off from reality to be useful or informative. This is not saying that modeling is bad in general. This is saying that any form of modeling needs to consider how much that particular model is able to reflect reality anyway and whether the results

are strong enough to come up with meaningful decisions in practice.

What is an experiment?

experiment

In an experiment we run the actual algorithm and/or system.

What could be the problem of an experiment?

Experiments typically lack abstraction, i.e. the results observed from an experiment may only be valid for one dataset, on one machine, for one compiler, for one operating system, for one implementation for the system, and so forth. In other words, a possible problem of experiments is that they might be overfitting to a particular scenario. Therefore, we have to be careful when trying to generalize results from an experiment.

overfitting

How are the three ways to measure the performance of a computer program correlated to effort/cost, reality, and generalizability?

In general, *and with notable exceptions in all cases*, all methods have different trade-offs w.r.t. effort/cost, abstraction from reality, and generalizability, see Figure 3.15.

1. effort/costs: analytical modeling is the cheapest method with the least effort. In contrast, simulations are more complex. The most expensive method is an actual experiment (if we include the costs for creating the algorithm/system).

2. reality: how much do the different methods abstract from reality or in other words: how closely can a method predict the behavior of an algorithm/system in reality? Analytical modeling is the most abstract method, a simulation is somewhat less abstract. Experiments are the least abstract method — they are "down-to-earth".

3. generalizability: this can be considered the inverse of overfitting. Obviously, the more abstract a method the higher the generalizability. Therefore, generalizability is high for analytical modeling, it is lower for simulations, and it it is typically low for experiments.

Again, this is a high-level view on the different trade-offs of the three principal methods in performance analysis. There are notable exceptions for all cases.

Quizzes

1. Dr. No has an algorithm that performs $n!/2$ operations in total, where n is the size of the input. He then runs his algorithm on an input set of size $n = 25$. If his computer can perform 10^9 operations per second, is Dr. No going to be happy when his algorithm finishes?

   (a) Definitely yes.

   (b) We have to ask his $grand^m children$.

2. Dr. No also has an algorithm for a certain problem that runs in $O(2^{n^{3/4}\log_2 n})$ time. The current best algorithm for that problem runs in $O(2^n)$ time. Assume that the multiplicative constants hidden by the O-notation are all 1. Theoretically speaking,

Dr. No's algorithm is a breakthrough. Practically speaking, at which input size $n > 3$ does Dr. No's algorithm start becoming relevant?

(a) 2

(b) 256

(c) 8192

(d) 16384

(e) 65536

3. Assume you have the following query: SELECT SUM(A) WHERE A BETWEEN a AND b, and you know that its selectivity is extremely low (many tuples qualify). What kind of gain do you expect to obtain using an index over column A for answering the query in contrast to answering the query by simply doing a linear scan?

   (a) Negligible gain

   (b) Considerable gain

4. Assume you have the following query: SELECT SUM(A) WHERE B BETWEEN a AND b, and you know that its selectivity is extremely low (many tuples qualify). What kind of gain do you expect to obtain using an index over column A for answering the query in contrast to answering the query by simply doing a linear scan?

   (a) Negligible gain

   (b) Considerable gain

5. You manage to parallelize a certain algorithm, yet, when using k threads you do not observe the desired k-fold increase in speed (execution time) w.r.t. the single-threaded version of the algorithm. What could be some of the factors that hinder the scaling of your parallel algorithm?

   (a) Cache misses

   (b) Remote memory accesses

   (c) Memory bandwidth

   (d) Room temperature

6. You manage to come up with an analytical model for the number of cache misses in a certain data structure. Moreover, you proved mathematically that your model is a tight lower bound, i.e. for all inputs you expect at least that many cache misses. In addition, you also validate your model using an experiment. However, in that experiment, you observe that there are sometimes less cache misses than your tight lower bound predicted. So in summary: the results from the experiment contradict your analytical model! What could be some explanations for the discrepancies between theory and practice?

(a) Your model does not consider the effect of memory hierarchy in its computations.

(b) Your model does not consider the effects of spatial locality.

(c) The processor of the machine has turbo boost enabled.

(d) The input is highly skewed and your model assumed perfect uniform distribution for the input.

3.4 Static Hashing, Array vs Hash, Collisions, Overflow Chains, Rehash

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Hash Functions
[LÖ09], Hash-based Indexing
[CSRL09], Section 11
[RG03], Section 8.3.1 and 11.1
[RAD15]

Further Reading:
W hashing comparison
W locality-sensitive hashing

Learning Goals and Content Summary

Why do some people say that the three most important techniques in computer science are hashing, hashing, and hashing?

hashing

Hashing is a central method in various areas of computer science with several important applications. Applications include checksumming, duplicate detection, similarity search, encryption, and pseudo-randomization (e.g. pseudo-random number generators). In the context of databases, hashing is a building block for query processing in particular for point queries, but it can also be used to a limited degree for range queries. Further applications of hashing in databases include join processing (see Section 4.1.2) and grouping (see Section 4.2).

What is the core idea of hashing?

The core idea of hashing is to map keys from a relatively large input domain $D_I = \{0, \ldots, N\}$ to a smaller output domain $D_O = \{0, \ldots, M\}$ where $M << N$. Once the keys have been mapped, they may be stored in very compact data structures, e.g. an array with only M slots. The mapping is done using a hash function $h() : D_I \mapsto D_O$. A design goal of most hash functions is to redistribute, i.e. decluster keys from D_I uniformly over D_O. However, hash functions may also be used to cluster similar elements like in locality-sensitive hashing (LSH) [GIM99].

hash function

decluster

cluster

locality-sensitive hashing

LSH

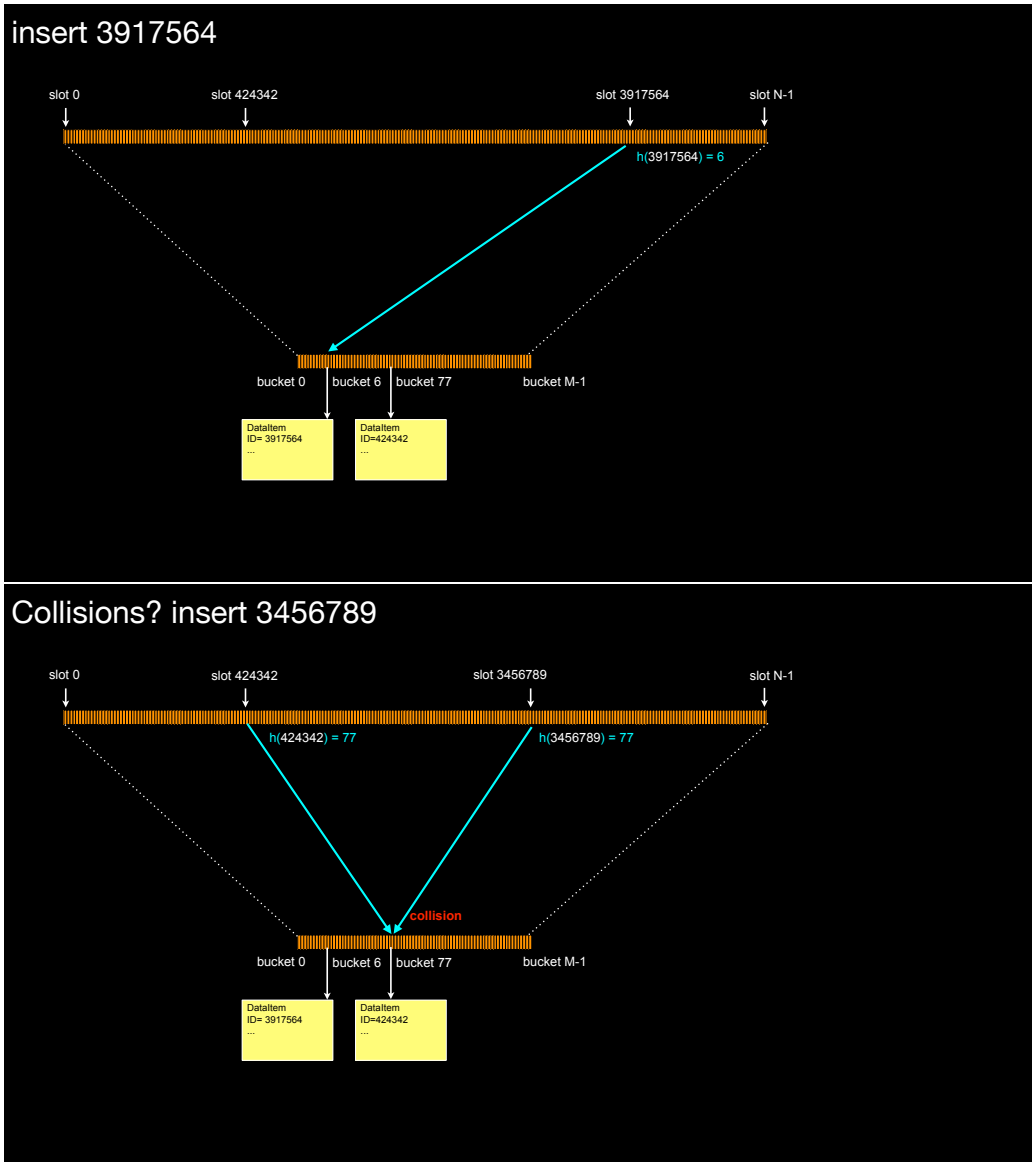


Figure 3.16: Inserting into a hash table may lead to collisions

How is a hash table organized?

There are hundreds of different hashing methods which vary in the hashing scheme they use, e.g. one vs multiple arrays, chained lists vs open addressing. They may also vary in the type of hash functions they use, e.g. multiplicative, Murmur. For an overview see the book by Cormen [CSRL09] or in the context of data management our own survey on hashing [RAD15]. In the video and *in the following discussion we will focus on chained hashing*. It is widely used in practice and also implemented in software libraries like STL and boost.

chained hashing

In chained hashing the hash table is simply an array of M slots. Each slot points to a (possibly empty) list of entries pertaining to this slot. Whenever we want to insert

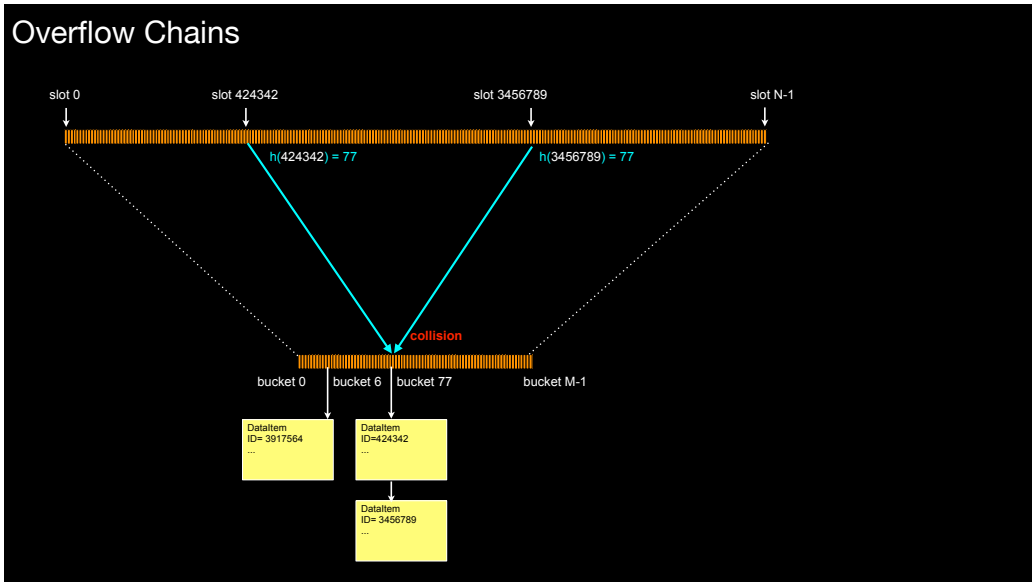


Figure 3.17: Handling collisions using overflow chains

a $(key, value)$-pair into the hash table, we compute the hash value of the key, i.e. $hv = h(key)$. Then we append $(key, value)$ to the list at slot hv. Notice that slot and bucket are used synonymously. In order to lookup entries in a chained hash table having a particular key, again we compute $hv = h(key)$. Then, we perform a linear search in the list found at hash bucket hv. It is important to perform this post-filtering step as, due to collisions, the different entries found in the list at slot hv may have different keys.

bucket

Notice that chained hashing can be outperformed by an order of magnitude by other hashing methods [RAD15].

What is the runtime complexity of hashing?

For chained hashing, looking up a particular slot is on the order of $O(1)$. Finding a particular item in a chain depends on the number of elements in the chain found at that slot. For a slot of length k it is on the order of $O(k)$. In general, many parameters affect the runtime of hashing. For instance, the execution costs of the hash function and the load factor of the hash table. Hashing is yet another good example of a method where a performance analysis solely based on runtime complexity can be highly misleading, see [RAD15].

What is a collision?

collision

A collision occurs when two different keys are mapped to the same value. In other words, given a hash function $h() : D_I \mapsto D_O$ and two keys $k_1, k_2 \in D_I$ where $k_1 \neq k_2$. If $h(k_1) = h(k_2)$, we call this a collision: "keys k_1 and k_2 collide in the same slot". A good hash function should create as little collisions as possible. But obviously, as D_I is larger than $D_O \Rightarrow \exists k_1, k_2 \in D_I$ where $h(k_1) = h(k_2)$. In other words, even if D_I is larger than D_O by only a single element, a collision is inescapable. See also Figure 3.17 for an example.

How can we handle collisions?

In chained hashing each slot stores a pointer to a list of values hashing to that slot. Hence, collisions are handled by appending them to the list available at the bucket pointed to.

rehash

Why would we rehash?

In general, we rehash if we need to increase (or decrease) the hash table. This may be suitable in cases where many slots are non-empty and/or the chains pointed to get to long (which deteriorates search time). Yet, rehashing is typically relatively expensive, i.e. we have to insert all existing entries into a new hash table (of larger size). This has costs on the order of $O(n)$. This investment should be amortized over several future inserts and/or lookups.

So, now I know enough about hashing?

Most probably not. This video and this chapter are just a very gentle introduction to the core ideas of hashing. The different hashing techniques are a topic in itself and typically part of separate textbooks on algorithms and data structures or taught in undergrad courses. Make sure you read the book by Cormen et.al. [CSRL09]. For a performance shootout of different hash tables in the context of query processing see [RAD15].

Quizzes

1. You have to make a decision about what data structure to use for a workload heavily based on range queries. You have no other knowledge about this workload. Which data structure would you use in general?
   (a) Hash table
   (b) B-tree
   (c) We should rather use table scan, than a data structure
2. Is it true that using hash functions over primary key attributes, i.e. no duplicates, results in a collision-free hash table?
   (a) Yes
   (b) No

Exercise

Let's assume a hashing-method that does not keep a list (an overflow chain) for each bucket like in the video, but rather keeps all entries in a single array of size m where each array slot corresponds to a hash bucket. If at any time we try to insert an element x into this hash-table and it turns out that the slot hashed to, say slot $0 \leq i < m$ is already occupied, we inspect slots $i+1$ through $m-1$ and then 0 through $i-1$ until we find an empty slot to insert x to.

(a) Provide pseudocode for `remove(int key)`.

(b) Provide pseudocode for `get(int key)`

(c) Discuss: what are possible problems of this approach compared to hashing with overflow-chains?

(d) Assume you change your insert strategy as follows: Given a function

$$D(z) := \begin{cases} i - h(z) & i \geq h(z), \\ i - h(z) + m. & i < h(z). \end{cases}$$

This function computes the distance of an element z from its ideal hash bucket. Now, whenever during an insert of key x you inspect a bucket i that already contains a key y and $D(y) < D(x)$, then you insert x into that bucket and keep on probing the next bucket using y. Discuss: what are possible advantages of this change?

3.5 Bitmaps

3.5.1 Value Bitmaps

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	Further Reading:
[LÖ09], Bitmap Index	[LÖ09], Bitmap-based Index Structures

Learning Goals and Content Summary

What is the core idea of a bitmap (aka value bitmap)?

bitmap

value bitmap

Assume a table `foo` and a column `foo.c`. Further assume that `foo` contains $|\texttt{foo}| = N$ rows and column `foo.c` has D distinct entries. The core idea of a bitmap is to keep one bitlist for each of the D distinct values in `foo.c` (D is called the cardinality of that column). Each bitlist has N bits. Thus, in total, we need $D \cdot N$ bits for the entire bitmap. Each bitlist D_i represents one distinct data value i of column `foo.c`. If $D_i[x]$ is true, this means that the data value of the xth row in column `foo.c` is i. If $D_i[x]$ is false, this means that the data value of the xth row in column `foo.c` is not equal i. Obviously, if each entry in column `foo.c` may only represent a single value (i.e. we follow the first normal form), it holds $\forall_x$ if $\exists y$ where $D_i[y] =$ true, then $\forall_{z \neq y} D_i[z] =$ false, i.e. for a particular row x only one of the bits in the different bitlists may be set.

bitlist

What is the size of an uncompressed bitmap?

uncompressed bitmap

The size of an uncompressed bitmap is $D \cdot N$ bits. Hence, for large D and/or N the bitmap may become too large to be suitable in practice.

What are typical bitmap operations and why are they very efficient?

Typical bitmap operations evaluate a boolean condition on the different bitlists belonging

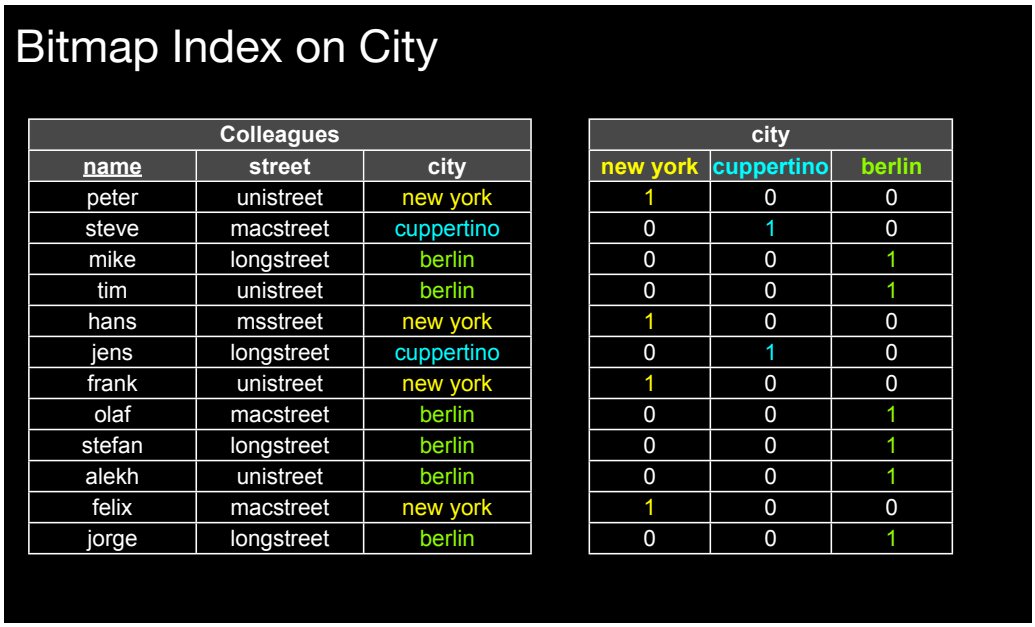


Figure 3.18: A (value) bitmap on attribute `city` of table `Colleagues`

to the bitmap. For instance, in the video on table `colleagues` we create a bitmap on column `colleagues.city`. Now, if we search for all rows where the `colleagues.city` is equal to "berlin" we may simply inspect bitlist D_{berlin} which already marks the qualifying rows. Similarly, conjuncts like `colleagues.city` is equal to "berlin" or `colleagues.city` is equal to "new york" may be evaluated by ORing bitlists D_{berlin} and $D_{\text{new york}}$. If bitmaps were created for multiple columns on that table, say `colleagues.city` and `colleagues.street`, we may also evaluate conditions like `colleagues.city` is equal to "berlin" AND `colleagues.street` is equal to "long street" by ANDing bitlist D_{berlin} of column `colleagues.city` with bitlist $D_{\text{long street}}$ of column `colleagues.street`.

How is the cardinality of a column related to the size of the bitmap?

The cardinality D of the column directly affects the total size of the bitmap index $D \cdot N$. Recall, that we do not need to create bitlists for values that are not represented in the column, i.e. we do not need to create a bitlist for a value i where $\forall_x D_i[x] = \text{false}$.

What are applications of bitmaps?

In general, bitmaps are useful for large datasets where queries contain several complex boolean conditions. Bitmaps are often used in Data Warehousing and OLAP. Bitmaps may also be used across different tables as a bitmap join index.

bitmap join index

Quizzes

1. What is the smallest addressable unit of memory in a computer?
   (a) One bit
   (b) One byte
   (c) One megabyte

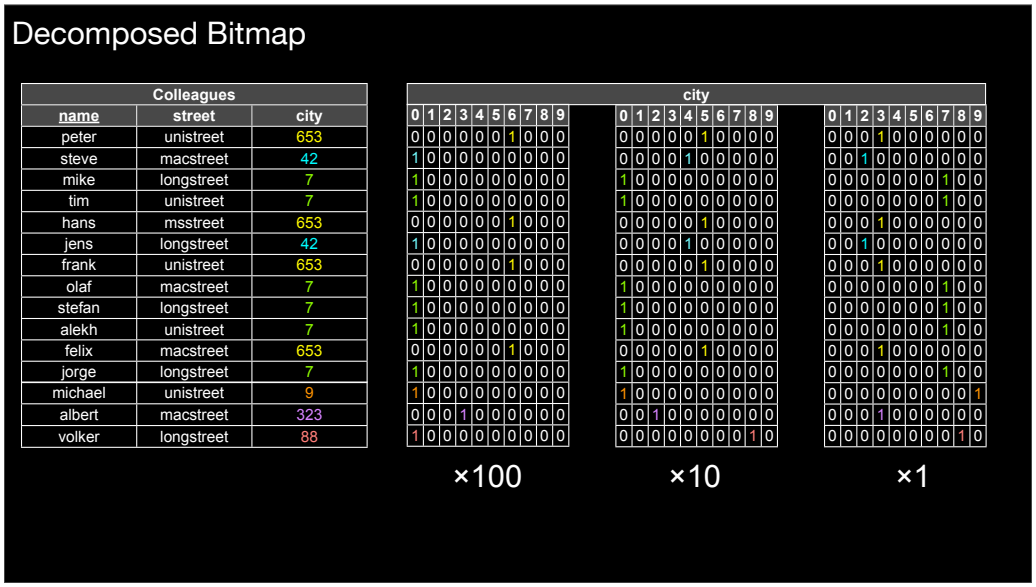


Figure 3.19: A decomposed (value) bitmap on attribute `city` of table `Colleagues`

2. One of the reasons for using bitmaps is to save space. How would you represent (in a computer) a bitmap for $N > 0$ elements?

   (a) Array of N of type boolean (in C++ or Java)

   (b) Array of N of type character

   (c) Array of $\lceil N/8 \rceil$ characters, assuming the size of the character type used is 8 bits

3.5.2 Decomposed Bitmaps

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

What is the core idea of a decomposed bitmap? **decomposed bitmap**

In a value bitmap, as explained above, we learned that a single value is represented by one particular bitlist, i.e., if we want to know whether for row x the value of an attribute is v, we need to check whether $D_v[x]$ is set to true. In contrast, in a decomposed bitmap, a value v is represented by a linear combination of bitlists, i.e., if we want to know whether for row x the value of an attribute is set to v, we need to check whether a predicate $\mathrm{p}(x, v)$ is set to true. Here we assume that $\mathrm{p}(x, v)$ is a linear combination of multiple bitlists. The decomposed bitmap variant explained in the video is defined for values in range $[0; 1000[$, it uses 10 bitlists for each of the three decimals 10^0, 10^1, and 10^2, i.e. 30 bitlists in total. Assume that v_j for $0 \leq j \leq 2$ returns the jth decimal of v, i.e. $v_j = (v \bmod 10^{j+1}) \operatorname{div} 10^j$.

Let's further assume that a bitlist of decimal j is referred to as $D_{j,v}[x]$. Then, we can use the following predicate:

$$\mathrm{p}(x, v) = D_{2,v_2[x]} \text{ AND } D_{1,v_1[x]} \text{ AND } D_{0,v_0[x]}.$$

For instance, in the example in the video, in order to compute the rows where `city`=653, we need to inspect at most three bitlists $D_{2,6}[x]$, $D_{1,5}[x]$, and $D_{0,3}[x]$. Only if all three bits are set, we know that in row x the value v is equal to 653.

What does this imply for bitmap operations?

In order to compute $\mathrm{p}(x, v)$ we may need to inspect multiple bitlists. Hence, we may need more I/O at query time.

What do we gain in terms of storage space?

Depending on the cardinality D of the column to index, we may need less space. Consider again an uncompressed bitmap with D different values and N different rows. For such an uncompressed bitmap we require $D \cdot N$ bits. In contrast, for a decomposed bitmap we require:

$$\lceil \log_{10}(D) \rceil \cdot 10 \cdot N \text{ bits.}$$

For an arbitrary base b, we require:

$$\lceil \log_{b}(D) \rceil \cdot b \cdot N \text{ bits.}$$

Obviously, an uncompressed bitmap can be seen as an extreme case where $b = D$:

$$\lceil \log_{D}(D) \rceil \cdot D \cdot N \text{ bits} = D \cdot N \text{ bits.}$$

Notice that this computation also holds for cases where the domain is not dense. For instance, et's assume that the domain contains only three values 42, 578, and 653 only. In this case we still have a cardinality of $D = 3$. However, we need an extra remapping step (through a dictionary) which remaps these three values to 0, 1, and 2. The latter, dense, value sequence can be efficiently represented by a decomposed bitmap, the former, sparse, value sequence 42, 578, and 653 not.

In order to determine whether a decompressed bitmap needs less storage than an uncompressed bitmap, we need to resolve:

$$\lceil \log_{b}(D) \rceil \cdot b \cdot N < D \cdot N \Leftrightarrow \lceil \log_{b}(D) \rceil \cdot b < D.$$

In other words, if the number of digits required to represent the domain times the base is smaller than the domain, a decomposed bitmap pays off storage-wise.

OK, but why not use domain encoding anyways? Then we will always win in terms of storage space! Domain encoding is less efficient!

This is true in terms of the compression ratio, however, the benefit of bitmap decomposition is not only the compression ratio. It is true that domain encoding requires $\lceil \log_{2}(D) \rceil$

bits which is more than the $B \times \lceil \log_B(D) \rceil$ bits (actually the leading group does not need all B bitlists in the general case), B being the base of the decomposition, required by bitmap decomposition. So, in terms of compression ration, domain encoding clearly wins. However, the goal here is to strike a balance between bitmap-style access at query time and compression ratio. The encoding used in bitmap decomposition is also called a K-of-N encoding [WLO+85]. This means, for each group of bitlists, exactly K out of N bits are set, in this case $K = 1$ and $N = B$. This implies that for the example used in the video, where $D = 1000$, $B = 10$, and hence three groups of 10 bitlists each, for any point query, we only need to scan three bitlists in order to identify the qualifying rows. In contrast, in domain encoding, we would need to scan all bits (10 in this case), as we cannot make assumptions on how many (and which) of the other bits are set. For large tables, and depending on the parameters, this may make quite a difference. For the example this is already a factor 3.3 in terms of I/O (or memory bandwidth).

Quizzes

1. Assume there is an attribute in a table with n tuples that you know can contain 1000 different values. However, only 15 different values appear in that column. What is the space requirement in bytes of that attribute if you use regular bitmaps to represent it?

   (a) $n \cdot 1000 \cdot sizeof(int) = n \cdot 1000 \cdot 4$ bytes

   (b) $n \cdot 15 \cdot sizeof(int) = n \cdot 15 \cdot 4$ bytes

   (c) $\lceil n/8 \rceil \cdot 15 \cdot sizeof(char) = \lceil n/8 \rceil \cdot 15$ bytes

3.5.3 Word-Aligned Hybrid Bitmaps (WAH)

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	Further Reading:
W javaewah	[WOS06]

Learning Goals and Content Summary

What is the relationship of WAH to RLE and 7-Bit encoding?

RLE

7-Bit

WAH can be considered a combination of two methods: (1) run-length encoding, see Section 2.4.3, and (2) x-Bit encoding, where $x = (c \cdot 8) - 1, c \geq 1$, see Section 2.4.4.

How does word-aligned hybrid bitmap (WAH) compression work?

word-aligned hybrid bitmap

WAH

compress

Consider an uncompressed input stream of x bits. In order to compress that stream, we segment the input stream into rows (aka chunks) of x =63 bits. Again, as in 7-Bit encoding, x may be changed to $x = (c \cdot 8) - 1, c \geq 1$. We simply need one bit less than

the number of bits in the segment to make this method work.

fill word

Now, in the compressed output stream, we represent adjacent rows containing only 0s or only 1s by a special fill word of 64 bits. For instance, if two adjacent input rows of 63 bits contain only 0s, we may compress these two rows to a single fill word of 64 bits. In other words, rather than storing $2 \cdot 63$ bits, we only store 64 bits. We set the additional leading bit of the 64 bit output word to mark that this is a fill word, i.e. the leading bit 63 is set to true. In addition, bit 62 contains the fill pattern $fp \in \{0, 1\}$ that is represented by this fill word. In the example, as both rows contain only 0s, we set bit 62 to 0. The remaining bits of the 64 bit output word contain the number of repetitions of the pattern contained in bit 62 (the unit for counting is the number of 63 bit input words, not the number of bits in the input). In the example, bits 0 to 61 are set to $10_2 = 2_{10}$.

literal word

All other input rows, i.e. rows that do not only contain 0s or 1s, are represented by a literal word of 64 bits. This means, we take an input row of 63 bits and add an additional bit in front which is set to 0. The resulting 64 bit word is written to the output.

decompress

How do we decompress?

Decompression is straightforward. Consider a WAH-compressed input stream of n 64 bit words. Recall, that each of the input words must either be a literal or a fill word. We process each input word one by one: if the leading bit of an input word is set to 0, i.e. it is a literal word, we append the remaining 63 bits to the output. Otherwise, if the leading bit of an input word is set to 1, i.e. it is a fill word, we append #repetitions·63 times the fill pattern $fp \in \{0, 1\}$ to the output.

How could WAH be improved?

WAH may be improved in many ways:

1. the fill pattern may be more complex, i.e. rather than using a single bit for the fill pattern (bit 62) we may use multiple bits to represent more complex, e.g. alternating patterns. In turn, fewer repetitions may be represented by a fill word.
2. a fill pattern only makes sense in case of repetitions of adjacent rows. A 63 bit input word containing only 0s or 1s may also be represented by a literal without gaining or losing anything. Hence, in the fill word we should store the number of repetitions as #repetitions - 2. Like that we increase the maximum number of repetitions that may be stored in a fill word from $2^{62} - 1$ to $2^{62} + 1$.
3. we may parallelize the method easily. For both compression and decompression, we may simply chunk the (uncompressed or compressed) input stream and treat each chunk by a separate thread.

See also [WOS06] for a discussion of possible improvements and variants of WAH.

Quizzes

1. Do word-aligned hybrid bitmaps always take up less space than original bitmaps for the same data set?

(a) yes

(b) no

2. Suppose we have a 93-bit long bit list, only the first bit is set to 0 (all other bits are set to 1), how many bits can you save if word-aligned hybrid bitmap is applied (the word size is 32 bits)?

   (a) _____

3. Assume we created a WAH bitmap of five 16-bit words where the signal bits of those 5 words are 0, 1, 0, 1, 0. What is the largest possible uncompressed bitmap size in bits we can get if we uncompress these 5 words assuming the word size is 16 bits?

   (a) 491535

   (b) 491505

   (c) 491520

   (d) 491538

Exercise

Assume a table with 992000 rows and a column with 992 different values. You want to index this column using a bitmap index. Assume further that your word size is 32 bit.

(a) How many bits do you need to store the bitmap index uncompressed?

(b) What is the worst-case size of any WAH compressed bitlist of this bitmap?

(c) Assume now that the values are distributed round-robin. How many words do you need to store all WAH compressed bitlists of this bitmap?

(d) What is the best-case distribution of the values w.r.t. compressed size? How many words do you need to store all WAH compressed bitlists of this bitmap?

3.5.4 Range-Encoded Bitmaps

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

range-encoded bitmap

What is the major idea of a range-encoded bitmap?

The major idea of a range-encoded bitmap is to change the semantics of the bitlists. Recall the semantics of a value bitmap as explained in Section 3.5.1. In a value bitmap on column `foo`, if for row x the bitlist $D_v[x]$ is set, this means that the attribute value of column `foo` in row x **is equal** to value v. In a range-encoded bitmap this is changed. Here, we keep bitlists of type $D_{\leq v}[x]$. The semantics is changed to: If for row x the

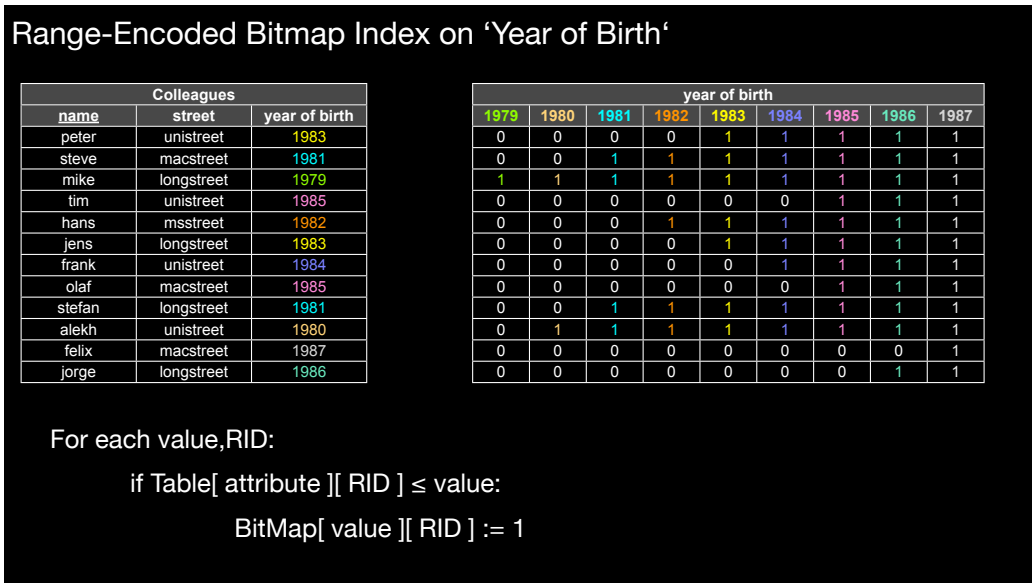


Figure 3.20: A range-encoded bitmap (value) bitmap on attribute `year of birth` of table `Colleagues`

bitlist $D_{\leq v}[x]$ is set, this means that the attribute value of column `foo` in row x **is less or equal to** value v.

How many bitlists do we need to represent a range-encoded bitmap?

A range-encoded bitmap with a cardinality of D may be represented by $D-1$ different bitlists. This is because, by definition, for the maximum value of D, let's call it $v_{max} = \max(D)$, its corresponding bitlist $D_{\leq v_{max}}[x]$ is set for all x. Hence, $D_{\leq v_{max}}[x]$ is redundant and we do not have to represent it explicitly.

What are the space requirements when compared to the standard uncompressed value bitmap?

If your table has n tuples and the cardinality of the column to index is D, we need $(D-1) \cdot n$ bits. So, the difference to a value bitmap is that we need n bits less.

How do we compute a range query?

A range query requesting all rows having discrete values in range $[v_1; v_2]$ may be computed by returning

$$D_{\leq v_2}[x] \text{ AND NOT } D_{\leq v_1 - 1}[x].$$

In other words, we compute all values smaller equal v_2 that are not smaller than v_1. Notice that we only require two bitlists for any range query rather than a possibly large set of up to D bitlists. For instance, to compute all rows having values in range $[1980; 1984]$ we return:

$$D_{\leq 1984}[x] \text{ AND NOT } D_{\leq 1979}[x].$$

point query

Can we also execute a point query?

Sure. A point query on value v can be translated to a range query $[v; v]$ which is then executed as described above. For instance, to compute all rows having a value of 1983,

we translate it to a range $[1983; 1983]$ and hence we return:

$$D_{\leq 1983}[x] \text{ AND NOT } D_{\leq 1982}[x].$$

What is the major trade-off of range-encoded bitmaps compared to value bitmaps?

The advantage of a range-encoded bitmap is that it allows us to execute any range query by inspecting only two bitlists. This is not possible with a value bitmap which needs up to D bitlists to compute the same result. An advantage of value bitmaps over range-encoded bitmaps is that any point query may be computed using a single bitlist. In contrast, a range-encoded bitmap requires two bitlists, even for point queries. In terms of storage requirements, both indexes need more or less the same space: $(D-1) \cdot n$ versus $D \cdot n$ bits.

Quizzes

1. For the Range-Encoded Bitmaps example in the video (assuming that the bit list with the highest value does not have to be represented physically, yet we memorize its key), how many bit lists have to be read to answer a query with the following WHERE clause: WHERE yearOfBirth = 1987?

   (a) 1

   (b) 2

   (c) 3

3.5.5 Approximate Bitmaps, Bloom Filters

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Bloom Filters
W 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:

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

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

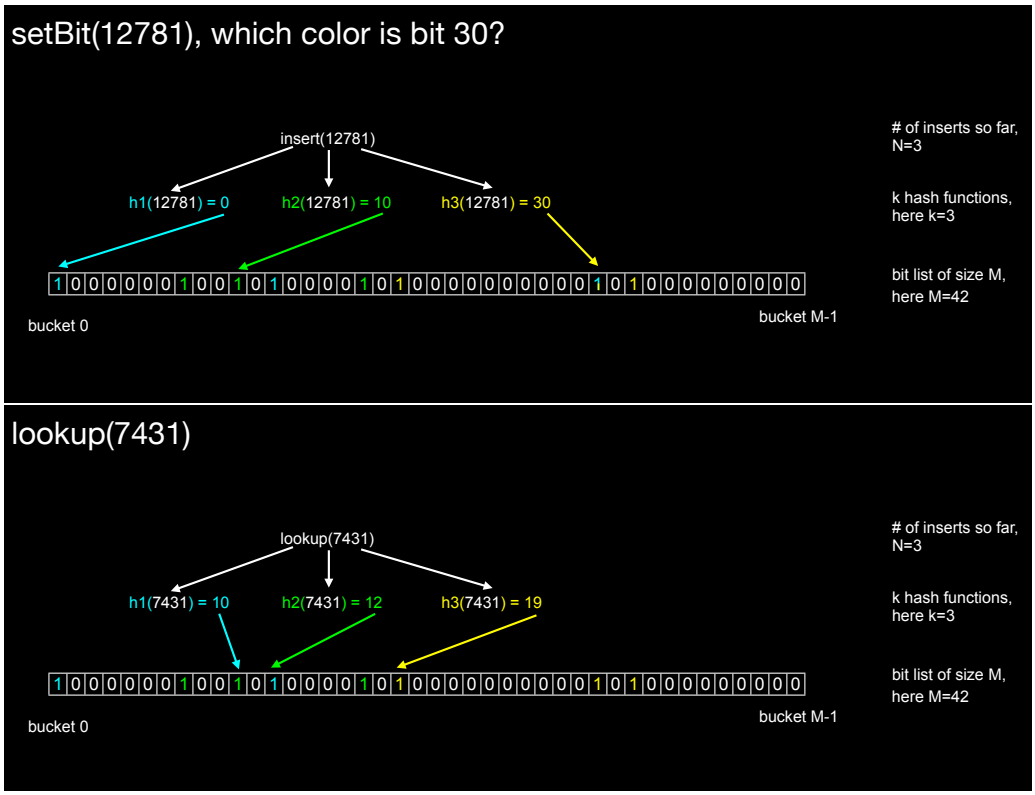


Figure 3.21: setting and looking up bits in a bloom filter

weakened to:

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

This implies there are cases where

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

false positive

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),

filter index

is a 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 a series of hash functions which all operate on the same bitlist. Similarly to chained hashing, we map an input domain of keys $D_I = \{0, \ldots, N\}$ to a smaller output domain $D_O = \{0, \ldots, M\}$ where $M << N$. In contrast to chained hashing, we do not store key/value-pairs in the hash table, but just set bits at particular slots, i.e. the slots do not point to a list of entries. In a bloom filter, a slot is simply implemented as a bit. Hence, we need a bitlist of length M. Also in contrast to chained hashing which uses

one hash function, in a bloom filter we may use multiple hash functions $h_1() : D_I \mapsto D_O, \ldots, h_k() : D_I \mapsto D_O$.

hash function

How do we insert a key into a bloom filter?

For each of the k hash functions we set slot $h_i(key)$ to `true`. This is done independently of whether a particular bit of a slot was already set before.

How do we lookup a key in a bloom filter?

For each of the k hash functions we check whether slot $h_i(key)$ is set to `true`. Only if all slots are set to `true` we return true. This implies that as soon as we detect a slot that is set to `false`, we can stop inspecting other slots, i.e. we terminate early, and return `false`.

Why would a bloom filter return a false positive?

The reason is the same as in chained hashing: collisions. Assume that we call index.hasKey(key) which returns `true`. Further assume, that key is not in the table. Hence, key is a false positive. How is that possible? Simply because the k bits inspected by index.hasKey(key) where not set through a previous insert of key into the bloom filter, but rather by insert-operations of other *keys* which happened to set those bits. This is a collision.

collision

What is the optimal number of hash functions?

Analytically, the optimal number of hash functions is

$$k = m/n \cdot \ln(2).$$

However, in practice, k needs to be an integer. This yields a false positive probability rate of

$$p(\text{“false positive”}) \approx 0.618503^{m/n}.$$

Quizzes

1. In the case of bloom filters, as the number of stored items increases, the false positive rate:
   (a) Decreases
   (b) Increases
   (c) False positive rate is independent of the number of items stored
2. While removing an entry from a bloom filter:
   (a) All the bits referenced by this item have to be set to 0
   (b) Setting only one bit to 0 would suffice (since a look-up for an item returns false even if a single referenced bit is 0)
   (c) In general, it is not possible to remove an entry from a bloom filter.
3. The False positive rate in bloom filters is decreased by:
   (a) Increasing the length of the bit list

(b) Decreasing the number of hash functions

4. In the context of databases, bloom filters are ideal for checking:

   (a) Whether an entry exists in a table

   (b) Whether an entry does not exist in a table

   (c) The number of times an entry appears in a table

Exercise

Your task is to design a software that is able to answer the following question: given a matriculation number consisting of 7 digits, decide whether the student is enrolled in the university. To make this process faster, you can create a Bloom filter on the matriculation numbers, and store it in main memory. Let's assume the university has $n = 20,000$ students.

Let's say you have $m = 200,000$ bits to store your Bloom filter, and 14 hash functions (k).

(a) What is the false-positive rate of your Bloom filter?

(b) Let's assume the average computation time for a hash function is 500 ns and the average random memory access time is 100 ns. Assume the worst case that the bloom filter is not in any of the caches. What is the cost of a lookup in the Bloom filter for an element that is contained?

(c) In case of a positive answer from the bloom filter, you still have to go to disk, and check whether the student is really enrolled. This triggers an additional lookup with an average lookup time for a single table entry of 5 ms.

Compare the expected costs of a random lookup: a disk-lookup (without using the bloom filter) vs a lookup using the Bloom filter as well (and going to disk on demand only)!

(d) How could you improve the parameters of your Bloom filter, i.e. k?

Exercise

The bloom filters introduced above have a big disadvantage: they do not support delete operations. You are given two hash functions:

1. $h_1(x) = x \mod 8$
2. $h_2(x) = (x/4) \oplus 3 \mod 8$

Where $\oplus$ denotes the binary XOR operation. These two hash functions are used to build a numbering bloom filter. In a numbering bloom filter you use more than a single bit for every bucket. Whenever a key is inserted into the numbering bloom filter, the counts of

all buckets that are hit are incremented by one (except if it would overflow). Assume you have 8 bytes available to store the entire numbering bloom filter and that it is initially empty (i.e. all counts are zero, bucket size is 1 Byte).

(a) Insert 123, 345, 3, 77, and 5. Draw the resulting filter after each insert.

(b) Can you perform deletes in such a data structure? If it is possible, explain under what circumstances. Delete 345 if possible.

(c) Assume the numbering bloom filter contains only the elements 345, 5, 6, and 7. What is the *empirical* false positive rate of the Numbering bloom filter w.r.t. the interval [0;8]? Hint: Use the concrete hash functions as they are specified in the beginning.

Exercise

Similar to bloom filters, in a numbering bloom filter we need to make the right choices for k, m as well as the number of bits s to use for each bucket.

In the exercise you have to plot several graphs. A free tool to create plots is gnuplot; an alternative to this is R. Assume you want to insert $n = 100,000$ elements into your bloom filter.

(a) Plot the false-positive rate for $k \in [1, 4]$ hash function against the number of bits m. For every k, what is the smallest bitmap size m to get a false-positive rate smaller than 1%?

(b) Assume you use $k = 2$ hash functions and you have $M = 362,880$ bits available to store your Bloom filter.

(i) What is the highest number of collisions of any bucket? Assume a varying number of bits per bucket where the number of bits per bucket is varied in $s = 1, \ldots, 10$. Hence the actual number of buckets of the numbering bloom filter is M/s. Either find an analytical solution for the expected value or run an experiment with sufficient (e.g. 1000) repetitions to empirically determine the expected value.

(ii) Now also plot the false positive rate against the different bucket sizes.

(iii) What is the bucket size in $s = 1, \ldots, 10$ that allows you to represent two times the number of expected collisions and still that has the lowest false positive rate?

Chapter 4

Query Processing Algorithms

4.1 Join Algorithms

4.1.1 Applications of Join Algorithms, Nested-Loop Join, Index Nested-Loop Join

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Nested Loop Join
[LÖ09], Index Join
[RG03], Section 14.4.1

Learning Goals and Content Summary

join algorithm

Why are join algorithms important?

Join algorithms are a fundamental building block of query processing. These algorithms are used internally by the database system not only to translate SQL-joins into executable programs, but also to implement grouping and aggregation (including duplicate elimination), as well as subqueries.

What is the possible impact of joins on query performance?

The impact of join algorithms on query performance may be tremendous. In particular, in those cases where the database system chooses the wrong join algorithm, e.g. nested loops for large input datasets, the impact on runtime may be substantial and lead to several orders of magnitude performance differences.

What are the four principal classes of join algorithms?

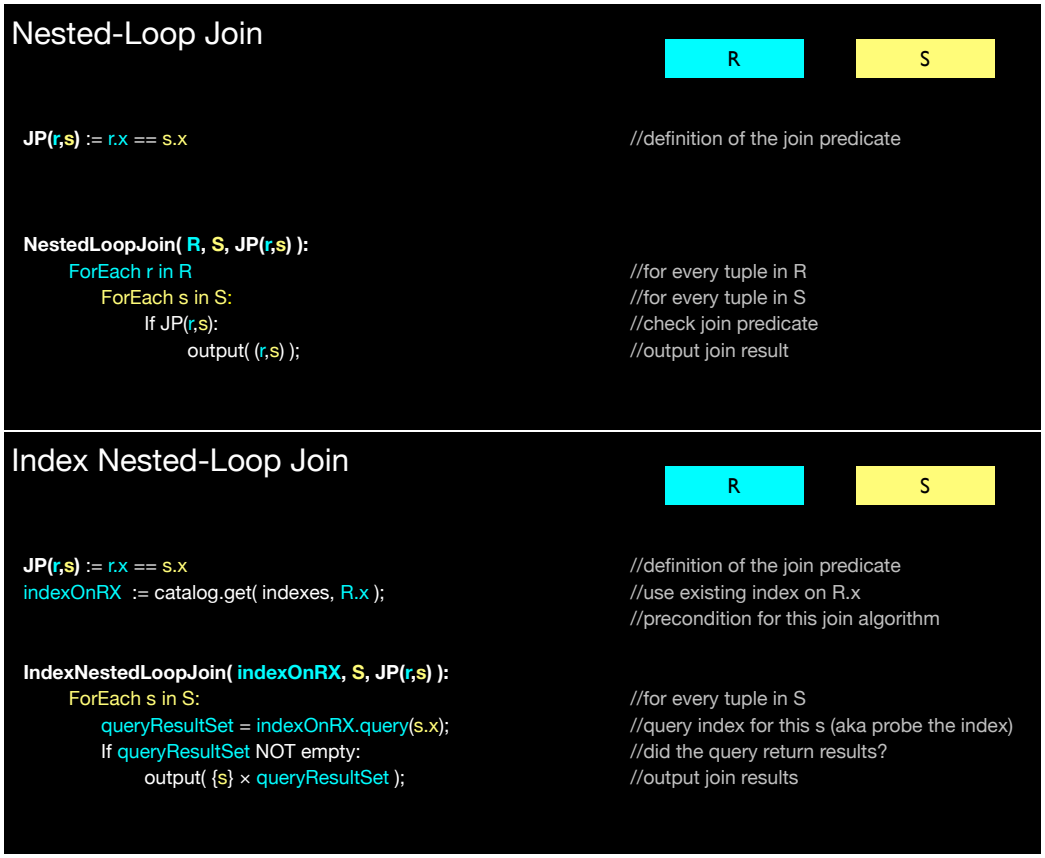


Figure 4.1: Nested-Loop vs Index Nested-Loop Join

1. nested-loop joins
2. index nested-loop joins
3. hash joins
4. sort-merge joins

This is the traditional classification of joins that you find in textbooks. However, note that this classification is somewhat misleading. As hash joins can be considered a special case of index nested-loop joins.

nested-loop join

NL

What is a nested-loop join (NL)?

Let's assume two relations R and S and a join predicate JP(r,s):= r.x==s.x. A nested-loop join simply implements the cross product of R and S followed by a selection, i.e.

$$\sigma_{\mathrm{JP}}(\mathrm{R} \times \mathrm{S}).$$

This is implemented by nesting two loops, one is iterating over all elements of R, the other over all elements of S. Every possible combination of elements from R and S is created and probed with the join predicate. Obviously, this naive implementation has a worst-case complexity of $O(|\mathrm{R}| \times |\mathrm{S}|)$. Hence, NL should only be used for:

1. small input sets (i.e., datasets that are small after having applied individual filters on the input datasets),
2. cases where no other join algorithm is available (e.g., complex join predicates, in particular join predicates that are a blackbox function),
3. for testing purposes to obtain baseline result sets.

For which type of join predicates does NL work?

This "algorithm" works for all possible join predicates.

index nested-loop join

What is index nested-loop join (INL)?

INL

In an index-nested loop join (INL) we replace *one of the loops* of a nested-loop join (NL) by an index. This works only for those cases where the join predicate can be evaluated efficiently by that index. In order to determine whether this is possible, we need to look inside the join predicate, i.e. we inspect the definition (and/or implementation) of the join predicate which may actually be a blackbox. In case of an equi join predicate like in the video, this is easy. Hence, we simply need an index structure supporting point queries, e.g. a B-tree index. In the video, we create an index on R and then probe all elements of S against R. Likewise, we could create an index on S and then probe all elements of R against S. INL is typically a good choice if one of the inputs R or S already has a suitable index available. Even if the index has to build first, INL may pay off.

What is required to run INL?

We need either an already existing index that was built on a suitable key such that we may exploit that existing index in INL. Or we may bulkload an empty index.

join predicate

For which type of join predicate does INL work?

This algorithm works for join predicates that may be translated into a sequence of index lookups. This is not only the case for equi joins. For instance, assume a join predicate JP(r,s):= abs(r.x-s.x) <= 42. This could be translated into a sequence of queries against an index on S.x as follows:

$$\forall r \in \mathrm{R} : \mathrm{indexOnSx.query}([r-42; r+42]).$$

In other words, for each element of R, we query the index on S.x using an interval query $[r-42; r+42]$.

What is the runtime complexity of INL?

The runtime complexity of INL depends on the complexity of an individual index lookup. In the case of a B-tree built on S where an individual insert and point query takes O(log |S|) time, the overall complexity is O(|R| log |S|) if we do not factor in the costs for building the index on S which costs O(|S| log |S|).

Quizzes

1. Which of the following relational operations can be performed with or as a side-effect of a join algorithm?
   (a) GROUP BY
   (b) INTERSECT
   (c) ORDER BY
2. Assume you have two tables R and S with $|R| = n$ and $|S| = m$. What is the worst-case complexity (number of comparisons) of a Nested-Loop Equi-Join on R and S?. You may assume that the predicate has $O(1)$ complexity
   (a) $O(n \log m)$
   (b) $O(n \cdot m)$
   (c) $O(n^2)$
   (d) $O(m^2)$
3. Would any kind of join predicate, regardless of how complicated it is, be suitable to implement the entire join with a Nested-Loop Join?
   (a) No
   (b) Yes
4. Would any kind of join predicate, regardless of how complicated it is, be suitable to implement the entire join with an Index Nested-Loop Join?
   (a) No
   (b) Yes

4.1.2 Simple Hash Join

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Hash Join
[RG03], Section 14.4.3

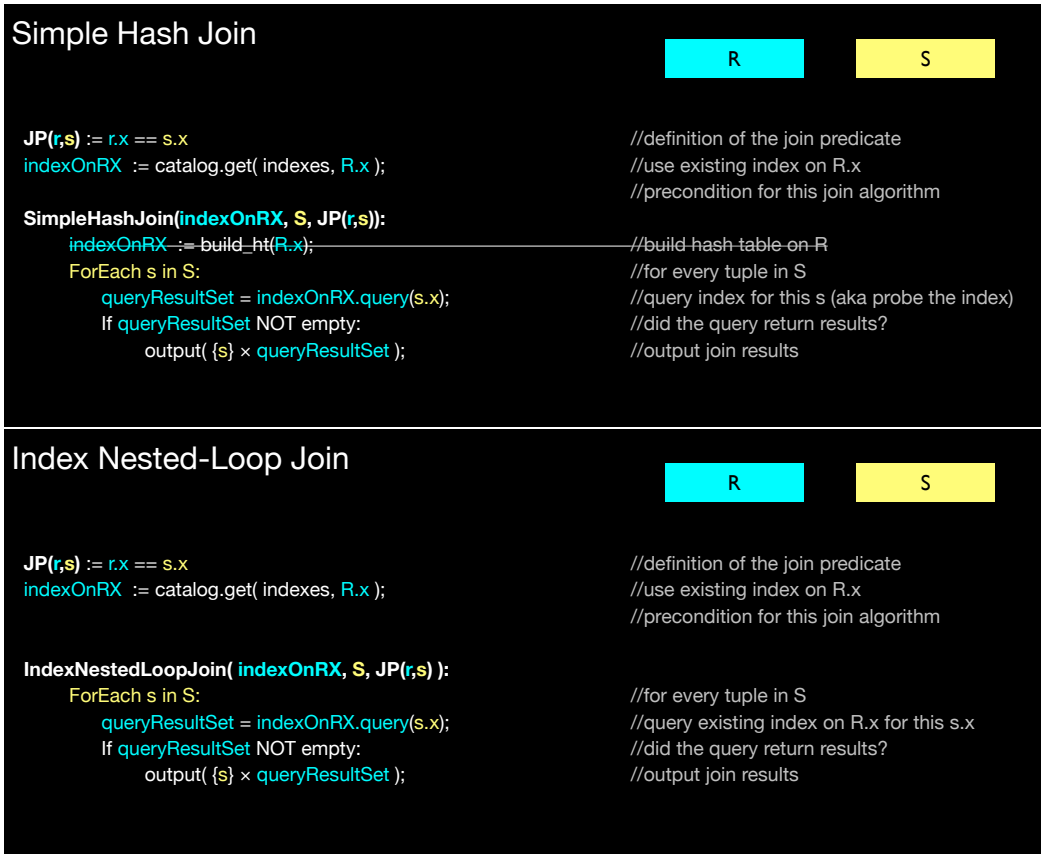


Figure 4.2: Simple Hash Join vs Index Nested-Loop Join

Learning Goals and Content Summary

simple hash join

How does simple hash join (SHJ) work?

SHJ

A simple-hash-join builds (or uses an existing) hash table on one of the inputs, say R. All elements of S are then probed against that hash table. SHJ is merely a special case of an index-nested loop join. In INL, just replace any occurrence of "index" by "hash table". For historic reasons, however, these classes of joins are often still treated as separate algorithms.

For which type of join predicates does SHJ work?

SHJ is more restricted in this regard than the more general INL as hash tables typically do not support range queries (see Discussion in Section 4.1.1).

Quizzes

1. Can the Simple Hash Joins be considered a special case of Index Nested-Loop Join?
   (a) Yes
   (b) No
2. If a Simple Hash Join uses hashing with overflow chains on attribute $R.x$ of table R, what elements are returned as the result of SELECT * FROM R JOIN S ON

R.x = S.x?

(a) Pairs of R and S that do not hash to the same bucket.

(b) All pairs of R and S that hash to the same bucket.

(c) All pairs of R and S that hash to the same bucket and for which $R.x == S.x$.

4.1.3 Sort-Merge Join, CoGrouping

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Sort-Merge Join
[RG03], Section 14.4.2

Learning Goals and Content Summary

sort-merge join

How does sort-merge join (SMJ) work?

SMJ

Assume an equi-join predicate JP(r,s):= r.x==s.x. In order to perform an equi join using SMJ, we first sort both tables on their join columns, i.e. we sort table R on attribute x and table S on attribute x. For each input table we may skip sorting if the table is already sorted according to the join attribute. Now, the main idea is to step through both tables synchronously. This is done by keeping two pointers, PR and PS. PR points to the first row of R, PS to the first row of S. Whenever the two pointers point to two rows having the same join key, we found a join result and output it. Then, in any case, we move the pointer currently pointing to the join key (either r.x or s.x) that is smaller. This process is repeated until the end of one of the inputs is reached. See Figure 4.3.

What has to be considered when deciding which pointer to move forward?

In a tie, i.e. if both PR and PS point to rows having the same join key, we may either move pointer PR or PS forward. This, however, may have consequences on the correctness of the sort-merge join algorithm. To make the algorithm correct, we have to consider which of the input tables contains duplicates. Let's assume that R is the left and S the right input table to the join. If R and S are related through a join predicate JP in a 1:N-relationship, in case of a tie, we should always move PS first. If it is a N:1-relationship, we should always move PR first. N:M-relationships should not occur, as they should have been resolved in database modeling by using a 1:N and a N:1-relationship. However, if we are not sure whether one of the join columns is duplicate free or the join is used on a non-foreign key relationship, we have to extend the algorithm to handle possible duplicates in both rows.

duplicates

How would I treat a situation where duplicates exist in both join columns?

In that situation (see also Figure 4.4) we have multiple options:

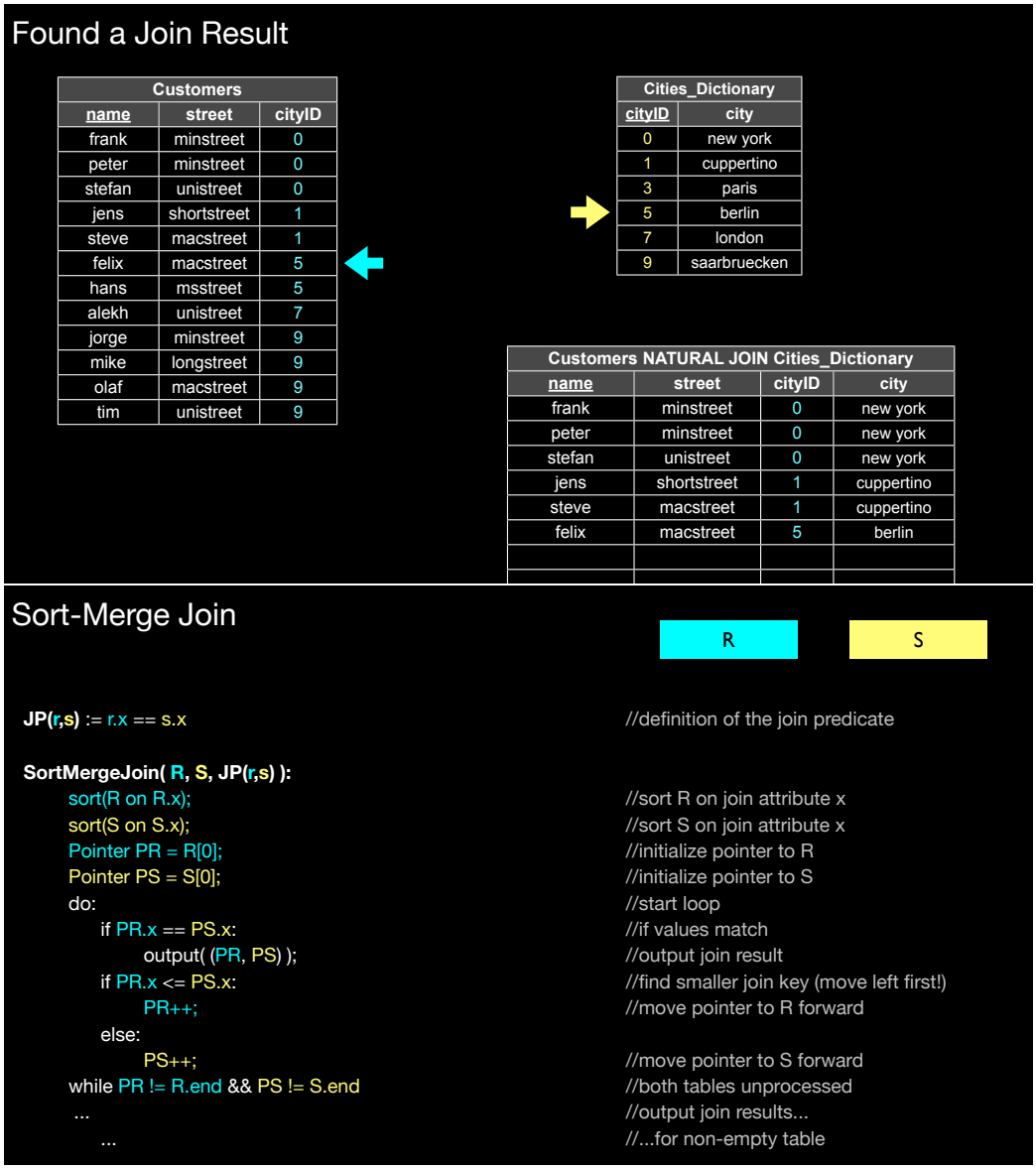


Figure 4.3: Synchronously scanning through two sorted tables in a merge join: example vs pseudo-code

1. modify SMJ allowing it to move one of the pointers, PR and PS, backwards in order to perform a cross product within two duplicate sets having the same join key.

2. keep a window of elements having the same join key for one of the input tables in main-memory. For instance, for table S and at all times, make sure that the set of elements having the currently treated join key is kept in memory. This is also the fundamental idea behind sweep-line algorithms which are wildly used in computational geometry, see [APR+98].

3. use any other algorithm allowing you to

   (a) cogroup both inputs and then

Figure 4.4: Foreign-key constraint typically imply duplicates in one table only; relationship of joins to cogrouping

(b) perform a join within each cogroup.

If the cogrouping is done such that all rows within each cogroup have the same key, then it is enough to use a cross product within each cogroup.

CoGroup

CoGrouping

What is the relationship between joins and CoGroups? What is CoGrouping?

Joins and cogrouping are highly related. Unfortunately, cogrouping is neither part of relational algebra nor SQL. In both, we may only group on a single input relation/table.

Cogrouping is simply an extension of the normal single-relation grouping to allow for multiple inputs. In other words, we group multiple inputs at the same time w.r.t. the same grouping key.

Quizzes

1. Assume you want to perform a Sort-merge Join on two tables R and S. However, both tables are already indexed in the system on the join attribute using a clustered B-tree index. Does the Sort-merge Join still have to perform the sorting step?

(a) Yes

(b) No

2. What is the worst-case complexity (number of comparisons) of the merge step in the Sort-merge Join algorithm? Assume that the number of tuples in tables R and S is n and m, respectively.

   (a) $O(n+m)$

   (b) $O((n+m)\log(n+m))$

   (c) $O((n+m)^2)$

3. Does the Sort-merge Join algorithm (as shown in the video with two pointers PR and PS, checking which of the inputs is smaller and then moving pointers forward) als work if both join attributes are non-unique?

   (a) Yes

   (b) No

4. Would any kind of join predicate, regardless of how complicated it is, be suitable to implement the entire join with a Sort-Merge Join?

   (a) No

   (b) Yes

4.1.4 Generalized CoGrouped Join (on Disk, NUMA, and Distributed Systems)

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

How do we define cogrouping?

There are multiple ways to define cogrouping:

1. **disjoint partitioning variant**: First, we need a grouping function $group(Tuple) \mapsto [0,\ldots,k-1]$. This function must be defined on all input sets. For every incoming tuple it assigns a group ID to the tuple. In addition, we require a single function $partition() : (Set, group()) \mapsto SetofPair < Set, groupID >$. This function partitions an input set into a set of *disjoint* groups. All tuples within a group have the same group ID. See Figure 4.6.

2. **labeling variant**: Another way to define cogrouping is by using separate labeling functions for each input relation. This may emulate a disjoint partitioning as above.

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

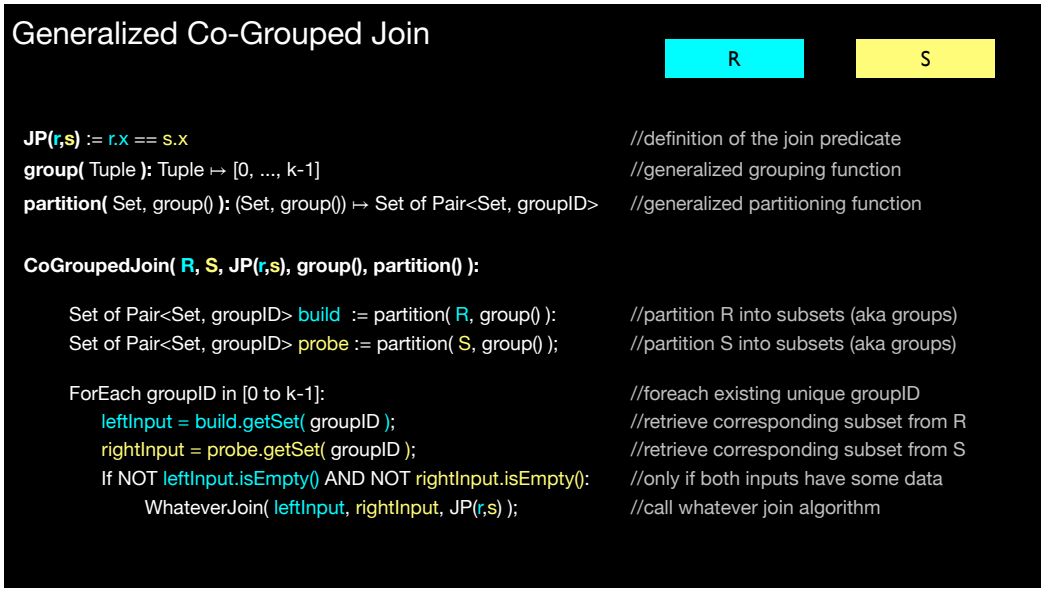


Figure 4.6: Pseudo-code of generalized cogrouped join

algebra. It is not specific to a specific implementation of a 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.
2. Distributed Join: a join algorithm that partitions data into buckets. Each bucket gets assigned to a separate computing node. Then each computing node works on its cogroup independently (be it on disk or in main-memory).
3. NUMA Join: a join algorithm that partitions data into, yes, surprise: buckets! Each bucket gets assigned to a separate memory region in a NUMA architecture. Then each CPU works on its cogroup independently (on NUMA-local main-memory).

So, it is always the same pattern: partition both input sets into buckets using the same partitioning method. After that, handle the cogroups independently. The only thing that changes is the layer of the storage hierarchy that is used to partition the data into buckets and then join the cogroups.

Many other special cases and variants can be derived using generalized cogrouped join as a framework.

Which phases can be identified in the algorithm?

1. Co-Partitioning Phase: partition all input relations
2. Join Phase: perform a join within each cogroup.

How can Generalized Co-Grouped Join be optimized to avoid calling that algorithm in certain cases anyhow?

Let's assume again that we only have to input tables, R and S, that should be joined through an equi join.

1. While partitioning R, we could collect the set of keys in an index, e.g. a Bloom Filter. Then, when partitioning S, we could discard all elements of S whose keys are not contained in that index. This means, elements of S that will not find a join partner in R will never be written to a cogroup in the partitioning phase.
2. After partitioning, for every cogroup where at least one of the sets of the cogroup is empty, we may discard the join on this cogroup as this cogroup simply cannot produce any join result.
3. After partitioning, we may repartition cogroups that are too big (e.g. in Grace Hash Join, a cogroup may be too big to serve as the input to a main memory join, hence we need to repartition).

Quizzes

1. Is it possible to join two tables R and S that are way larger than the main memory of the machine the database is running on?
   (a) Yes

(b) No

2. When co-grouping the tuples contained in two tables R and S, would it suffice to co-group only one of the tables?

   (a) Yes

   (b) No

3. Which of the following join algorithms is not a co-group join algorithm?

   (a) Grace hash join

   (b) Index nested-loop join

   (c) NUMA join

4. Assume we are executing the Co-group Join algorithm explained in the video, and assume that all co-groups on the join key are already computed. If we are doing an Equi-Join, could the final answer (join) be computed from this point on using only a cross-product algorithm?

   (a) Yes

   (b) No

Exercise

Assume you have two relations R and S stored on disk. R has a size of 100 pages and S of 120 pages, respectively. Furthermore you have 14 pages of main memory available including space for input and output buffers. You are given a hash function h() that allows you to partition S into k equally-sized partitions, where k can be chosen by you (note: assume that the sum of the partition sizes in terms of pages is equal to the size of their input dataset. Depending on the k you choose this may lead to slightly unbalanced partition sizes). Due to statistics kept in the database, you already know before executing the join that the sizes of the partitions created by h() on R are highly skewed, i.e. the first partition R_0 contains 90 of the 100 pages of R and the remaining 10 pages are equally distributed among the remaining partitions $\mathrm{R}_1, \ldots, \mathrm{R}_{k-1}$. Assume for simplicity that an in-memory hash table on a data page does not consume any extra space.

(a) How many partitions k do you need to create to perform a generalized co-group join, where at least one side of each co-group fits into main memory?

(b) How many blocks do you need to read and write to perform the co-group join of (a) not counting the I/O-costs for writing out the join result and **not** counting the costs for the initial read of R and S (as we have to read them anyway)?

(c) How can you improve the algorithm to minimize the total number of blocks read and written? What is the number of blocks read and written then? (Hint: What could you do with the available main memory?)

(d) How to specialize this join of (a) to run in a multicore NUMA architecture (single machine)? Provide (high-level) pseudo-code.

(e) How to specialize this join of (a) to run in a distributed system (k machines)? Provide (high-level) pseudo-code.

Exercise

All of them work well for equi-joins. You are given two tables A and B and the following query:

```
SELECT *
FROM   A, B
WHERE  sqrt(abs(A.x - B.x)) < 3
```

(a) Which of the join algorithms we already learned about can be used to implement this query? And if so, how?

(b) Provide pseudo-code of an efficient join algorithm for the above query.

4.1.5 Double-Pipelined Hash Join, Relationship to Index Nested-Loop Join

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

double-pipelined hash join

DPHJ

When does it make sense to consider using double-pipelined hash join (DPHJ)?

Double-pipelined hash join is suitable in cases where:

1. we need to produce at least some join results early, i.e. we do not want to wait until a hash table was populated with an entire input set (like in SHJ) or we do not have the time to sort sort the inputs (like in SMJ)
2. the inputs are delivered by a stream (or any sort of pipelining concept) which:
   (a) may temporarily block, i.e. for a certain time interval it does not deliver any results anymore for whatever reason (e.g. network problem)
   (b) is unbounded, hence there is no way to put it entirely into a hash table or sort it

How does DPHJ work? In what sense is this algorithm symmetric?

The algorithm works as follows: we initialize one empty hash table for each input set, let's call them $\mathrm{HT_R}$ and $\mathrm{HT_S}$.

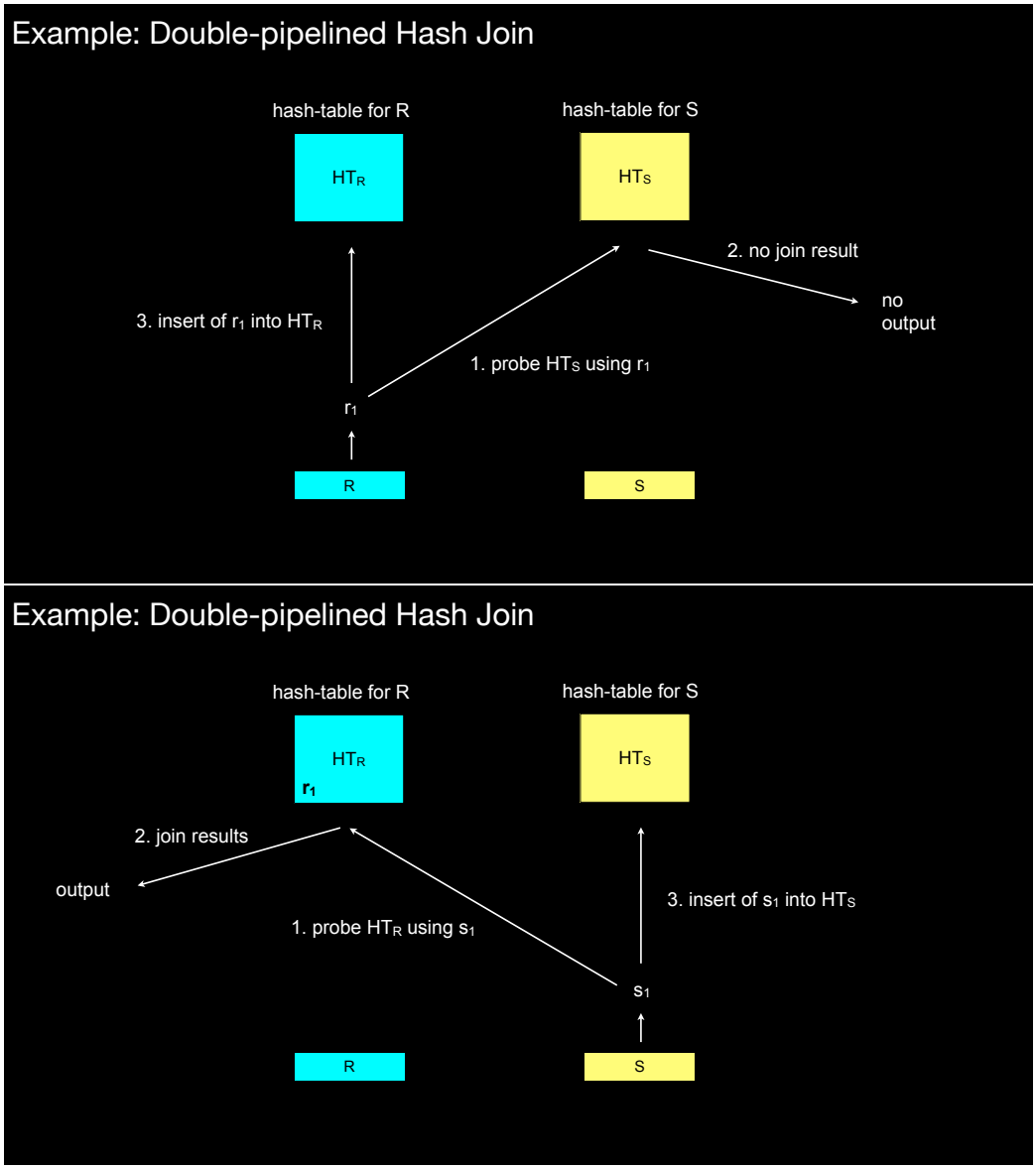


Figure 4.7: Processing element of R is mirror-inverted the same as processing an element of S

1. decide which input set to draw from. Let's assume we draw tuple r_1 from R (see Figure 4.7). Now, we
2. probe $\mathrm{HT_S}$ using r_1 as the lookup key
3. if that probe produces results, return the cross product of r_1 and the result set as a join result
4. insert r_1 into $\mathrm{HT_R}$

Now, we need to decide again which input to draw from:

1. decide which input set to draw from. Let's assume we draw tuple s_1 from S (see

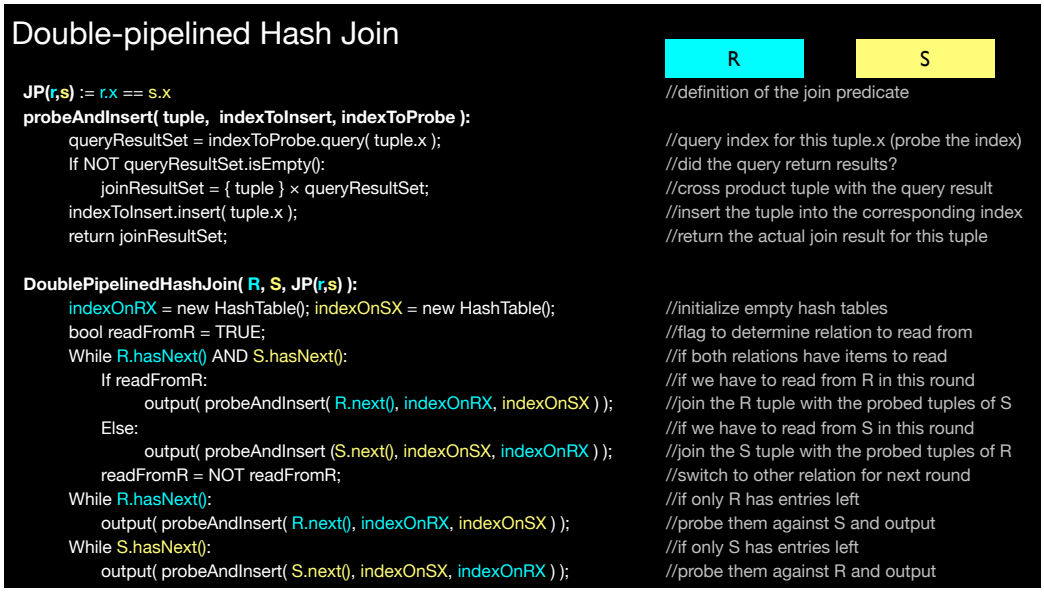


Figure 4.8: Pseudo-code of DPHJ and its difference over INLJ.

Figure 4.7). Now, we

2. probe HT_R using s_1 as the lookup key
3. if that probe produces results, return the cross product of s_1 and the result set as a join result
4. insert s_1 into HT_S

And so forth, ...

So, in this algorithm we simply exchange the roles of HT_R and HT_S as well as the roles of r_1 and s_1. That is why this algorithm is symmetric.

So a high-level way of phrasing this principle would be:

1. decide which input set to draw from. Draw element x from that input.
2. probe HT of other input using x as the lookup key
3. if that probe produces results, return the cross product of x and the result set as a join result
4. insert x into this input's HT

double-pipelined index join

DPIJ

What is double-pipelined index join (DPIJ)?

SHJ simply instantiates INL using a hash table. As discussed above, SHJ can be considered a special case of INL. It is the same historic confusion as INL versus SHJ. For DPHJ as well there is no need to use a hash table as an index, any other index supporting the join predicate efficiently works well. For instance, you could run this algorithm using two B-trees.

What is the relationship between DPIJ and Index Nested-Loop Join?

INL can be considered a special case of DPIJ in the sense that INL makes a decision on a specific order of drawing elements. For DPIJ we did not define an order and just said that the order may depend on the availability of the input tuples (in particular in a streaming environment). In contrast, INL first draws *all* elements from the left input, then it draws *all* elements from the right input. Obviously, if you stick to this drawing order from the beginning, there is no need to build (and keep) a hash table for the right input. There is also no need to probe any element of the left input against elements stored in such hash table. And then DPIJ becomes exactly the same as INL. To be more precise, DPIJ mimics the behavior of INL.

Does double-pipelined index join have a build and a probe phase?

Not really. Only *for each element treated* you may split the algorithm into a build phase (inserting the element into the hash table) and a probe phase (querying the other hash table with that element). Again, this separation exists only for each element treated. There is no split into two separate phases over the execution of the entire join as in INL or SHJ.

What is the difference between DPIJ and INLJ w.r.t. the join results produced over time?

INLJ

DPIJ may produce join results very early. In contrast, in INLJ we first have to build the index — which may take a lot of time. In INLJ, only after having built the index, we may produce join results. Obviously, even though DPIJ may produce join results early, the number of join results produced will be relatively small in the beginning as initially the hash tables are scarcely populated.

Quizzes

1. Assume you have a collection of 10 bee hives on which you have different kinds of sensors, say outer temperature, inner temperature, temperature right below the hive, etc. The sensors constantly output data that you are interested in analyzing, so a join operation is a natural operation to perform on the data. What join algorithm would you choose for this scenario?

   (a) Nested-Loop Join

   (b) Sort-Merge Join

   (c) Double-pipelined Hash Join

2. Assume you want to join two tables R and S, each of which have a fixed number n and m of tuples, respectively. What hash-based join algorithm would you rather choose for the join if you are interested in the least number of calls to a hash function?

   (a) Double-pipelined Hash Join

   (b) Simple Hash Join

4.2 Implementing Grouping and Aggregation

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[RG03], Section 14.6

Learning Goals and Content Summary

Hash-based Grouping and Aggregation

R

grouping attribute: R.x
aggregation function: aggregate(MultiSet<R'>): MultiSet<R'> ↦ <ScalarType>, [R'] ⊆ [R]

```
HashBasedGrouping( R, aggregate() ):
    HashMap hm = new HashMap();                  //initialize hash map
    List group = NULL;                           //handle to group of R.x

    ForEach r in R:                              //for every tuple in R
        If NOT hm.contains( r.x ):               //check if already seen this key
            group = new List();                  //create new group
        Else:                                    //i.e. key already in hash map (key seen before)
            group = hm.get( r.x );               //get existing group from hash map
        group.append( r );                       //append element to group
        hm.put( r.x, group );                    //insert/replace group in hash map

    ForEach key in hm:                           //loop over all existing keys in hash map
        group = hm.get( key );                   //retrieve result group for that key
        aggregationResult = aggregate( group );  //call aggregation function on that group
        output( key,  aggregationResult );       //output result pairs
```

Sort-based Grouping and Aggregation

R

grouping attribute: R.x
aggregation function: aggregate(MultiSet<R'>): MultiSet<R'> ↦ <ScalarType>, [R'] ⊆ [R]

```
SortBasedGrouping( R, aggregate() ):
    sort( R on R.x );                            //sort R on grouping attribute R.x
    Pointer PR = R[0];                           //initialize pointer to first tuple of R
    Value currentGroupValue = R[0].x;            //value of R.x for this group
    List group = new List();                     //create new group
    Do:                                          //start loop
        If PR.x != currentGroupValue:            //if current tuple belongs to same group
            aggregationResult  = aggregate( group );   //aggregate existing result group
            output( currentGroupValue,  aggregationResult );   //output result pairs
            group = new List();                  //create new group
            currentGroupValue = PR.x;            //re-init value of R.x for this group
        group.append( PR );                      //append this tuple to result group anyway
        PR++;                                    //move pointer forward to next tuple
    While PR != R.end;                           //end of table?
    aggregationResult = aggregate( group );      //aggregate existing result group
```

Figure 4.9: Pseudo-code of hash-based vs sort-based grouping and aggregation

Would it be fair to say that grouping and aggregation are just like cogrouping but using a single subset for each cogroup?

grouping
aggregation
cogrouping

Yes.

Which phases can be identified in hash-based grouping?

hash-based grouping

1. grouping phase: build hash table on the left input (if there isn't already a hash table available on that key)
2. aggregation phase: for each distinct key existing in the hash table: call aggregation function on elements of that group

What exactly is kept in the hashmap?

hashmap

In general, you need to keep the entire tuple of the input. However, depending on your grouping and aggregation attributes (which implicitly project the tuples) you may keep less data in the hashmap. For instance, assume a query SELECT A, sum(B) FROM foo. Independently of the schema of table foo, for the hashmap we simply use attribute A as the key and the running sum on attribute B as the value. There is no need to store any other information. Notice that this optimization may only be applied to so-called distributive aggregation functions.

OK, but what is a distributive aggregation function?

distributive aggregation function

Assume an aggregation function $f : X \mapsto Y$. Let $\{X_0, \ldots, X_n\}$ be a disjoint and complete partitioning of X, i.e. $X_i \cap X_j = \emptyset \; \forall i \neq j$ and $\bigcup_i X_i = X$. Assume there exists a function $h : X \mapsto Y$ such that

$$h\Big(\big\{f(X_0), \ldots, f(X_n)\big\}\Big) = f\big(\{X_0, \ldots, X_n\}\big).$$

Then we call $f()$ a distributive aggregation function. In other words, we may compute this aggregate by first aggregating partitions of the data using $f()$. These initial aggregates are then further aggregated using $h()$. The aggregation is *distributed* over different function calls. Notice that $h()$ maps from X to Y just like $f()$.

So why is sum() distributive?

Because we may define $h() = g() = sum()$.

Is avg() distributive?

No, however, $avg()$ is algebraic.

Sounds great, hmm, but what exactly is an algebraic aggregation function?

algebraic aggregation function

Similar to a distributive aggregation function, we want to perform the aggregation in steps. However, we simply cannot find two functions $h()$ and $f()$ that operate on the same domains. We need more freedom in choosing these functions. So, again our user-defined aggregation function is $f : X \mapsto Y$. We need two functions $h()$ and $g()$ where

$$g() : X \mapsto W \wedge h() : \{W\} \mapsto Y.$$

Here, W is any suitable "intermediate" domain. Now we can search for two functions $h()$ and $g()$ such that:

$$h\Big(\big\{g(X_0),\ldots,g(X_n)\big\}\Big) = f\big(\{X_0,\ldots,X_n\}\big).$$

If these functions exist, we call $f()$ an algebraic aggregation function.

So why is avg() distributive?

Because we may define

$$g() : X \mapsto (Y, \texttt{int}) \wedge h() : \{(Y, \texttt{int})\} \mapsto Y$$

where $g()$ returns a pair with the sum and a count of the inputs and $h()$ simply adds up these pairs and returns the quotient as a result.

sort-based grouping

Which phases can be identified in sort-based grouping?

1. grouping phase: sort table on the grouping key (if it is not already sorted w.r.t. that key)
2. aggregation phase: for each distinct key in the sorted sequence: call aggregation function on elements of that group

Quizzes

1. Could the Sort-based Grouping algorithm highly benefit from an existing B-tree on the grouping attribute?
   (a) Yes
   (b) No
2. Could the Sort-based Grouping algorithm benefit in general from an existing hash table on the grouping attribute?
   (a) Yes
   (b) No
3. In the Hash-based Grouping and Aggregation algorithm, once the hash table has been created and we proceed to perform the actual aggregation, the pseudo-code loops over the set of existing keys to retrieve the content of all non-empty hash buckets and then perform the actual aggregation. Could we implement this loop without calling the hash function for each existing key again?
   (a) No, as this would aggregate different keys within the same bucket (hash collisions)
   (b) No, as we cannot identify the buckets of the existing keys.
   (c) Yes, as we could simply iterate over the hash buckets to identify non-empty buckets. Within each bucket the entries have to be post-partitioned anyway.

4.3 External Sorting

4.3.1 External Merge Sort

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[RG03], Section 13.1 – 13.2
W Main Memory Merge Sort Recap
[LÖ09], Index Join
W Merge-sort with Transylvanian-saxon (German) folk dance
W Why it is important to know about Sorting in any Job

Learning Goals and Content Summary

Figure 4.10: Pass 0: Run generation from first to (in this case) fourth run

When does it make sense to run external merge sort?

external merge sort

It makes sense to use external merge sort when a dataset is too large to be sorted in main-memory. That is the vanilla answer.

But let's think a bit more about this. Why not simply use quicksort on a memory-mapped or ram-buffered file? So, let's assume we have n pages to sort, but only $m < n$ pages of main memory. The critical part for quicksort is the initial phase when it is recursively partitioning the data and those partitions do not yet fit into main memory.

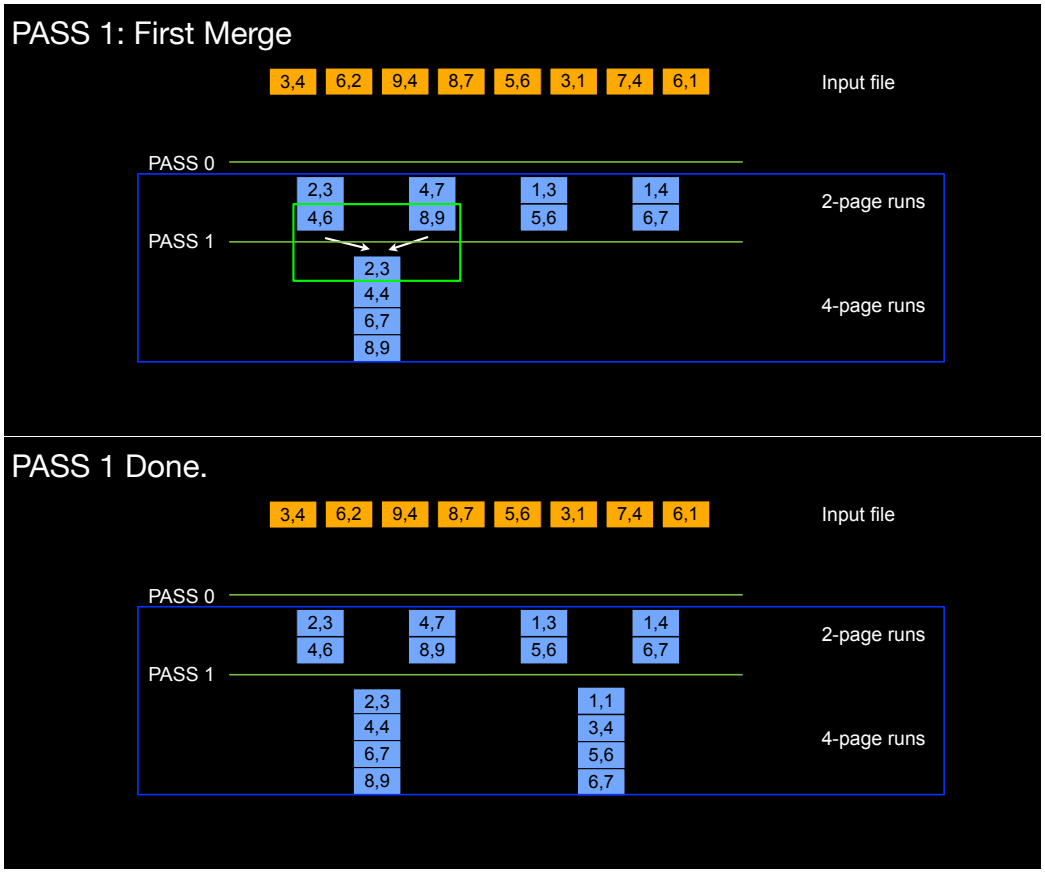


Figure 4.11: Pass 1: Merge runs containing two pages each into larger runs containing four pages each

In order to perform one pivot step on an input partition p, quicksort will read the entire partition from the beginning and from the end until both pointers meet. In addition, whenever there is a swap of a pair of elements, the elements change a page and hence those pages have to be written back to disk (alternatively, we could write out the two new partitions to a separate place). So, in summary, for a single partition step, we can assume that you have to read and write the entire input partition once. Every pivot step partitions the data into two sub-partitions. You need to recursively partition until the partitions fit into main memory. Then you simply run quicksort on them.

The main problem with this algorithm is that it performs a binary partitioning, i.e. in one step the input partition is split into two sub-partitions only. This may implicitly lead to a lot of random I/O if both the input and the output partitions reside on the same hard disk and/or the underlying I/O-system does not make good use of the available main-memory, i.e. the page sequences read or written at a time are too small. This triggers another problem: you have to continue the binary partitioning until the partitions fit. Recall, that the number of recursion levels of this strategy is $\log_2\lceil n/m \rceil$. For instance, if the dataset is 8 times bigger than the available main memory, we will need three recursion levels. This means, the entire dataset is read three times and written three times.

Notice that another, minor problem, with this algorithm is that it reads the input

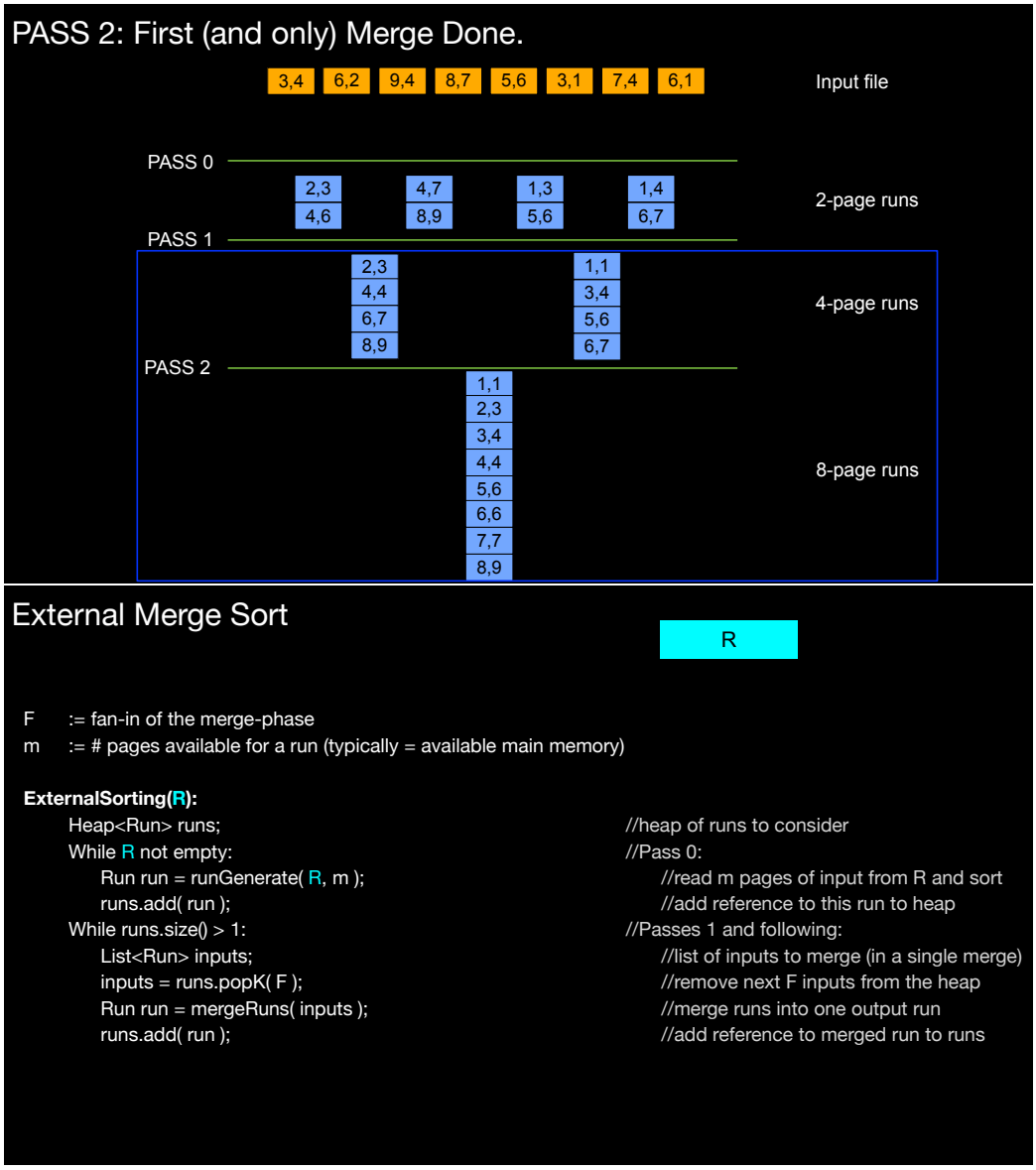


Figure 4.12: Pass 2: Merge runs containing four pages each into larger runs containing eight pages each; pseudo code of external merge sort

using two input streams. This means, it uses four streams in total (two for reading and two for writing) rather than three (one for reading and two for writing).

So how could we fix this? We need an algorithm that performs less passes over the data. However, at the same time that algorithm should not introduce too many random I/O-operations. Balancing the amount of passes over the data and random I/O is difficult and depends heavily on the concrete combination of hardware, operating systems, and your algorithm.

For the binary partitioning discussed above, it is easy to see that a better strategy is to partition by reading one input and writing F output partitions at a time. In partitioning, F is called the fan-out. The number of partitions is limited by $F < m - 1$ as we need

fan-out

to keep a buffer of the size of one page for reading the input and a buffer of one page for each output. If we increase F, we will need more random-I/O-operations, but less passes over the data and vice versa.

Exactly the same balancing happens in external merge-sort. However, in contrast to a recursive partitioning, we start from the other direction: we first start by creating sorted "partitions" (called runs in the following). Then we recursively merge these sorted "sub-partitions" into larger sorted partitions, hopefully with an $F > 2$.

run generation

What is run generation?

In run generation we fill the entire available main-memory with data, quicksort the data, and write the sorted data out to disk. We call this sorted sequence on disk a run. Obviously, if we use quicksort, a run has the same size as the available main-memory.

run

merge phase

What happens during the merge phase of external merge sort, i.e. pass > 0?

pass

We repeatedly merge multiple input runs into one larger input run. We continue with this process until we end up with a single sorted run on disk — which is the final result of the sort operation. It is a good idea to keep track of the existing runs using a heap which prioritizes by the size of the run, i.e. the shortest run has the highest priority. This leads to balanced merge-trees.

fan-in

What is the fan-in F?

F

The fan-in F is the number of input runs we merge in a single merge operation.

What happens during a single merge of external merge sort?

The merge may be implemented in many ways. A good strategy is the following. If m is the number of main-memory pages available, distribute those pages evenly to serve as input buffers for the F input runs and one output buffer for the output run.

Use a heap with F elements. The heap contains a metadata-entry for each input buffer, i.e. a pointer to the next untreated element of that input buffer and the sort key of that element. The heap prioritizes smaller elements. Now, we repeatedly remove the top-element from the heap and write it to the output buffer. We re-insert an element into the heap for the input just drawn from. If the output buffer overflows, we write it to disk. Likewise, if an input buffer becomes empty, we refill it with more data from the corresponding input run. We continue until all input runs have been entirely processed.

Does the output buffer have the same size as the input buffers?

Assume we have F input runs of size n/F each and m pages of main memory. For simplicity, assume that we do not count the extra space required for bookkeeping (the tiny heap for merging as explained above). Assume we reserve $k_I \cdot F$ pages for the input buffers and k_O pages for the output buffer and of course: $k_I \cdot F + k_O \leq m$. Hence,

$$k_O = m - k_I \cdot F.$$

Or:

$$k_I = (m - k_O)/F.$$

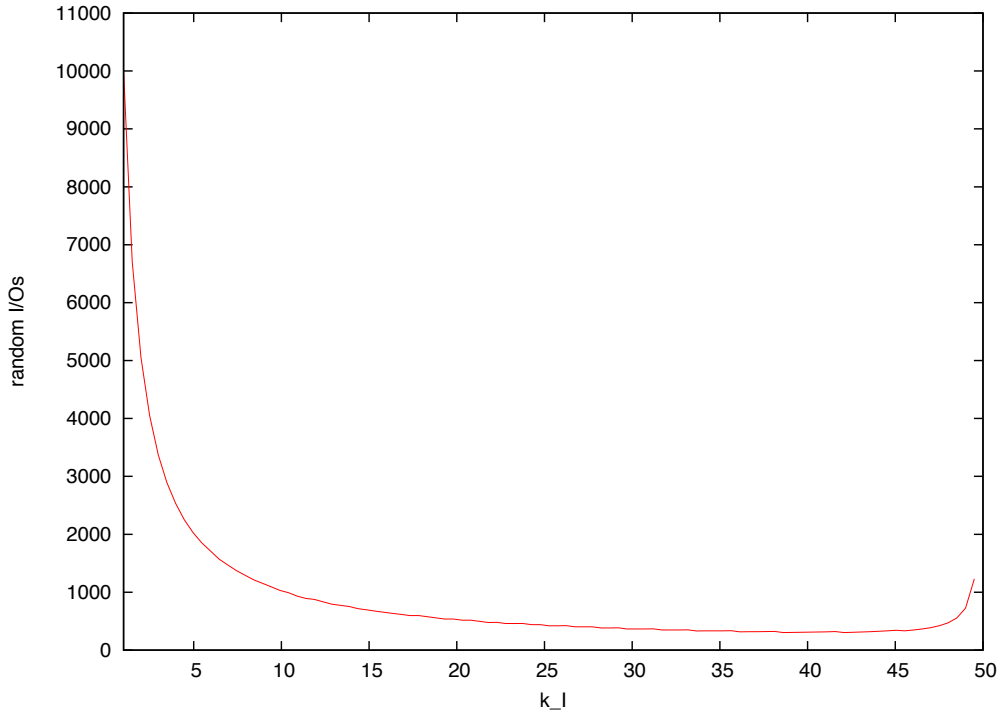


Figure 4.13: A single merge-operation: number of random I/O-operations when varying the size of the input buffers over the size of the output buffer for $n = 10,000$, $m = 1,000$, and $F = 20$. If we increase F the "valley" gets more narrow.

We can assume that each refill of an input buffer as well as every write of the output buffer triggers a random I/O-operation. Hence, the number of random I/O-operations of a single merge-step is:

$$f_{rand}(k_I, F) = F \cdot \lceil (n/F)/k_I \rceil + \lceil n/(m - k_I \cdot F) \rceil$$

Here, n and m can be considered constants. An example for $n = 10,000$, $m = 1,000$, and $F = 20$ is shown in Figure 4.13. We can observe that for these settings there is a wide range where we obtain a close to minimal number of random I/Os. We learn that it is important to avoid two situations:

1. the input buffers are relatively small, in contrast, the output buffer has many pages ($k_I < 10$ in the figure),
2. the input buffers have are relatively big, in contrast, the output has few pages ($k_I > 47$ in the figure).

Why would it be a bad idea to implement merge sort with a fan-in of $F = 2$?

As explained above for the case of partitioning, choosing $F = 2$ is a bad trade-off as you will need many merge-levels. The number of merge levels in external sorting is $\lceil \log_F(n/m) \rceil$. Hence, the bigger F the better — again: if we ignore random I/O which increases for a larger F.

Why would it also be a bad idea to implement merge sort with a fan-in $F = m - 1$ (where m is the number of pages available in main memory)?

Again, and as discussed for a single merge (see above), this is only optimal w.r.t. the number of passes over the data, i.e. the amount of data read and written. However, it leads to a tremendous amount of random I/O-operations which may lead to slow runtimes.

You did not forget that this specific variant is a very simple version of the algorithm, right?

External-merge sort may be optimized in many ways. In this section we just focussed on its core ideas. For a good discussion of various optimizations see [Gra06].

Quizzes

1. Assume you want to sort 1,000,000,000 4-byte (positive) integers with the property that each one of them is strictly less than 1,000,000,000. These integers are stored on disk on a server with only 2 GB of RAM. What algorithm would you call to sort these numbers?

   (a) Quicksort

   (b) Radix Sort

   (c) External Merge Sort

Exercise

Suppose you have a hard disk with the following characteristics given by the vendor:

Seek time = 4 ms, **Read/write bandwidth** = 150 MB/s, **Hard disk block size** = 4 KB.

Assume that your disk performs a seek only if you read into or flush from a different buffer, than the previous I/O operation did. Assume that inputs to a merge are uniformly read, i.e. all inputs to a merge are exhausted at the same time.

Your task is to sort 2^{20} 4–byte integers, having 64 KB of memory at your disposal. Compare the seek- and scan costs in the worst case of sorting using external merge-sort for the following three variants:

(a) 2–way external merge-sort, using two memory blocks for the output

(b) 2–way external merge-sort, using half of the memory as an input buffer, and half of it as an output buffer

(c) M–way external merge-sort, using a single memory block for the output (where $M+1$ is the number of available memory blocks)

4.3.2 Replacement Selection

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:	Further Reading:
[Knu73]	W heapsort

Learning Goals and Content Summary

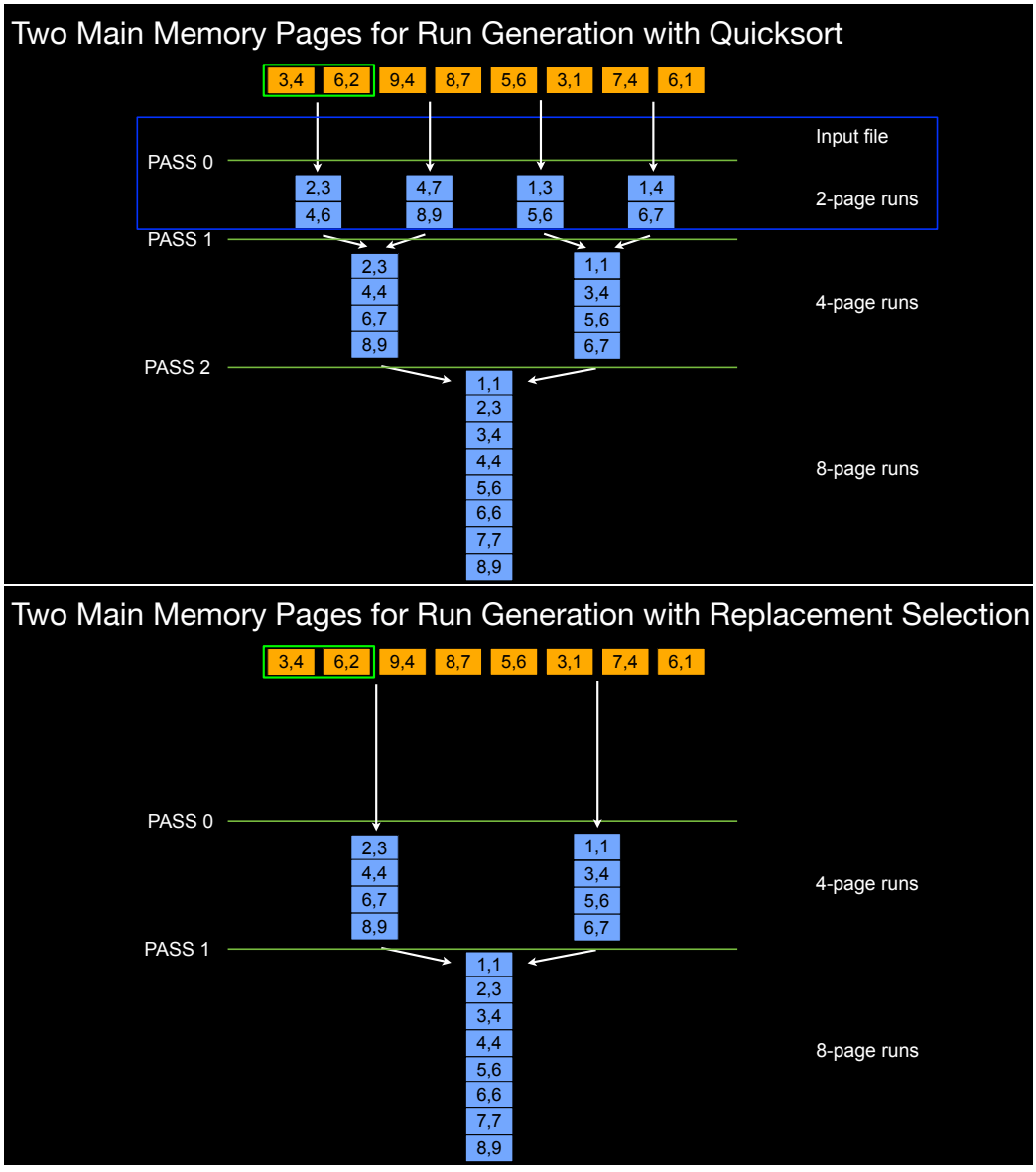


Figure 4.14: Run generation: Quicksort vs replacement selection

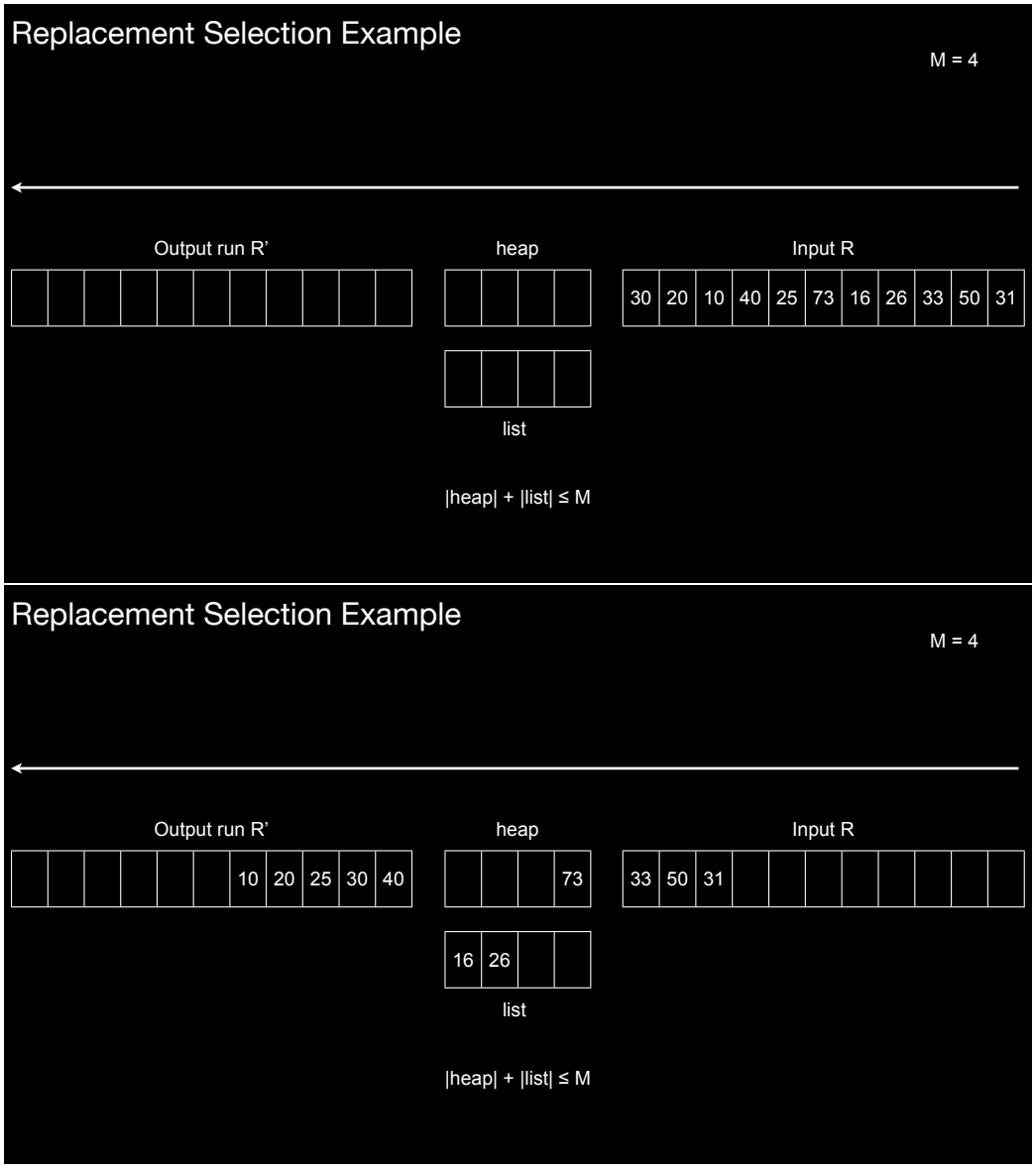


Figure 4.15: replacement selection: using a heap for everything that is smaller than the last pivot and a list for everything that is larger

replacement selection

Which overall impact does replacement selection typically have when used with external merge sort? Why would we use this algorithm?

In general, by using replacement selection we can expect the input runs to become larger than the available main-memory. Hence, we have less input runs compared to using quicksort. Therefore, we may have less merge operations, a smaller fan-out for individual merge operations or even less merge levels in the merge-tree. See Figure 4.14 for an example with $n = 8$ input pages and $m = 2$ pages of main memory for run generation. (Yes, we ignore for a moment that a merge operation needs three main memory pages as a minimum anyway. The example is kept intentionally minimalistic.)

What is the core idea of replacement selection?

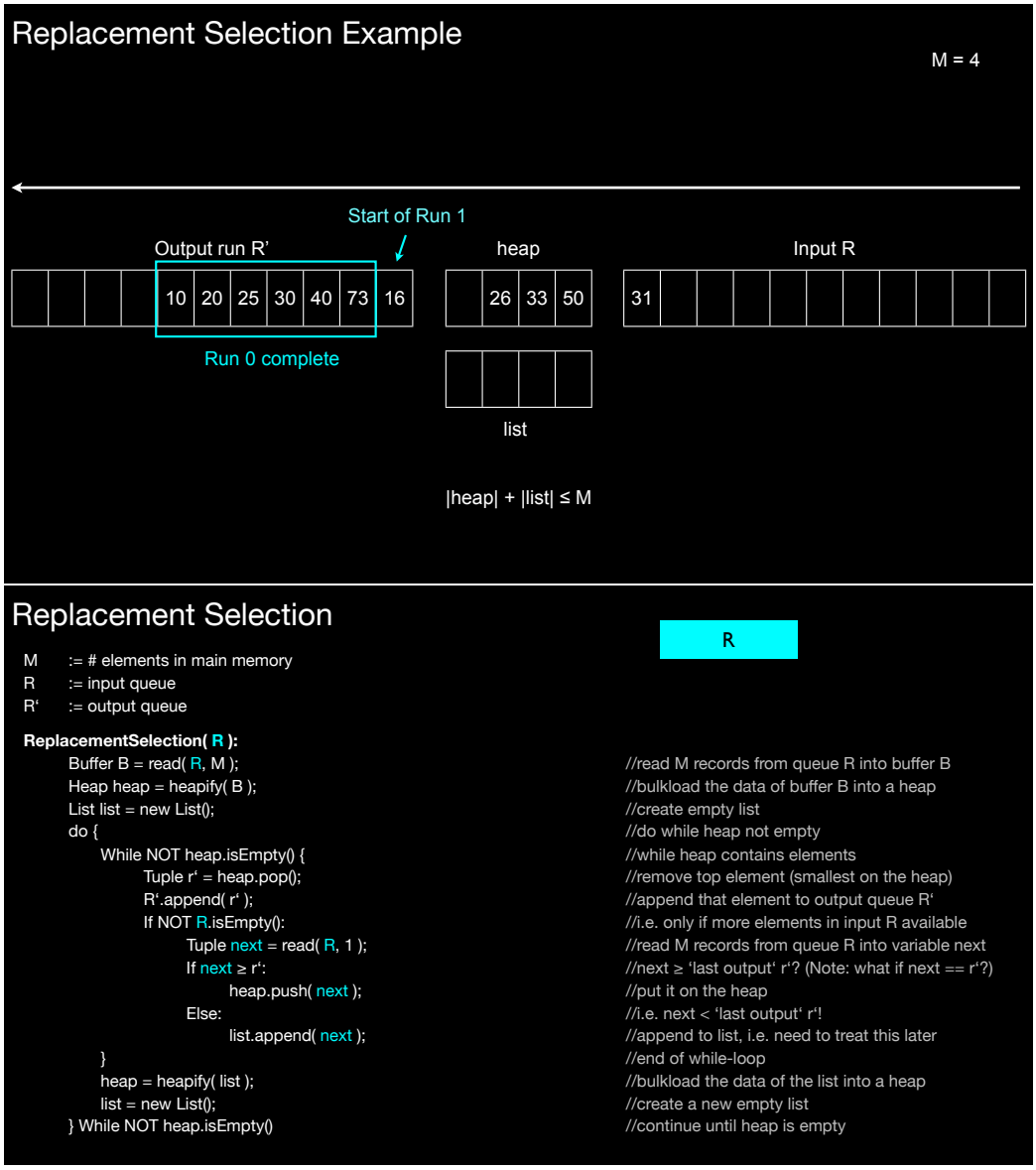


Figure 4.16: replacement selection may produce runs that are larger than the available main memory; pseudo-code of replacement selection

In replacement selection we use all available main memory for sorting, just like in quicksort, however, rather than simply outputting the sorted sequence available in main memory, we output elements one by one. Whenever we remove the currently smallest element r' from main memory and write it to an output run, we fill the empty space with the next element from the input, say *next*. For *next* we check whether we can place it correctly into the sorted sequence available in main-memory. This is the case if *next* is larger than or equal to the last element r' written to disk. Otherwise, i.e. if *next* is smaller than r', we also keep *next* in main-memory, but do not consider it to be part of the sorted sequence.

What is the main trick used in replacement selection?

The main trick here is to make use of the room that becomes available while writing out a run to disk. For data read in that process we simply have to be careful to understand whether we can still integrate it into the sorted output currently written or not.

In order to maintain the two types of elements in main memory (elements that can or cannot be integrated into the sorted sequence), it is easy to see that we do not need to maintain a sorted sequence in main-memory. The only functionality we need is being able to identify the smallest element from the set of elements that can be integrated into the sorted sequence. This is supported by priority-queues like a min-heap (or a max-heap if we want to sort in descending order).

As a heap can operate on an array, and the input elements have a fixed-size, it is a good idea to allocate an array consuming all available main-memory. The head of that array represents the heap, the tail is used to buffer elements that cannot be integrated. During run generation the size of the heap is decreased for every element that cannot be placed in the sorted sequence. Once the heap size is zero, a new output run is started. Also see the examples and pseudo-code in Figures 4.15 and 4.16.

What is the relationship between heapsort and replacement selection?

Replacement selection may be considered a variant of heap sort in the sense that in heapsort the heap is iteratively decreased to make room for the sorted sequence. In contrast, in replacement selection the sorted sequence is written to an output stream. The room that becomes available is exploited to either integrate more elements into the heap or buffer elements that need to be considered for the next run.

quicksort

What is the typical size of a run generated by replacement selection compared to quicksort?

Typically the size of an output run generated by replacement selection is twice the size of the available main-memory. See Donald Knuth's book for a discussion [Knu73].

When is an element inserted into the heap?

If the elements is larger or equal the last element written to the output run.

When is an element inserted into the list?

If the element is strictly smaller than the last element written to the output run.

When does a run end?

A run ends if the heap size becomes zero, i.e. there are no more elements available in main memory that may be integrated into the sorted sequence currently being written to the output run.

cache-efficiency

What can be said about the cache-efficiency of this algorithm?

In terms of cache-efficiency, replacement selection inherits the properties (and problems) of heapsort. Heapsort is optimal in a theoretical sense (also recall our discussion in Section 3.3), i.e. every insert and remove-operation to the heap takes in the worst case $O(\log n)$ comparisons. In total this leads to a worst case of $O(\log n)$. However, heapsort is not very-cache-efficient: in particular for lower levels of the heap the likelihood increases that an element is not available in a cache and thus we receive an LLC. In contrast,

for quicksort and depending on the pivot strategy, the worst case complexity is $O(n^2)$. Quicksort is simply less robust than heapsort. However, in general, quicksort is much more cache-friendly. In most real world cases quicksort is faster than heapsort. This is (probably) the reason why it is used in many standard libraries like STL or boost. However, quicksort may degenerate heavily for some inputs.

So what is the best sorting algorithm in a database?

Interestingly, this is not easy to answer because it depends heavily on the storage system and data layouts used. Even for the very simple case of sorting fixed-size data in main-memory the answer is non-trivial, see [ASDR14] for an experimental comparison of various multi-threaded main-memory methods.

Quizzes

1. Is it theoretically possible to create sorted runs of size m even though the available RAM of the server cannot hold m elements at the same time?
   (a) Yes
   (b) No
2. Is it true that Replacement Selection always produces the sorted sequence of its input set of elements?
   (a) Yes
   (b) No
3. Is it true that Replacement Selection always produces a set of sorted sequences of its input set of elements?
   (a) Yes
   (b) No
4. In Replacement Selection, how many passes are performed over any given (whole) input disk block during the execution of the algorithm?
   (a) Only one
   (b) The number of elements it contains
5. Assume you run Replacement Selection on a sequence of $n > 0$ numbers that is sorted in decreasing order on disk. If the server has enough RAM to keep $m << n$ of those numbers at the same time, what is the number of increasing runs produced by the Replacement Selection Algorithm?
   (a) $\lceil n/2 \rceil$
   (b) $\lceil m/2 \rceil$
   (c) $\lceil n/m \rceil$

Exercise

Assume you have a main memory buffer of four tuples at any given time and also that you have enough space reserved in memory for input and output operations. You wish to create initial runs from the following sequence of sort keys using replacement selection:

25, 11, 71, 12, 8, 66, 69, 32, 71, 81, 96, 76, 37, 42, 51, 62, 2, 14, 27, 87

(a) Perform run generation and show each step of it.

(b) How many runs did you create? How large is each run?

4.3.3 Late, Online, and Early Grouping with Aggregation

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

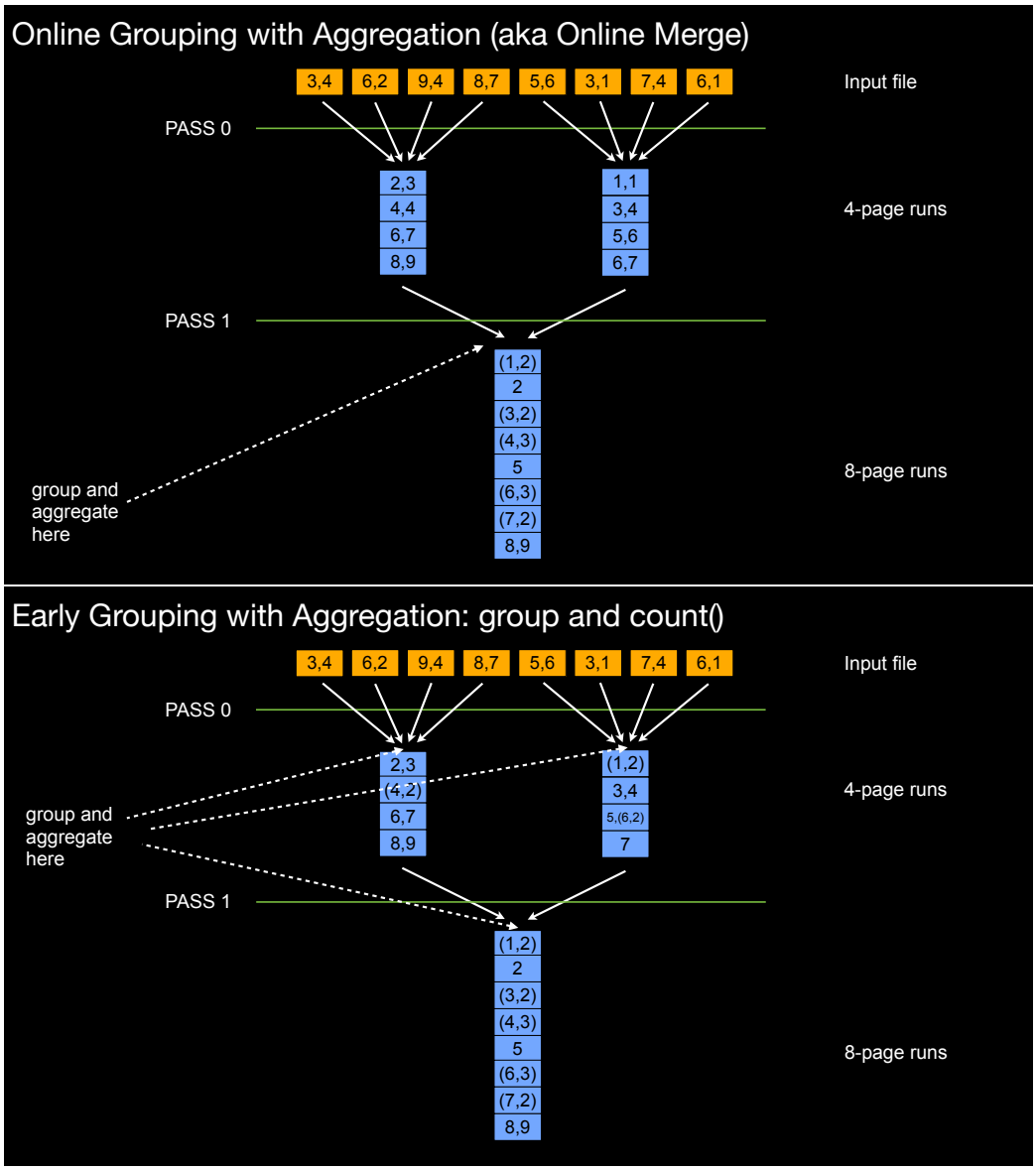


Figure 4.17: Grouping and aggregation: online vs early

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 grouping step. Hence "late aggregation" may include "late grouping" as a component. We

will differentiate among the two things in the following and always talk about "grouping and aggregation".

late grouping and aggregation

What is late grouping and aggregation?

In late grouping and aggregation we first fully sort the data using external sorting. This includes writing the sorted data to disk. After that the sorted data is read. In that process we assign adjacent rows having the same grouping key to a group and for each group we compute the aggregates.

In summary, in this method we do not overlap the execution of external sorting with grouping and aggregation.

online grouping and aggregation

What is online grouping and aggregation?

In online grouping and aggregation we perform external sorting partially, i.e., we perform run generation and all merges except the final merge. This means the fully sorted dataset is not written to disk. After that we perform the final merge while grouping and aggregation is done at the same time. In other words, while we read the remaining F runs from disk, we merge them in main-memory and feed them directly into the grouping and aggregation operation. Just like in standard grouping and aggregation, in that process we assign adjacent rows having the same grouping key to a group and for each group we compute the aggregates.

In summary, in this method we overlap the execution of the final merge of external sorting with grouping and aggregation.

early grouping and aggregation

What is early grouping and aggregation?

In early grouping and aggregation whenever we generate a run (be it using quicksort or replacement selection) or whenever we execute a merge, we feed the sorted output stream directly into the grouping and aggregation operation. This means, not only the final merge is directly fed into the grouping and aggregation operation (like in online grouping and aggregation). In contrast, every time a sorted stream is generated, we exploit this as an opportunity to already perform grouping and aggregation.

Just like in late and online grouping and aggregation, every time we group we assign adjacent rows having the same grouping key to a group and for each group we compute the aggregates.

Notice that the aggregate functions need to be distributive or algebraic for this method (see Section 4.2).

In summary, we fully overlap the execution of external sorting with grouping and aggregation.

What has a huge impact on the actual savings for early grouping and aggregation?

Early grouping and aggregation works best if only few groups are created. So the general pattern where this method shines is a query like SELECT foo, count(*) FROM table. If the cardinality of attribute `foo` is small, only few distinct groups will be created, and the savings in I/O will be relatively high.

Can early grouping and aggregation only be used with grouping or also with aggregation-only queries?

Assume we do not specify a grouping condition like in SELECT count(*) FROM table. Then there is no need to group the data, hence there is no need to sort, we simply aggregate the data into a single value.

In other words, recall that we sort the data in order to group, however the data within *within* a group does not need to be sorted. Then there is no need for grouping and hence sorting, we simply aggregate. And that aggregation is (kind of) early anyway...

What is the amount of data transferred by the different methods?

Let's assume we want to perform a sort-based grouping (see Section 4.2) using external merge-sort for sorting (see Section 4.3.1). Let's define 1 R (1 W) to be the I/O-costs (just the amount of data transferred) for one read (write) **of the entire dataset**. Let's assume that T corresponds to the I/O required to write the grouped and aggregated dataset. Notice that we will assume the typical case that $T \leq W$, but this does not have to be the case: it depends on the grouping attribute and the type of aggregation.

Assuming $k = \lceil \log_F(n/m) \rceil$ merge-levels (as discussed in Section 4.3.1), for external sorting, we will require 1 R and 1 W for run generation plus 1 R and 1 W for each merge-level, i.e. in total $(k+1)$ R and $(k+1)$ W. However, the total data transferred to perform both sorting and grouping/aggregation varies considerably across the different methods. See Table 4.1.

	Grouping and Aggregation Method		
	Late	**Online**	**Early**
where?	after external sorting wrote the sorted output to disk	during final merge of external sorting	during run generation and every merge of external sorting
I/O	$(k+2)R+(k+1)W+T$	$(k+1)R+kW+T$	$R+(x)+T$

Table 4.1: Methods for grouping and aggregation and their I/O-costs

What are drawbacks of both online and early grouping and aggregation?

In early grouping and aggregation we create a sorted version of the input table as a side-effect. That sorted table may be exploited for future queries requiring the same sort order. In online and late grouping and aggregation we never materialize that sorted output on disk.

Quizzes

1. Assume you have the following query: SELECT number, sum(*) FROM input GROUP BY number. If you perform this query on a set of numbers so large that it does not fit into main memory, and you compute the queries by External Merge Sort + Grouping + Aggregation, what is the main bottleneck in the performance obtained?

   (a) Number of comparisons

   (b) I/O operations

2. What is (definitely) an easy fix to the problem of multiple read/write operations in the last step of External Merge Sort + Grouping + Aggregation?

   (a) Use Replacement Selection to create bigger runs each time

   (b) Group and aggregate the output directly right in the last pass

3. When grouping elements, these elements can be stored as tuples of type (i, m), meaning that input element i appears m times in the input run. If we group and aggregate directly right from the very beginning, in the Early Grouping and Aggregation algorithm, could the size of the output stream of the early aggregation be larger than its input?

   (a) Yes

   (b) No

Chapter 5

Query Planning and Optimization

5.1 Overview and Challenges

5.1.1 Query Optimizer Overview

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Query Plan
W PostgreSQL System Catalog

Learning Goals and Content Summary

What is the task of the query parser? **query parser**

The task of the query parser is to translate a declarative query (the WHAT part), i.e. an SQL statement, to an algebraic expression. That algebraic expression resembles relational algebra, however, it may contain additional annotations, e.g. specific hints available in the SQL statement.

What is the high-level task of the query optimizer? **query optimizer**

The high-level task of the query optimizer is to translate the algebraic expression into an executable program (the HOW part). That executable program computes the result to the SQL query. Typically, the goal of the query optimizer is to generate an optimal program w.r.t. runtime. However, this does not need to be the optimization goal. For instance, alternative optimization goals may be to generate a program that is never worse than a full scan or a program that is not more than k-times slower than the optimal program. The latter goals emphasize robustness of query optimization rather than overall performance. **robustness**

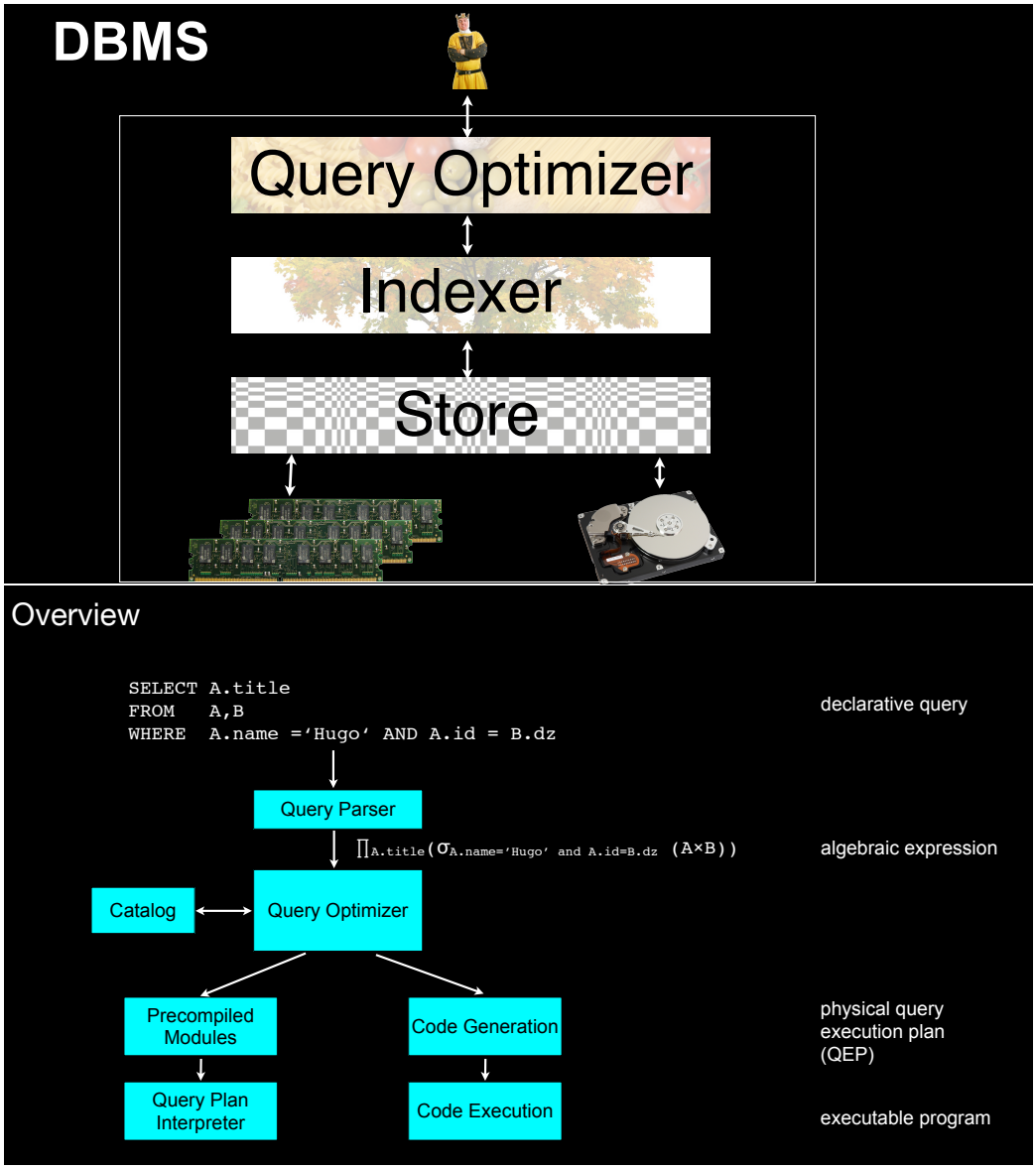


Figure 5.1: The positioning of the query optimizer in a DBMS; input and output of the query optimizer

What is the relationship between the WHAT and the HOW aspects query planning?

The SQL statement does not specify HOW the result to that query should be computed. Using an SQL query the user just declares WHAT he wants to retrieve from the database. The query optimizer then has to decide HOW to retrieve and compute that data from the database. That decision is an optimization problem that must be solved either when the SQL query is submitted or in advance, e.g. using prepared statements or any other form of pre-optimized queries like query warehousing.

statistics

Why would the query optimizer need statistics about the data?

The query optimizer requires statistics about the data stored in the database as the decision for an optimal executable program may depend on those statistics. For instance,

in order to decide for an SQL query whether to access rows in a table through a full scan or an unclustered index depends on the selectivity of the query. If we keep meaningful statistics for the database, e.g. histograms of data distributions, we may estimate or even precisely compute that selectivity and hence make a better decision.

Where does the database obtain those statistics from?

Statistics are derived from the database contained in the database management system. An important consideration is whether statistics are computed and updated automatically or whether the user (or a database administrator) has to trigger statistics collection manually. Make sure to check out the manual of your database product to be sure that statistics are collected and updated regularly. If not, the query optimizer will not be able to come up with optimal executable programs.

query plan interpretation

What is query plan interpretation?

In query plan interpretation a program is generated using pre-compiled modules (called operators). Multiple operators are combined to form a query plan. A query plan is a directed acyclic graph (DAG) where the nodes are operators and the edges implement any variant of pipelining (typically an iterator-interface). That query plan is than interpreted by an executable program called the query plan interpreter (also called ONC-interface). Query plan interpretation is typically used by disk-based systems. In these systems, the CPU-overhead of interpretation is negligible compared to the I/O-costs. In a main-memory system, query plan interpretation is a bad choice as the overheads for interpretation may overshadow the total query execution costs. Make sure to not confuse query plan interpretation with interpretation used in scripting languages (like Python or PHP). In query plan interpretation, interpretation happens on a much more coarse-granular level (modules or operators) whereas scripting languages need to interpret every line of code.

query plan

code generation

What is code generation?

Code generation means that the query optimizer generates an executable program without relying on pre-compiled modules like operators. Also, in contrast to query plan interpretation where we generate an operator DAG that is then executed by the query plan interpreter, in code generation, we generate an executable. Any programming language may be used to generate code, e.g. C++, assembly or LLVM.

programming language compilation

What is the relationship between query optimization and programming language compilation

The high-level task of a programming language compiler is to compile a program written in a specific programming language (e.g. Java, C++) into an executable binary. A query optimizer has a very similar task: it translates a program (a declarative SQL query) into an executable binary. Hence, query optimization can be considered a domain-specific compilation problem. A major difference to general purpose compilation is that in query optimization the variety of input programs is much more restricted. This allows us to apply domain-specific optimizations, e.g. join enumeration or tuple reconstruction. In addition, more precise statistics, e.g. statistics on data distributions, are available and can be considered for the optimization and compilation process. Notice that even though

the problem is domain-specific, this does not imply that the problem gets much simpler.

Quizzes

1. What is the task of the query optimizer in a database system?
   (a) parsing queries
   (b) answering the query
   (c) defining a plan to efficiently answer a query
   (d) gathering statistics of the database
2. What is the output of the query optimizer?
   (a) The result of the given query
   (b) The parse-tree of the query
   (c) A plan (algorithm) that tells how a query should be executed efficiently
   (d) Statistics of the tables touched by the query
3. What kind of information do you think is useful to the query optimizer?
   (a) Distribution of the attribute values (e.g. histograms)
   (b) Record count for each table
   (c) The number of user accounts in the database
   (d) Buffer sizes

5.1.2 Challenges in Query Optimization: Rule-Based Optimization

Material

Video:	Original Slides:	Inverted Slides:

Learning Goals and Content Summary

canonical form

What is the canonical form of a query?

If we translate a SQL query into relational algebra using only the base operators of relational algebra (cross product, selection, projection, union, minus, and rename), we get the canonical form of an SQL query. This canonical form serves as the input to further optimizations. It is the most basic (and trivial) form of a query plan.

query plan

DAG

When would the canonical form be a DAG (directed acyclic graph)?

If any input relation (or subtree) is input to more than one other operator, we receive a DAG. For instance, assume a self-join, i.e. we join a relation R with itself. Then R is at the same time the right and the left input to a binary join. Therefore the canonical form of the query is a DAG.

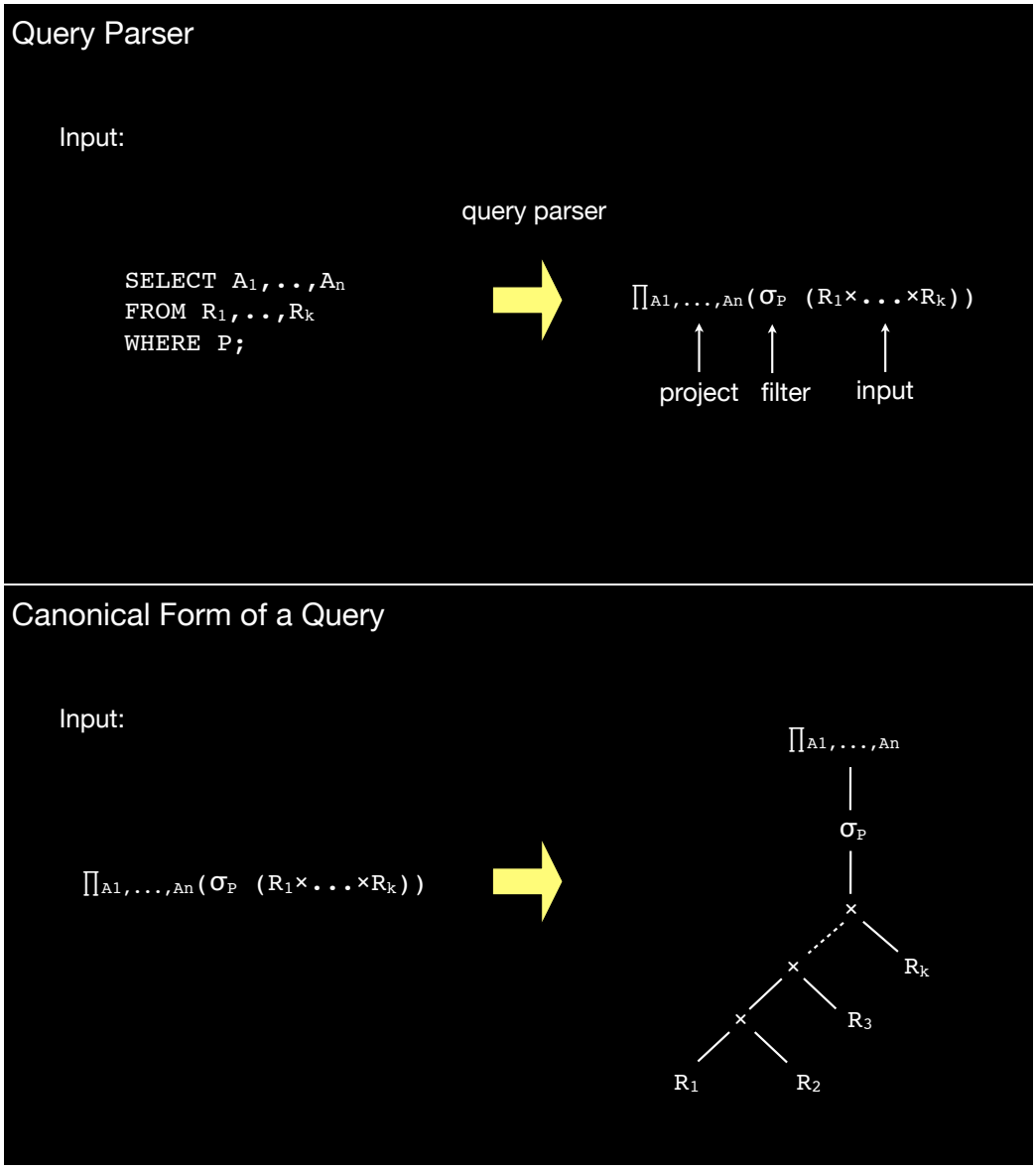


Figure 5.2: The query parser and the canonical form of a query

rule-based optimization

What does rule-based optimization do in principle?

In rule-based optimization we apply transformation rules to plans. A rule should be applied only if it matches a certain condition. For instance, a rule might match a condition where a selection is applied on a single table after applying a cross product. The rule should then rewrite the plan to push down the selection through the cross product to perform the selection before the cross product.

rule

What does a single rule do?

A single rule takes as its input a plan and emits a transformed plan. The transformed plan, when executed, produces the same result as the input plan.

What is the advantage of breaking up conjunctions of selections?

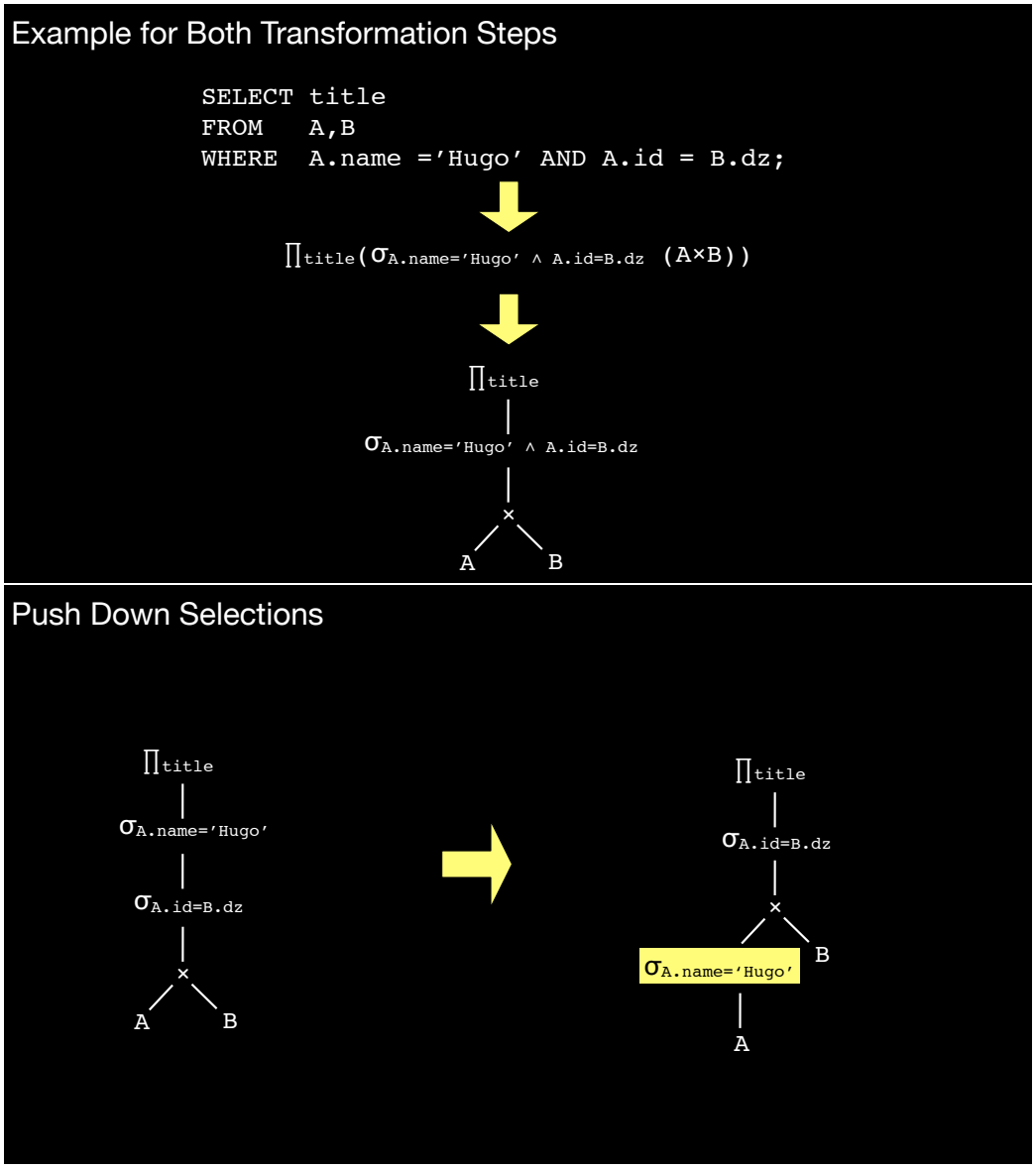


Figure 5.3: Example transformation from SQL to the canonical form; a query rewrite pushing down a selection predicate

If we break conjunctions into individual selections, this may enable us to apply additional rules, e.g. we may then be able to apply selection push-down.

push down

Why does it make sense to push down selections?

This makes sense as it allows us to remove data from query processing as early as possible. Hence, that data does not have to be transmitted or processed by other operations. Selection push-down is a very generic technique that appears in many places of query processing, e.g. distributed query processing.

Why should we prefer joins over cross products whenever possible?

In most cases, except pathological ones where joins are not selective, a join has a much

better runtime complexity, typically $O(n)$ or $O(n \log n)$, than a cross product, $O(n^2)$. Cross products are only useful for very small inputs, if runtime is not critical or for testing purposes.

When is it possible to introduce a join?

If a cross product is followed by a selection using a predicate that is defined on both inputs of the cross product, we may rewrite the cross product and the selection to a join.

When is it possible to push down projections?

Projections may be pushed-down through an operator if that operator does not require attributes which are removed by that projection. For instance, if a join is defined on an attribute A but the projection does not preserve A, we may not push it down. Still, we may push it down by increasing the attribute list of the join to also include all additional attributes required by the operator.

What does the data layout of the store have to do with pushing down projections?

data layout

Processing data in a row store is (and should) be very different from processing data in a column store. In general, in a row layout it is somewhat easier to push-down selections and hence filter out tuples. In contrast, in a column store, it is easier to push-down projections and hence filter out entire columns. See also Section 5.3.5.

What are the most important rules?

The three most important rewrite rules are:

1. push down selections and projections
2. combine selections and cross products into joins
3. insert additional projections

Quizzes

1. What is the main task of the rule-based optimizer (RBO)?
   (a) To apply certain pre-defined rules that could optimize the query plan
   (b) To analyze the current workload and optimize the query plans based on that information
   (c) To efficiently parse the queries
2. Given two queries Q1 and Q2. Assume a rule-based optimizer is called on Q1 and then on Q2. The result of the calls may lead to different plans even though
   (a) the strings of Q1 and Q2 are equal.
   (b) the strings of Q1 and Q2 are equal, except: some constants in the WHERE-clauses differ.
   (c) Q1 is semantically equivalent to Q2, i.e. it returns the same result set on any instance of the database having the same schema.

(d) Q1 returns the same result set as Q2.

3. Could the rule-based optimizer actually have a negative impact on query performance rather than a positive one?

   (a) Yes

   (b) No

5.1.3 Challenges in Query Optimization: Join Order, Costs, and Index Access

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Access Path
[LÖ09], Join Order

Learning Goals and Content Summary

join order

What is join order?

For a query plan that contains multiple joins, we often have a choice in which order we execute the different join operations. For instance, assume three input relations A, B, and C and a query $A \bowtie C \bowtie B$. We may execute this plan as either $(A \bowtie C) \bowtie B$, i.e. the join of A and C is executed first, or alternatively $A \bowtie (C \bowtie B)$ or $(A \bowtie B) \bowtie C$. Notice that a particular order assumes a suitable join predicate on all pairwise joins. If no such predicate exists, that pairwise join is equal to a cross product and hence it is likely that the particular join order does not have to be considered for query optimization.

What may be the effect of picking the wrong join order?

If we pick an unsuitable join order, we may end up with a slow query plan. See also Figure 5.4. In this example, we consider the sizes of intermediate results produced by the joins as our optimization criterion. The join $C \bowtie B$ produces a large intermediate result set which then serves as the input to the join with A. In contrast, the join $A \bowtie C$ produces a relatively small intermediate result. Hence, the input to the join with B is smaller. For this example, the join order $(A \bowtie C) \bowtie B$ is optimal w.r.t. to the number of intermediate results produced. Notice that the third plan $(A \bowtie B) \bowtie C$ is assumed to be non-selective and hence we did not consider it.

Is the join order problem only relevant for joins?

No, the "join order" problem exists for all binary operations that are commutative and associative, e.g. union, intersection, and cross product. In particular for intersections, e.g. in the context of a search engine intersecting IDs from different posting lists, the

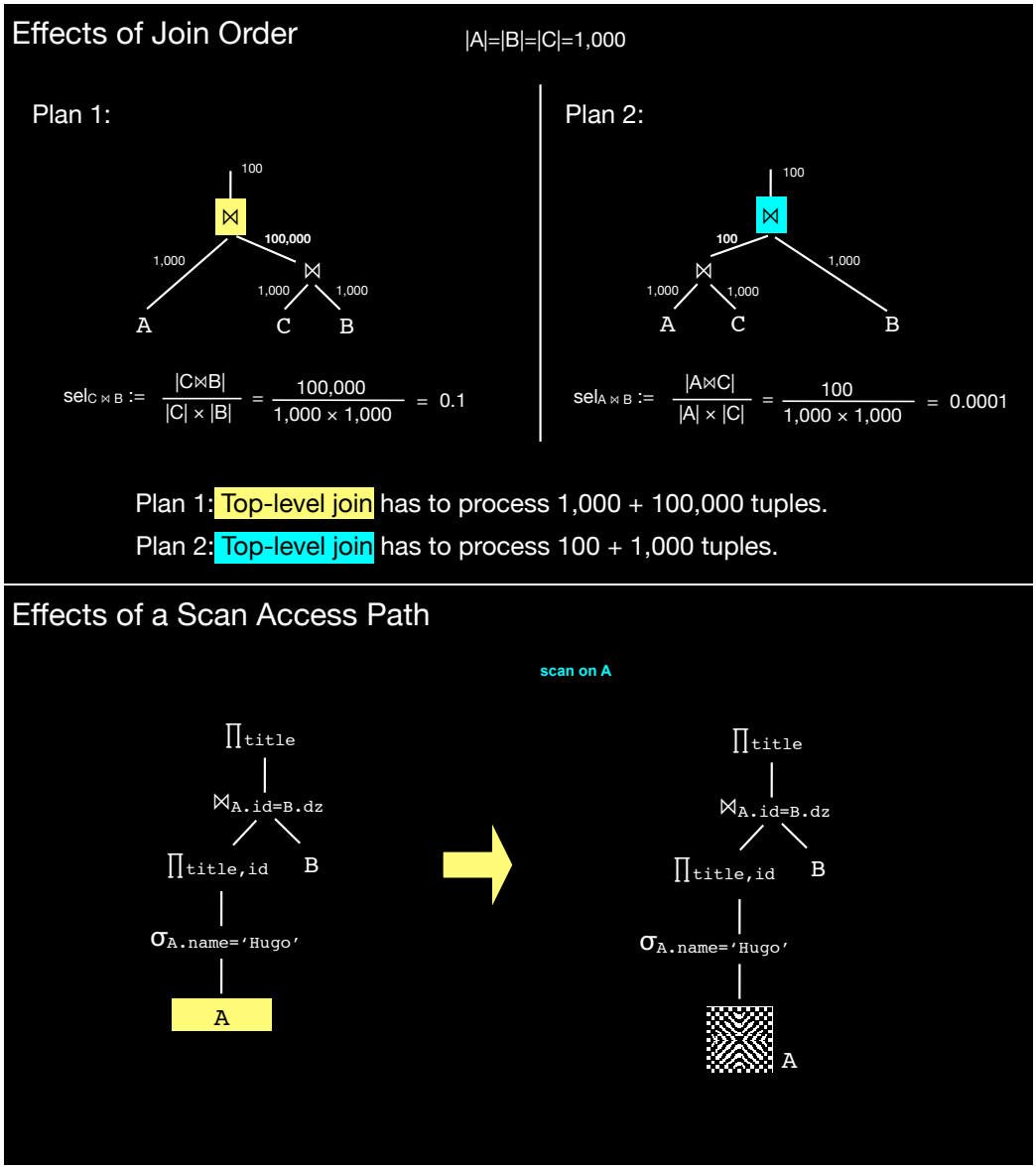


Figure 5.4: The possible effects of join orders and access paths

order of intersections may make a difference.

How do we pick the right access path?

access path

Unfortunately, this is not as easy as we might imagine. For instance, if we have the choice of scanning a table or using an index, it sounds like a trivial decision: let's use the index as it is computationally cheaper than any scan. However, like that we may end up with a query that is orders of magnitude slower than a simple scan. How is that possible? The issue is again the I/O-costs of a particular access method rather than the computational costs. The I/O-costs may overshadow the overall costs of a query by orders of magnitude. Hence, in particular for an unclustered index which may trigger many random accesses to the database store, we should be careful with our decision for a particular access

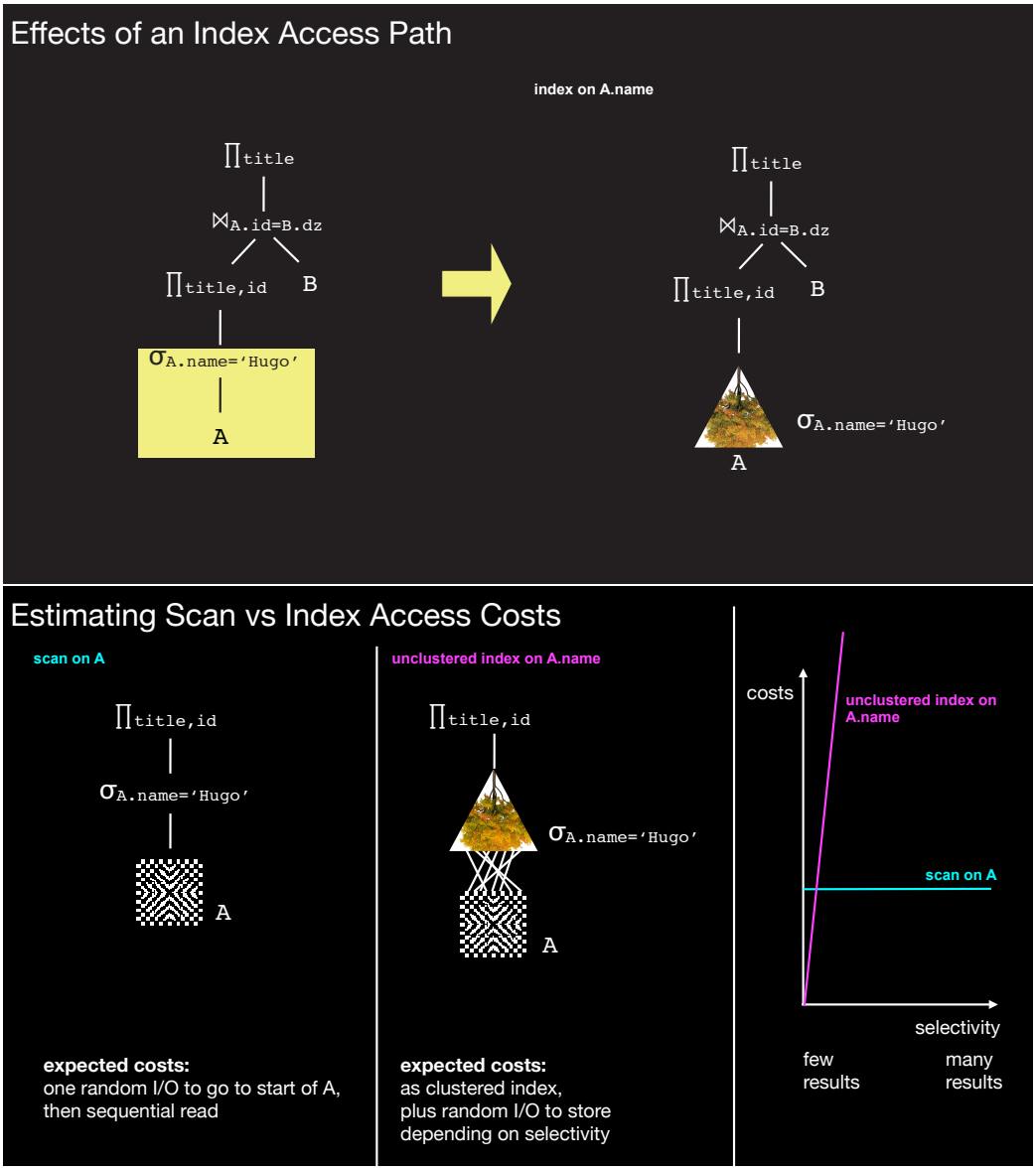


Figure 5.5: Effects of an index access in a plan; estimating costs for scans vs estimating costs for index accesses

plan. Unclustered indexes are only competitive for highly selective queries (few tuples 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?

See Table 5.1 for a textual overview of the estimated costs. These costs can then be

Scan	**Clustered Index**	**Unclustered index**	**Covering index**
one random I/O to go to start of A, then sequential read	one random I/O to fetch leaf, one random I/O in store, then ISAM	as clustered index, plus random I/O to store, depending on selectivity	one random I/O to fetch leaf, then ISAM, no need to go to store

Table 5.1: estimated I/O-costs of different access methods

expressed by a cost model. That cost model must be calibrated to reflect the reality of the system, i.e. we need to benchmark the performance characteristics of the underlying hardware. Those measurements should then be used to fine-tune the cost model. Recall that:

> Essentially, all models are wrong, but some are useful.
>
> [George E. P. Box]

Why should we be very careful with unclustered indexes?

Unclustered indexes are very fragile w.r.t selectivity estimates. This means, if we estimate an index to return 100 result tuples, but in reality the index returns 200 result tuples, this difference may be enough to slow down the final query substantially. Hence, if in doubt: do not use an unclustered index.

What may happen if the selectivity estimates of a query are wrong?

selectivity estimate

Again, we may end up with a query plan that is by orders of magnitude slower than the optimal plan.

Quizzes

1. When performing a join operation does the order of the operands have any impact on the performance of the join in general?
   (a) Yes
   (b) No
2. When the query optimizer decides to use an index for retrieving data for operands in a certain operation, say a join, would it be possible that the use of this index slows things down?
   (a) Yes
   (b) No

3. Assume you have to perform a query that has a very low selectivity on attribute A, i.e. the size of the output is not much smaller in comparison to the input size of the query. The database ships with a physical design advisor. It suggests to build an unclustered index on A (this index is not currently present in the system). Does the suggestion of the physical design advisor make sense to speed up future queries that are similar to the current one?

   (a) Yes, definitely

   (b) Rather not

4. When the query optimizer is trying to decide how to perform a join operation, what kind of optimization fits better?

   (a) Cost-based optimization

   (b) Rule-based optimization

5.1.4 An Overview of Query Optimization in Relational Systems

Material

Literature:
[Cha98] Sections 1–4.1.1

Additional Material

Literature:
[LÖ09], Query Optimization
[LÖ09], System R (R*) Optimizer
[LÖ09], Query Processing

Learning Goals and Content Summary

What should we be aware of when reading this article?

We should be aware that this article describes the state-of-the-art in query optimization as of 1998. In addition, this article focusses on disk-based systems. Query optimization for main-memory and other systems may be very different. However, in order to understand how query optimization works for the latter, it is is important to understand these basic techniques first.

physical operator

What is a physical operator?

A physical operator is a software entity operating on one or multiple input streams and producing one or multiple output streams. In the context of a relational database system a physical operator is considered a specific algorithm and/or implementation of a

logical operator

logical operator. A logical operator is any of the relational algebra operators, e.g. join, projection, intersection, etc. A physical operator is one possible implementation of a logical operator. For instance, the logical operator *join* may be implemented as a physical operator taking two input streams and producing one output stream. Algorithmically,

we may use either a hash-join (see Section 4.1.2), a sort-merge join (see Section 4.1.3) or a co-grouped join (see Section 4.1.4). Hence, we have a least three different physical join operators. Most databases allow you to inspect the physical plan generated for a given SQL query and also display the physical operators used in that plan. In most database systems this can be done by writing `EXPLAIN` in front of the SQL-command.

algebraic representation

What is the algebraic representation of a query?

logical plan

An algebraic representation of a query (aka logical plan) is any suitable representation of relational algebra, e.g. rather than writing $(R \bowtie S) \bowtie T$ we may write `join(join(R,S),T)`. However, that representation should not contain information on physical properties, i.e. the specific algorithms used to implement the algebraic operations. The latter is done in a physical plan.

cost estimation

What is a cost estimation in the context of the query optimizer?

query optimizer

The cost-based query optimizer estimates the costs of (possibly) a large number of plans. Based on these estimates the query optimizer chooses one plan for execution. Recall our discussion in Section 5.1.3.

search space

What type of search space did System-R use?

System-R

The search space of System-R focussed on linear trees (aka left-deep trees).

linear trees

What is a left-deep tree?

left-deep tree

Given a sequence of input relations $R_1, \ldots, R_n$, a left-deep tree starts by a binary join on R_1 and R_2. Then it iteratively adds binary joins $R_3, \ldots, R_n$.

bushy tree

What is a bushy tree?

A bushy tree is a tree that is not left deep, i.e. at least for one of the joins in that plan its right input is not an input relation but another join operation. Notice that the union of the set of bushy trees and the set of left-deep trees forms the set of all possible trees.

set of bushy trees

set of left-deep trees

What must be considered when applying transformations?

We can apply transformations...

1. heuristically, i.e. we assume that the transformation always improves the plan. In that case, plan enumeration is applied only after applying certain transformations. A hidden assumption here is that even after applying those transformations, we can still find the globally optimal plan. However this may not be true in all cases. A transformation may rewrite a plan such that only a locally optimal plan is reachable in cost-based enumeration.

2. cost-based, i.e. transformations are used as part of cost-based plan enumeration in the first place. Like that we can find a globally optimal plan, however, we need to estimate the costs of a larger number of plans.

interesting order

What is an interesting order?

Originally, an interesting order described a situation where one operator, e.g. a sort operator, produces its output ordered by a particular attribute. That order could then

be exploited by one of the next operators. For instance, assume we do a sort-based grouping followed by sort-merge join. Further assume that both grouping and join are on the same attribute. In that situation, we do not need to sort the output of the sort-based grouping again in order to perform the merge-join as they are already sorted. We exploit the 'interesting order' produced by the grouping for the following join. In the general case, any interesting physical property produced by a subplan may be useful and possibly exploited by subsequent operators. Other examples of exploitable physical properties that either exist before executing a query or are created as a side-effect of query processing include: partitioning, partially sorting, and indexing.

interesting physical property

bushy join

intermediate relation

Do bushy join sequences require materialization of intermediate relations?

In Section 4.1.1. of the article, you find the sentence "Bushy join sequences require materialization of intermediate relations." Well, this is not entirely true. A simple counter example is the case where all joins are merge joins on the same attribute. This leads us to a more general question: when to materialize what exactly in a query plan? We can materialize in-between physical operators, inside physical operators (sometimes we materialize entire relations inside an operator!) or we stop thinking along physical operators altogether. We will look at this in Section 5.3 in more detail.

Quizzes

1. What is the main task of the rule-based optimizer (RBO)?
   (a) To apply certain pre-defined rules that could optimize the query plan
   (b) To analyze the current workload and optimize the query plans based on that information
   (c) To efficiently parse the queries
2. Given two queries Q1 and Q2. Assume a rule-based optimizer is called on Q1 and then on Q2. The result of the calls may lead to different plans even though
   (a) the strings of Q1 and Q2 are equal.
   (b) the strings of Q1 and Q2 are equal, except: some constants in the WHERE-clauses differ.
   (c) Q1 is semantically equivalent to Q2, i.e. it returns the same result set on any instance of the database having the same schema.
   (d) Q1 returns the same result set as Q2.
3. Could the rule-based optimizer actually have a negative impact on query performance rather than a positive one?
   (a) Yes
   (b) No

Exercise

You are given the following query:

```
SELECT A.b, B.c, C.d, D.e
FROM   A, B , C , D
WHERE  A.x=B.x AND A.a<24 AND A.c=C.d AND B.d=D.b
       AND D.a>24 AND C.d=D.c AND C.b>45 AND B.x
          =23
```

(a) Create a relational algebra tree using only cross products, selections, and projections for this query.

(b) Use rule-based optimization to improve the logical plan. Explain what you did and draw the new plan as a relational algebra tree.

5.2 Cost-based Optimization

5.2.1 Cost-Based Optimization, Plan Enumeration, Search Space, Catalan Numbers, Identical Plans

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
see 5.1.4

Further Reading:
W Definition from Wolfram
W Definition from WIkipedia
[OL90]

Learning Goals and Content Summary

What is the overall idea in cost-based optimization?

The overall idea of cost-based optimization is to:

1. enumerate the set of all plan alternatives,
2. then for each plan estimate the costs of executing it,
3. pick the plan with the lowest estimated costs and execute it.

What is the number of possible left-deep plans for n input relations?

The number of possible left-deep plans for n input relations is $n!$.

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:

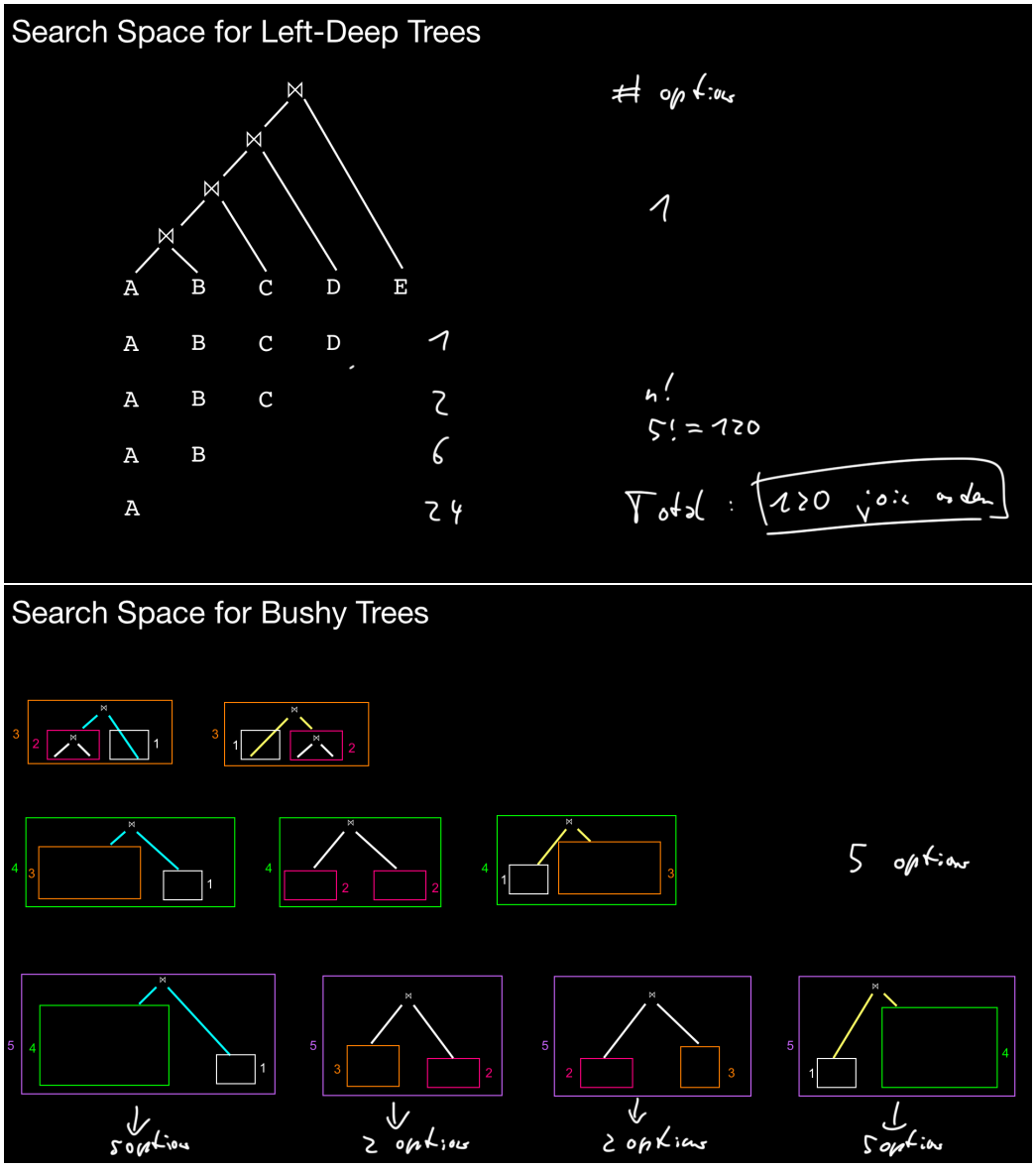


Figure 5.6: Search space for left-deep and bushy trees

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

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 the sequence 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, ...

What is the hidden assumption in this calculation when counting bushy plans?

The hidden assumption in this calculation is that all binary joins are asymmetric and hence cannot be commuted, i.e. $R \bowtie S$ is assumed to be always different from $S \bowtie R$. Notice that this is not necessarily always the case, e.g. `SortMergeJoin(R,S)` = `SortMergeJoin(S,R)`. Similar symmetries may exist for partitioning joins.

Quizzes

1. What is the rough idea of a cost-based query optimizer?
   (a) Explore the search space to obtain the optimal way of performing the given operation under a given cost model.
   (b) Apply certain pre-defined rules that are known to potentially give good performance
2. What is the (exact) total number of options when the query optimizer considers left-deep trees? Assume n is the number of operands involved and that the operation is commutative and associative.
   (a) 1
   (b) 4^n
   (c) n
   (d) $n!$
3. Can a bushy plan degenerate into a left-deep plan?
   (a) Yes
   (b) No
4. What is the (exact) total number of options when the query optimizer considers bushy plans? Assume n is the number of operands involved, and that the operation is commutative and associative. C_n denotes the n-th Catalan number.
   (a) C_{n-1}
   (b) $n! \cdot C_{n-1}$
   (c) $C_{n-1}!$
   (d) $n^{C_{n-1}}$
5. Which of the following parenthesization describes left-deep plans? The symbol () represents a binary operation and A, B, C, D, E are the operands
   (a) (((A,B), C),(D,E))
   (b) ((((A, B), C), D), E)
   (c) (((A, B),(C, D)), E)
   (d) (A, (B, (C, (D, E))))

6. Assume you have two different plans that produce the same number of intermediate results. Should you be careful when choosing which algorithm to use to perform the corresponding join operations?

   (a) Yes

   (b) No

5.2.2 Dynamic Programming: Core Idea, Requirements, Join Graph

Material

Video:	Original Slides:	Inverted Slides:
		(slides for 5.2.2, 5.2.3, and 5.2.4)

Additional Material

Literature:
V Recursion with memoization
[LÖ09], Query Optimization (in Relational Databases)

Further Reading:
W Dynamic Programming

Learning Goals and Content Summary

join graph

What is the join graph?

The core idea of a join graph is to represent the join predicates for a given query. The join graph allows us to restrict join enumeration to plans without cross products. A join graph is defined as $G = (V, E)$ where the set of vertices V represents all relations referenced in the query and the set of edges E represents all join predicates (also inferable join predicates).

As any pair of relations may trivially be "joined" through a cross product anyway, a join graph may be considered to be dense, i.e. all edges are at least set to the join predicate $JP(x, y) = true$ or to a (non-trivial) selective join predicate. However, typically only the non-trivial join predicates are considered and displayed in a join graph. Thus, whenever there is a (non-trivial) join predicate on two relations, we represent that join predicate through an edge. In the video, we use thin white edges to visualize the trivial cross products and thick yellow lines to visualize nontrivial join predicates.

dynamic programming

What is the core idea of dynamic programming?

The core idea of dynamic programming is to construct larger optimal solutions by assembling smaller optimal solutions. In other words, dynamic programming works best if an optimal solution to a problem may be constructed by first solving smaller subproblems which may then be combined to solve a bigger problem.

What are the two requirements for dynamic programming ro make sense?

There are two requirements:

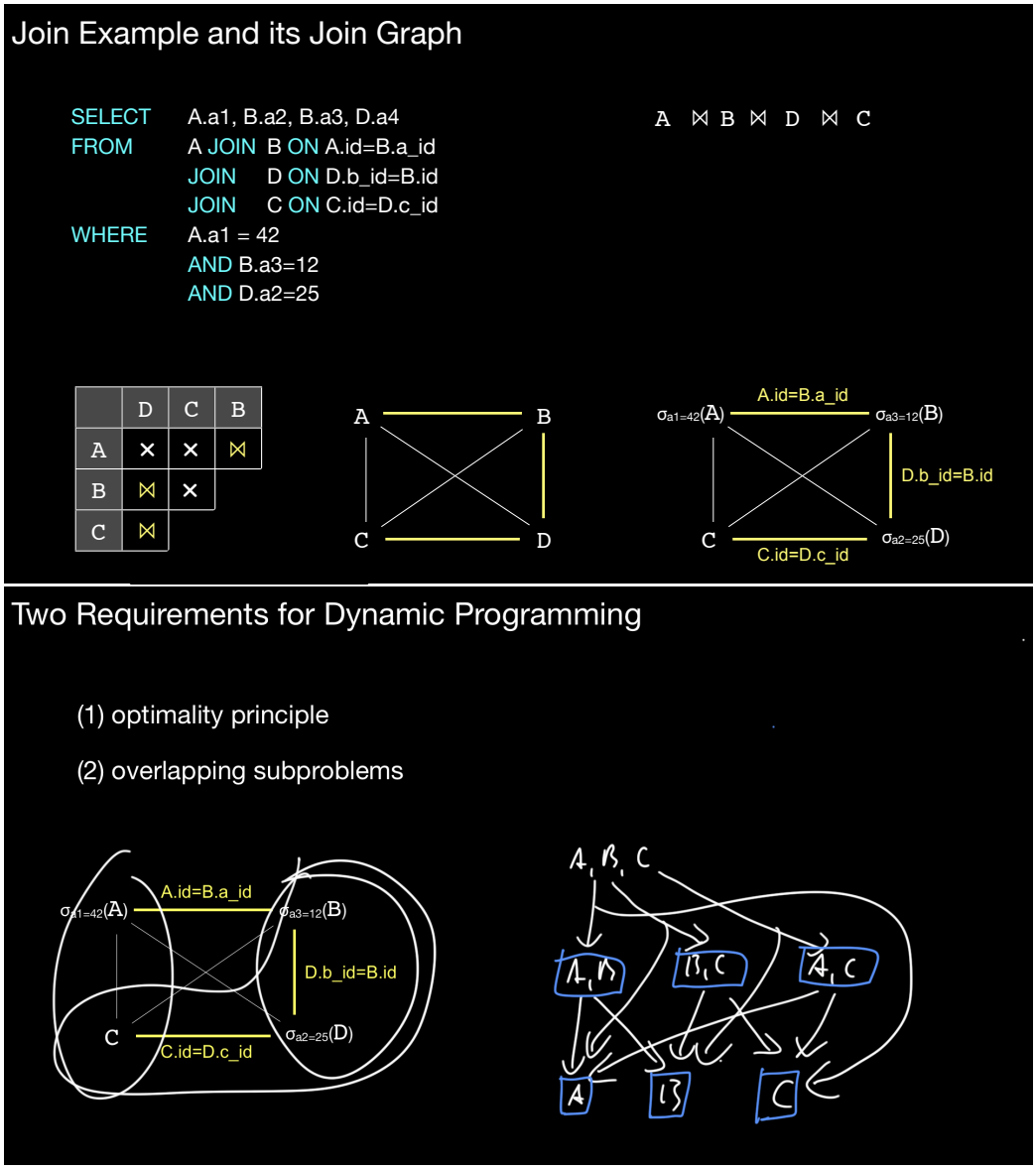


Figure 5.7: The join graph and its relationship to dynamic programming

1. **Optimality principle.** This means that an optimal solution can be found by assembling optimal solutions to smaller subproblems. In the context of plan enumeration, this means that in order to find an optimal join order to a query referencing relations $R_1, \ldots, R_n$, we may find that optimal join order by first computing optimal solutions on subsets of the relations $R_1, \ldots, R_n$ and then composing those subsets into bigger sets until eventually we find the optimal solution for the entire set of relations. Notice that this principle **does not** imply that **any** combination of subsets leads to the optimal plan. For instance, finding optimal smaller solutions on relations $R_1, \ldots, R_k$ and $R_{k+1}, \ldots, R_n$, for $1 < k < n$, only and then combining them may not find the optimal solution. This may be the case if the optimal solution contains R_1, R_n as an optimal sub-solution.

2. **Overlapping Subproblems.** This means that we face a situation where the subproblems overlap, i.e. the subproblems are not disjoint. If subproblems do not overlap, i.e. they are disjoint, we could directly use divide-and-conquer. Consider the case of partitioning (like in Quicksort): once we partitioned the data into partitions, each partition may be treated independently (e.g. by a separate thread). Once all partitions have been treated, we simply combine the results. This principle is applied recursively in Quicksort.

   In contrast, when dealing with overlapping subproblems, we cannot partition the problem into smaller disjoint subproblems. In other words: there (potentially) exist multiple partitionings of the same problem.

   This is exactly the case in join enumeration: assume a query referencing relations R_1, R_2, R_3. Let's assume we simply want to compute the optimal join order (we do not even consider physical join algorithms and we assume that all joins are symmetric). Then, we must not simply combine $R_1 \bowtie R_2$ (an optimal subplan) with R_3 as this would be one random subplan of size two combined with the remaining table. Using this optimal subplan may not necessarily result in an optimal plan for the entire join operation. Hence, in order to find the optimal join order among R_1, R_2, R_3 we also have to consider the optimal subplans for $R_2 \bowtie R_3$ and $R_1 \bowtie R_3$.

optimal subplan

What is an optimal subplan?

Assume a query referencing relations $R_1, \ldots, R_n$. A subplan SP references a real subset S of those relations, i.e. $S \subset \{R_1, \ldots, R_n\}$. We call SP optimal if no other subplan with lower costs exists for S.

Optimal subplans are the building blocks of dynamic programming. We may use them as we assume that join orders may be computed relying on the optimality principle explained above (in Section 5.2.3, we will see a situation where the optimality principle does not hold).

Quizzes

1. What are the requirements for applying dynamic programming?
   (a) the optimal solution can be constructed from the optimal solutions of its subproblems
   (b) smaller subproblems should appear multiple times in larger subproblems
   (c) the join graph is connected
2. When solving a problem using dynamic programming, are subproblems that are encountered more than once recomputed each time?
   (a) No
   (b) Yes
3. Dynamic programming solves a problem by breaking it into smaller pieces. What is then the difference between dynamic programming and a divide-and-conquer algorithms?

(a) no difference at all

(b) with dynamic programming we can optimally solve problems with dependent subproblems

(c) with divide and conquer we usually solve problems with independent subproblems

5.2.3 Dynamic Programming Example without Interesting Orders, Pseudo-Code

Material

Video:	Slides:
	see 5.2.2

Additional Material

Literature:	Further Reading:
see 5.2.2	see 5.2.2

Learning Goals and Content Summary

How does dynamic programming work when applied to the join enumeration problem?

dynamic programming

join enumeration

Assume a query on input relations $R_1, \ldots, R_n$. In the join enumeration problem, we apply dynamic programming as follows:

1. for each input table $R_1, \ldots, R_n$ compute the set of subplans that access a single input table
2. for each subset: only keep the subplan with the lowest estimated costs
3. set $plansize = 2$
4. compute all subplans that access a subset of size *plansize* from the input relations $\{R_1, \ldots, R_n\}$; use smaller plans (already stored in previous steps) as building blocks to compute those plans
5. for each subset: only keep the subplan with the lowest estimated costs
6. if *plansize* is equal to n, we are done, else: increase *plansize* by one and goto step 4.

What does the pruning step do?

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.

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

iteration

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

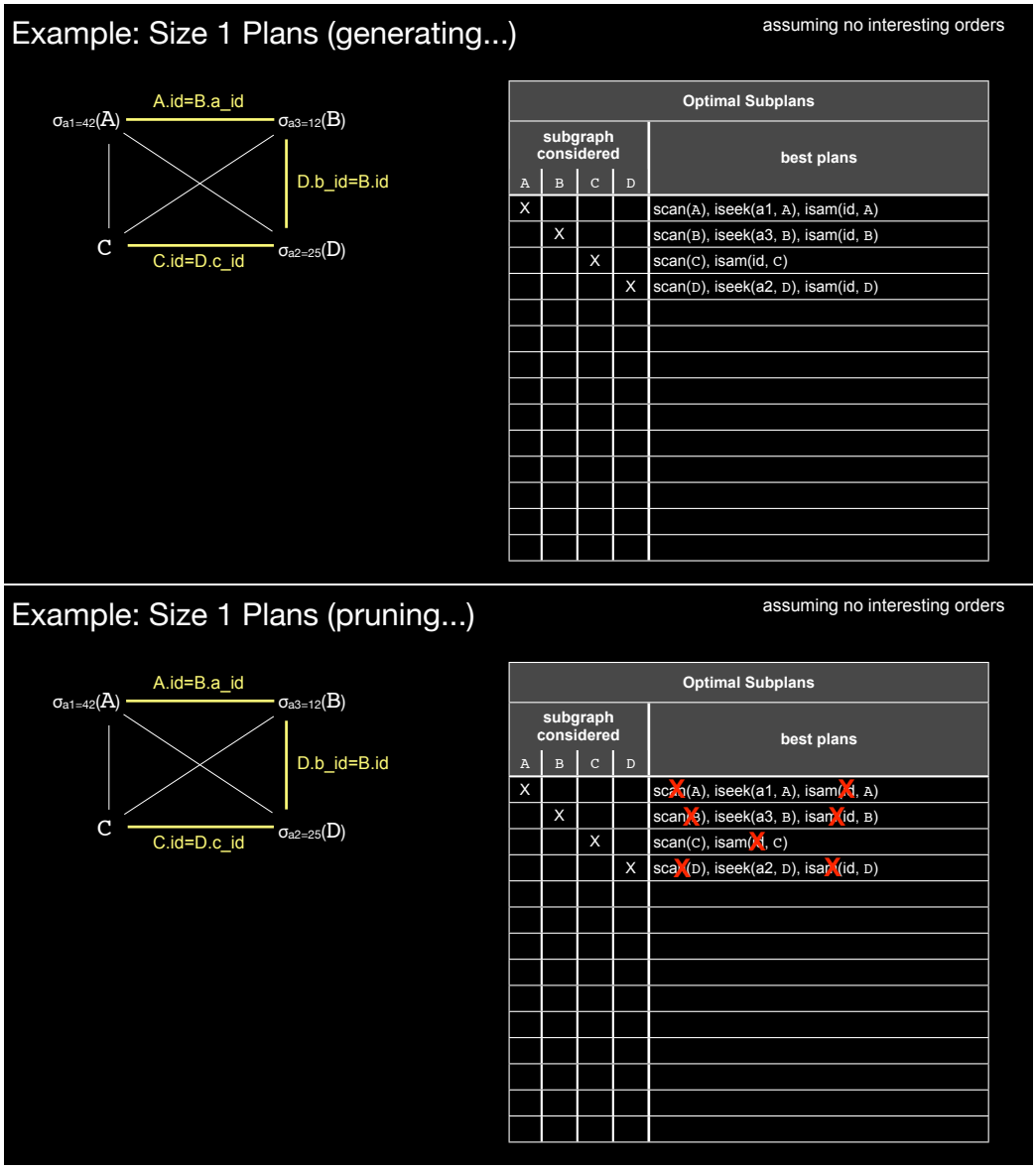


Figure 5.8: Dynamic programming: size 1 plan generation and pruning

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:

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

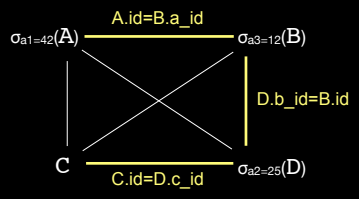
mergePlan()

What does the mergePlan()-method do? How would it merge two subplans?

Merging two disjoint subplans implies connecting those subplans through either a join (if a join predicate exists) or through a cross product. Recall that different join algorithms vary in costs. Hence, they must be considered in the costing of the merged plan.

Example: Size 3 Plans (pruned)

assuming no interesting orders



Optimal Subplans				
subgraph considered				best plans
A	B	C	D	
X				iseek(a1, A)
	X			iseek(a3, B)
		X		scan(C)
			X	iseek(a2, D)
X	X			iseek(a1, A) SHJ iseek(a3, B)
X		X		iseek(a1, A) CP scan(C)
X			X	iseek(a1, A) CP iseek(a2, D)
	X	X		iseek(a3, B) CP scan(C)
	X		X	iseek(a3, B) SHJ P iseek(a2, D)
		X	X	iseek(a2, D) SHJ scan(C)
X	X	X		(iseek(a1, A) SHJ iseek(a3, B)) CP scan(C)
X	X		X	(iseek(a1, A) SHJ iseek(a3, B)) SHJ iseek(a2, D)
X		X	X	(iseek(a2, D) SHJ scan(C)) CP iseek(a1, A)
	X	X	X	(iseek(a3, B) SHJ iseek(a2, D)) SHJ scan(C)

Dynamic Programming

```
costs( Plan ): ↦ Integer                                  //cost function estimating costs for a plan

DynamicProgramming( R1, ..., Rn, costs() ):               //relations Ri; cost function for pruning
    For i = 1 to n:                                       //i.e. all S ⊂ {R1, ..., Rn} with |S| == 1:
        optPlan[{Ri}] := AccessPlans( Ri );               //get all possible plans for Ri
        prune( optPlan[{Ri}], costs() );                  //prune set of plans using cost function

    For plansize = 2 to n:                                //for all subplans having at least two inputs
        ForEach S ⊂ {R1, ..., Rn} with |S| == plansize:   //inspect each proper subset S of that size
            optPlan[S] := ∅;                              //initialize optPlan[S] with empty set
            ForEach O ⊂ S:                                //inspect each proper subset O of S
                optPlan[S] ∪=                             //extend optPlan[S]-entry to contain...
                    mergePlans( optPlan[O], optPlan[S\O] ); //..the merged plan of two optimal subplans
                prune( optPlan[S], costs() );             //prune set of plans using cost function
    prune( optPlan[{R1, ..., Rn}], costs() );             //final pruning, i.e. pick the final plan
    return optPlan[{R1, ..., Rn}];                        //return the final plan
```

Figure 5.9: Dynamic programming: size 3 pruned plans and pseudo code

Quizzes

1. What paradigm does dynamic programming use?
   (a) Bottom-up
   (b) Top-down
2. Are cost models important for designing dynamic programming algorithms for databases?
   (a) Yes
   (b) No

3. Does dynamic programming work by building up a table, that has to be kept explicitly, containing the optimal solutions for the sub-problems met so far?

   (a) No

   (b) Yes

5.2.4 Dynamic Programming Optimizations: Interesting Orders, Graph Structure

Material

Video:	Slides:
	see 5.2.2

Additional Material

Literature:	Further Reading:
see 5.2.2	see 5.2.2

Learning Goals and Content Summary

dynamic programming

Why do we have to change the simple dynamic programming algorithm to consider interesting orders?

Dynamic programming relies on the optimality principle. This principle is violated if two subplans joining the same subset of relations produce different interesting orders. Hence, if we do not change the dynamic programming algorithm, we may not find the optimal plan.

When exactly would we change the dynamic programming algorithm?

When pruning subplans, we need to keep for each subset of relations **AND** different interesting order the best plan.

Hence, we change dynamic programming as follows (compared to the algorithm shown in Section 5.2.3). Assume a query on input relations $R_1, \ldots, R_n$. In the join enumeration problem, we apply dynamic programming as follows:

1. for each input table $R_1, \ldots, R_n$ compute the set of subplans that access a single input table
2. for each subset: **for each interesting order**: keep the subplan with the lowest estimated costs
3. set $plansize = 2$
4. compute all subplans that access a subset of size $plansize$ from the input relations $\{R_1, \ldots, R_n\}$; use smaller plans (already stored in previous steps) as building blocks to compute those plans
5. for each subset: **for each interesting order**: keep the subplan with the lowest estimated costs

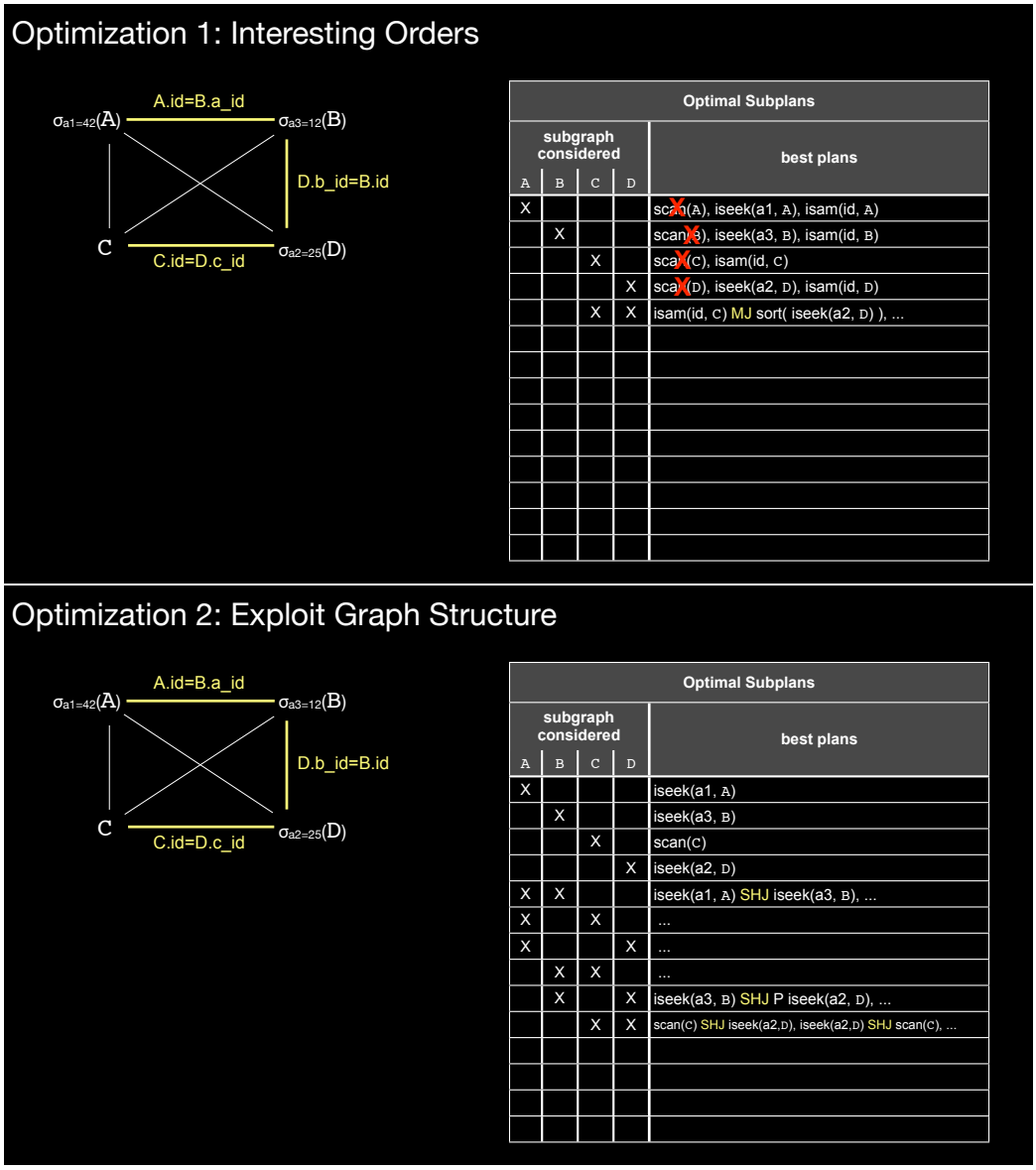


Figure 5.10: Dynamic programming: plan generation considering interesting orders and join graph

6. if *plansize* is equal to n, we are done, else: increase *plansize* by one and goto step 4.

Why would we exploit the join graph in dynamic programming?

The join graph allows us to detect whether a pair of relations can be combined through a selective join or through a cross product. Hence, we may exploit the join graph to never enumerate join orders containing cross products.

Quizzes

1. If the query optimizer does not take into consideration Interesting Orders when making decisions, would it be possible that its decision is actually not the best one?
   (a) Yes
   (b) No
2. Which of the following properties can be the benefit of choosing the right physical subplan?
   (a) Sort order
   (b) Query selectivity
3. Which of the following optimizations, when designing dynamic programming algorithms, are explained in the video?
   (a) Interesting/physical property
   (b) Greedily choosing plans
   (c) Take graph structure into account

Exercise

Assume you want to compute the following query:

```
SELECT A.b, B.c, C.d, D.e
FROM   A, B , C , D
WHERE  A.x=B.x AND A.c=C.d AND B.d=D.b AND C.d=D.
    c
```

You have no indexes available and the relations have the following sizes in tuples:

$$|A| = 20.000, |B| = 20.000, |C| = 20.000, |D| = 30.000$$

You are also given the following join selectivities:

$$sel_{A \bowtie B} = 0.01, sel_{A \bowtie C} = 0.02, sel_{B \bowtie D} = 0.02, sel_{C \bowtie D} = 0.03$$

You can assume the selectivities to be independent.

(a) How many different join orders are possible for this query?

(b) How many rows would you produce in the dynamic programming table to solve the join ordering problem if you include cross products and relation access plans?

(c) Perform dynamic programming to find the optimal join order. Take the join graph into account. Assume you have only simple hash join available and the following cost model:

$$C_{SHJ}(R \bowtie S) = |R| + |S|.$$

The cost of executing a join is just the number of input tuples it needs to process. Use a table with the following columns:

Subproblem	Cost	Plan	Outputsize

(d) Perform dynamic programming to find the optimal join order. Assume that only Grace hash joins (GHJ) and sort-merge join (SMJ) are available. The cost models are defined as follows: If a join has two inputs R and S, then

- The cost model for GHJ is:

$$C_{GHJ}(R \bowtie S) = 3 * (|R| + |S|).$$

- The cost model for SMJ is:

$$C_{SMJ}(R \bowtie S) = |R| * (\lceil \log_{1024}(|R|) \rceil + 1) + |S| * (\lceil \log_{1024}(|S|) \rceil + 1)$$

where $|R| * \lceil \log_{1024}(|R|) \rceil$ and $|S| * \lceil \log_{1024}(|S|) \rceil$ are the costs of sorting the inputs.

Use a table with the following columns:

Subproblem	Cost	Plan	Outputsize	Interesting Property

5.3 Query Execution Models

5.3.1 Query Execution Models, Function Calls vs Pipelining, Pipeline Breakers

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Pipelining

Learning Goals and Content Summary

What are the options for executing a query plan?

A query plan is not directly executable. We have to somehow translate it into executable code. There are several options for that including

1. function libraries: we wrap each physical operation, e.g. a hash join, into function. Then, every query plan can simply be translated into nested function calls. For

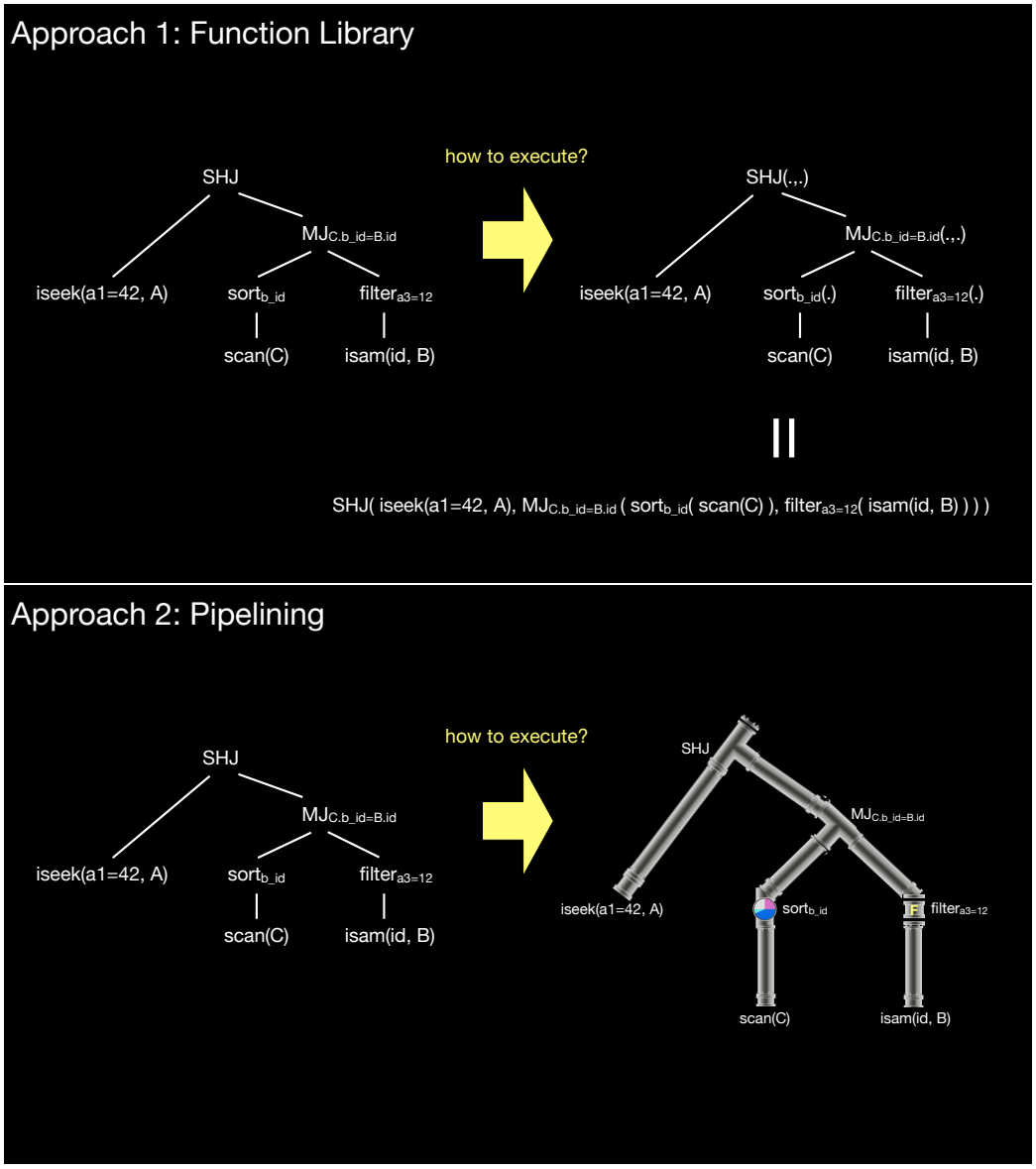


Figure 5.11: Translating a query plan to function calls or some sort of pipeline

instance, we implement a filter as a function, a join as a function, as well as a scan. Like that we could treat a join on two relations R1 and R2 as a call:

join(filter(scan(R1), "a=42"), filter(scan(R2), "b=2"), "R1.ID=R2.x");

In this approach, function calls whose output is used by more than one other function, i.e. the query plan is a DAG, may be materialized explicitly and then passed 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.

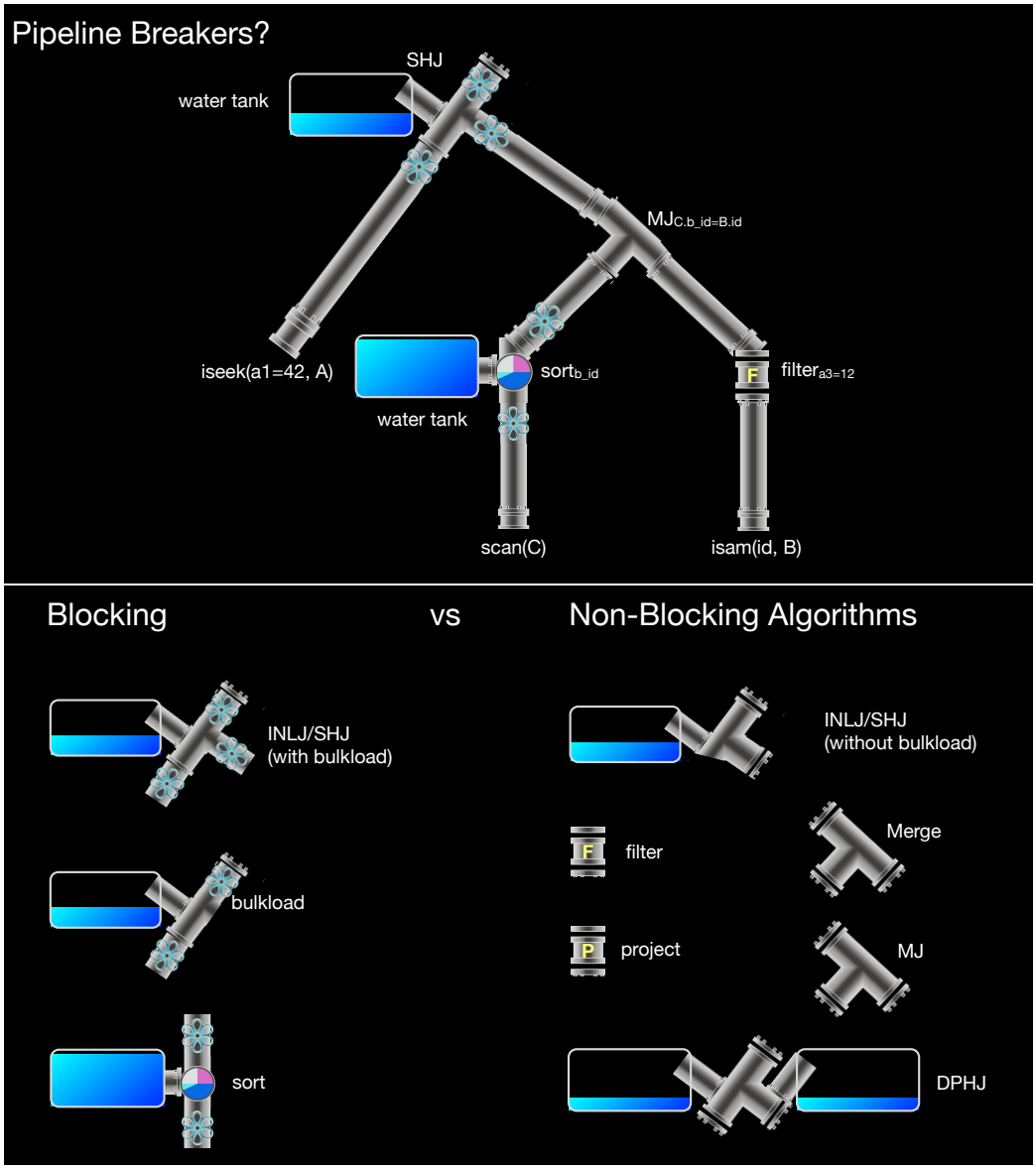


Figure 5.12: Blocking and non-blocking algorithms used in query pipelines

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.

What is the core idea of pipelining?

Notice that by *pipelining*, we do not mean one specific implementation like Unix pipes. We rather consider pipelining a generic concept which may be implemented in many different

ways. The core idea of pipelining is to allow functions to return some of their results before running to completion. Whether a function can in principle return some of its result before completion depends on the type of computation performed by the function. For instance, a min()-function, which may be implemented by scanning its entire input to return the minimal element, cannot return its result before consuming its entire input to completion (as the non-consumed input may always have another element that is smaller than the ones already consumed).

pipeline breaker

What is a pipeline breaker?

A pipeline breaker is a function that may only return a (partial) result after consuming its entire input to completion. Examples of pipeline breakers include: grouping, aggregation, sorting, and partitioning. In contrast, some functions can be streamed perfectly, i.e. they inspect one tuple from the input and immediately produce 0, 1 or many results. Examples include: filter (selection), projection, and map.

Why is it important to consider pipeline breakers in query processing?

It is important to be aware of pipeline breakers as they implicitly materialize their entire inputs. Thus, in terms of pipelining, they do not have an advantage over simple function calls.

What are the three stages of simple hash join and index nested-loop join?

Both SHJ and INLJ have three stages each:

1. **reading:** they consume the left input (the build relation) entirely. In that process, implicitly, the entire left input is materialized.
2. once **the left input is exhausted**: using the buffered data, the algorithms bulkload a hash table or any other suitable index structure. Notice that this phase may be interleaved with step 1. For instance, in SHJ once an item has been read, it is typically directly, i.e. item-wise, inserted into the hash table.
3. **looping and probing:** the right input (the probe relation) is read item-wise, each item is considered a query against the previously built hash table/index. A possible result of each query may be returned immediately.

Hence, steps 1 and 2 are blocking. Step 3 is non-blocking.

What are the three stages of quicksort?

Quicksort has three stages:

1. **reading:** if quicksort does not operate directly on the data structure used in the previous step, e.g. an array, quicksort consumes its input entirely. In that process, implicitly, the entire input is materialized (and copied) in the algorithm.
2. **sorting:** once the input is exhausted, quicksort fully sorts the data.
3. **outputting:** the output is returned item-wise

Hence, steps 1 and 2 are blocking. Step 3 is non-blocking. Notice that steps 2 and 3 may overlap if the recursion in quicksort uses depth-first search, i.e. every time a leaf-partition has been fully sorted, data in that partition may be returned directly. However, this does not change the blocking behavior of quicksort: the algorithm may only return the first tuple of the sorted output, i.e. the min or max element of the input set, once the entire input has been consumed to completion.

What are the three stages of external merge sort and external merge sort?

External Merge Sort has three stages:

1. **reading (with run generation):** external merge sort consumes its input entirely. In that process, implicitly, the entire left input is materialized into runs, i.e. phase 0 of external merge sort.
2. **merging (without final merge):** once the input is exhausted, external merge sort performs all merges except the final merge on.
3. **outputting (with final merge):** the final merge is run and the output is returned item-wise

Hence, steps 1 and 2 are blocking. Step 3 is non-blocking. Notice that steps 1 and 2 may overlap, i.e. some runs may already be merged before all runs have been created.

What are the blocking and non-blocking building blocks of query pipelines?

blocking

non-blocking

Figure 5.12 summarizes the most important blocking and non-blocking building blocks used in query pipelines.

Quizzes

1. Once the physical execution plan for a given query has been determined, can the query be executed right away?
   (a) Yes
   (b) No
2. Is it in general a good idea to materialize intermediate results as they are created?
   (a) Yes
   (b) No

5.3.2 Implementing Pipelines, Operators, Iterators, ResultSet-style Iteration, Iteration Granularities

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Iterator
W Java Iterator interface
W Java ResultSet interface

Learning Goals and Content Summary

pipeline

How do we implement a pipeline, i.e. how do we get from the high-level idea of pipelining to a concrete implementation of pipelining?

There are multiple ways to implement pipelining. The most prominent one is unix-pipes, i.e. multiple shell commands connected can be connected via the '|'-symbol, e.g. 'foo | bar'. Here foo's output is piped to bar and used as its input. However, many more implementations of pipelining exist, e.g. any form of producer-consumer, operators, iterators, cursors, and java ResultSets.

What are the core ideas and purposes of pipelining?

The core idea of pipelining is to overcome the limitations of a simple function call which would deliver its result in one step only, i.e. after having fully completed. In contrast, pipelines allow us to forward a subset of the result to the next operation. This implies, in a pipeline like 'foo | bar', we may already forward a subset of foo's result to bar which in turn may start computing without waiting for foo to complete. This may serve multiple purposes: (1) better resource utilization, i.e. higher degrees of parallelization, (2) less memory consumption, i.e. intermediate results do not (always) have to be materialized, and (3) lazy evaluation and early termination, i.e. operations may not need their entire inputs to compute a results, e.g. assume you compute the minimum over an already sorted sequence.

operator interface

What is the core idea of the operator interface?

An operator interface allows us to return the result in chunks where each chunk represents a subset of the entire result. Each call to `next()` returns the next chunk. The `next()`-function should also be implemented in a way that this call triggers exactly as much work as is required to be able to return the next chunk, but not more (lazy evaluation).

lazy evaluation

hasNext()

What happens in a hasNext()-call to an iterator?

iterator

In `hasNext()` the iterator needs to determine whether a subsequent call to `next()` may return another chunk. In other words, the sole purpose of `hasNext()` is to determine whether the iterator may still deliver further result subsets or whether the iter-

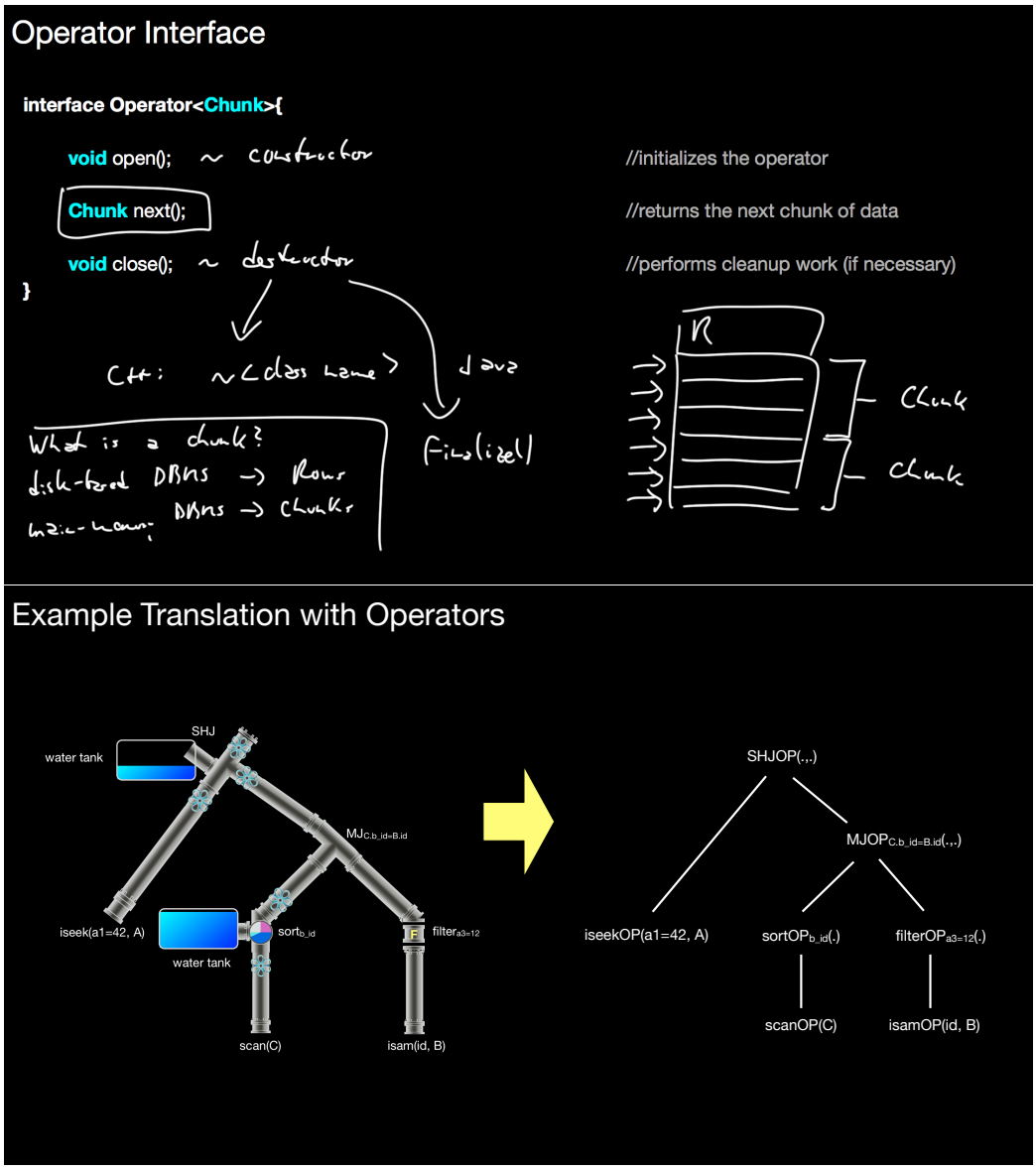


Figure 5.13: The operator interface and an example translation of a pipelining using operators.

ator is already exhausted (completed). Unfortunately, in order to compute the result to `hasNext()`, often the entire next chunk has to be computed anyways. Hence, in practice it is a better design choice to not use a `hasNext()`-method but rather let `next()` indicate whether the element returned is part of the result or indicating that the iterator/operator is exhausted.

What is a chunk and which special cases of chunks are important?

chunk

Again: a chunk represents a subset of the final result. Therefore, we have many different options to choose what a chunk represents. In a disk-based (volcano-style) pipeline each chunk may represent a tuple or a horizontal partition/set of tuples. In a main-memory system a chunk may represent an entire column or a horizontal partition of a column

(This is sometimes called vectorized pipelining. System examples include VectorWise.)

What would be a simple, textbook-style, translation of a plan using operators?

A simple translation would be to let each physical operation, e.g. a hash join or a sort-merge join, implement the operator interface. A join will require two input operators to obtain its input. In turn a join will feed its results to one parent operator. An example translation of a pipeline implemented by operators is shown in Figure 5.13.

ResultSet

What is ResultSet-style iteration?

JDBC uses ResultSets. This kind of interface is similar to iterators and operators, but not quite the same. The major difference is that in a ResultSet `next()` moves an internal pointer forward to the next result chunk. However, `next()` does not directly return that next chunk. In contrast, `next()` only returns a bool indicating whether the result set is exhausted or not. the actual data contained in a chunk has to be accessed through additional get*-methods. For instance, in JDBC chunks are considered to be rows. Attributes in a row are numbered. Hence, a call `getString(int attributeIndex)` returns the `attributeIndex`th attribute of the row as a string.

row

Are ResultSets restricted to rows or can we use it for columns as well?

column

JDBC assumes each chunk to represent one row. However, ResultSet-style iteration could also be used to consider each chunk a column, a horizontal partition of a row or any other suitable subset of the result set.

page

Would it make sense to iterate over pages?

Yes, absolutely. Assuming row layout, each page represents a horizontal partition (assuming no row crosses a page boundary). In a column layout, each page represents a horizontal partition of a column, again: the latter is called vectorized processing.

Quizzes

1. Which of the following is NOT a method of the Operator interface for implementing pipelines explained in the video?

   (a) open()

   (b) next()

   (c) remove()

5.3.3 Operator Example Implementations

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[BBD+01]

Learning Goals and Content Summary

Selection Operator

```
class Selection implements Operator<Row>{
    private Operator<Row> input;                                  //internal handle to input Operator
    private Predicate<Row> sel;
    public Selection(Operator<Row> input, Predicate<Row> sel){    //constructor
        this.input = input; this.sel = sel;
    }
    public void open(){input.open();}                             //initializes the operator
    public Row next(){                                            //returns the next row of data
        For (Row tmp = input.next(); tmp != NULL; tmp = input.next()){
            If( sel.execute( tmp ) ){
                return tmp;
            }
        }
        return NULL;                                              //signal end of input
    }
    public void close(){input.close();}                           //performs cleanup work (if necessary)
}
```

Save the State as an Attribute

```
class Enumerate implements Operator<Integer>{
    private int current, from, to;
    public Enumerate(int from, int to){                           //return [from;to], i.e. both including
        this.from = from; this.to = to;
    }
    public void open(){                                           //initializes the operator
        current = from;                                           //initializes the state of the operator
    }
    public Integer next(){                                        //returns the next row of data
        If (current <= to){                                       //if still in range
            Integer nextToReturn = current;                       //this is what we return
            current++;                                            //need to increment internal state
            return nextToReturn;                                  //return next element
        }
        Else return NULL;                                         //signal end of input
    }
    public void close(){}                                         //performs cleanup work (if necessary)
}
```

Figure 5.14: Pseudo code for selection and enumeration operators

How do we implement specific operators without unnecessarily breaking the pipeline?

As a general rule we should implement operators such that they return subsets of their result as early as possible. Whether it is possible to return a specific result without running the operator until completion depends on the type of operation computed by the operator. For instance, if we want to compute the maximum value of an input of unsorted integers, it is impossible to return that maximum without consuming the entire input of the operator. Recall our discussion on pipeline breakers in Section 5.3.1.

projection **selection** **operator**

Is it hard to implement a projection or selection operator?

Not really. Let's assume row-wise iteration, i.e. we consider a chunk to represent a row. A selection operator may inspect each input tuple one by one whether it fulfills the selection predicate. If the selection predicate holds, the selection operator may immediately return that tuple. If the selection predicate does not hold, the selection operator continues consuming its input until it finds a qualifying tuple. Similarly, for a projection, we rely on a mapping function projecting one input tuple to an output tuple anyways. This mapping function may be applied for each tuple independently. Hence, a projection operator simply needs to consume one tuple from its input. Then it apples the mapping function and returns the result. Only if its parent operator requests the next tuple, the projection operator will continue processing.

loops

How do we handle loops in operator implementations?

If we want to return some result while being in the middle of a loop, we need to make sure that a subsequent call to `next` does not reinitialize the loop. This means, if we use a loop in a `next`-call, we need to be able to continue the loop at the position where we terminated it in the last `next`-call. This can easily be achieved by removing the variable initializations we are iterating on from the loop. Those variables become part of the state of the operator, i.e. attributes of the class we are implementing. For instance, in Figure 5.14, in class `Enumerate`, we introduce a private attribute `current`. This attribute stores the current position of the loop. Any call to `next` will access that attribute and change its state.

state

What do we mean by 'state' here?

State may be defined in different ways. In the context of operators we use state to denote any variables that are introduced and kept as part of an instance of an operator. This means, by state we mean additional state introduced to be able to continue processing later on for a specific operator. Examples of state that may be kept by an individual operator include: loop counters, temporary or buffered data, and intermediate results. We ignore in this discussion external state, i.e. state that is kept not solely for the purpose of this operator. Examples for external state include indexes that are kept by the database system and any temporary data that is buffered outside the scope of the operator.

Quizzes

1. What is the minimal set of methods each operator has to implement based on the Operator interface?
   (a) open(), next(), close()
   (b) open(), hasNext(), next(), close()
   (c) constructor, open(), next(), close()
2. Which of the following relational algebra operators can be implemented as stateless Operators on rows:
   (a) projection
   (b) selection
   (c) aggregation with grouping
   (d) natural joins

5.3.4 Query Compilation

Material

Literature:
[Neu11] (Sections 1 to 4)

Additional Material

Code Example:
W Java Code Example for Compiling Predicates

Code Example Slides:
W slides

Literature:
W Visitor Pattern
[Vig13]

Learning Goals and Content Summary

What is the performance problem with operators?

Operators may incur a lot of overhead in terms of function calls for `next`-calls, i.e. in the extreme case there is one call for each row at each operator in the execution plan. These costs may not be a big problem in a disk-based system where the I/O-costs typically overshadow the function-call overheads. In a main-memory system however, the costs for these function calls may be substantial and become a bottleneck in query processing.

compiling code

How is the pipeline organized logically when compiling code?

The query plan is divided into blocks where each block is run as one pipeline until the next natural pipeline breaker. A natural pipeline breaker is, for instance, a hash table materializing all of its inputs. Then each block is translated into program code (either using C/C++, LLVM or any suitable combination).

LLVM

C++

How to translate entire plans into LLVM/C++ code?

An entire plan ins translated using a mix of C++ and LLVM. The idea is to keep complex code, e.g. precompiled index structures or operators, written in C++, however code actually accessing data is kept in LLVM. The major design rationale is to write all 'hot' code, i.e. code that is executed many times in LLVM. Switching here and there to C++ does not hurt much as long as long as the heavy lifting is done in LLVM.

What is the biggest advantage of using LLVM?

The biggest advantage of using LLVM is orders of magnitude faster code compilation times compared to C++-compilers. This allows for just-in-time-compilation, i.e. when the query is issued to the database systems it may be compiled on the fly and executed directly. This typically does not pay off with C++ as the compilation times cancel out the benefits of compiled code.

Quizzes

1. Is there any disadvantage of using the Operator or Iterator interface (implemented) to execute query plans?
   (a) No, there aren't any.
   (b) Yes, they usually incur a virtual (function) call, or a call via a function pointer, which is more expensive than a regular function call.
   (c) Yes, they can only be applied when the Chunk parameter is implemented as pages of memory.
2. What is the main idea presented in the paper for executing queries in a DBMS?
   (a) The query plan is rewritten into a more efficient algebraic representation.
   (b) The query is directly translated into (a compact and efficient) machine code - thus considering any given query as a small program that must be executed by the DBMS.
3. The method described in the paper aims mainly at (on a high level)?
   (a) Maximizing data- and code locality.
   (b) Producing codes without branches.
   (c) Vectorizing all loops.
4. What is the goal of the method presented in the paper?
   (a) To make query execution operator-centric
   (b) To make query execution data-centric
5. At which point in query planning does the method explained in the paper differ from the traditional (Operator-, Iterator-based) scheme?
   (a) After the initial algebraic expression is created, the method explained in the paper generates an optimized version, while the traditional scheme does not.

(b) After the initial algebraic expression is created and optimized, the method explained in the paper compiles this optimized algebraic expression into machine code.

(c) After the physical query plan is created, some joins are reordered, then the plan is translated into machine code.

6. What tool is used to generate machine code for a given query?

   (a) Java Virtual Machine

   (b) C++ compiler

   (c) Low Level Virtual Machine (LLVM)

7. Could the method presented in the paper be parallelized, say by SIMD instructions and/or multi-core parallelization?

   (a) Yes, data is consumed in chunks, which allows for parallelization.

   (b) No, data flow is highly dependent on execution flow, which make parallelization impossible.

8. From the experiments shown by the author comparing different systems, what can we conclude?

   (a) The new LLVM-based code is not better than existing solutions.

   (b) The new LLVM-based code is not more efficient, but compiling such optimized code is way faster than the existing solutions.

   (c) The new LLVM-based code is indeed more efficient, both in execution- and compilation time, although the latter is in general the main advantage of the method presented.

5.3.5 Anti-Projection, Tuple Reconstruction, Early and Late Materialization

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Further Reading:
[MBNK04]
[AMDM07]
[IKM09]

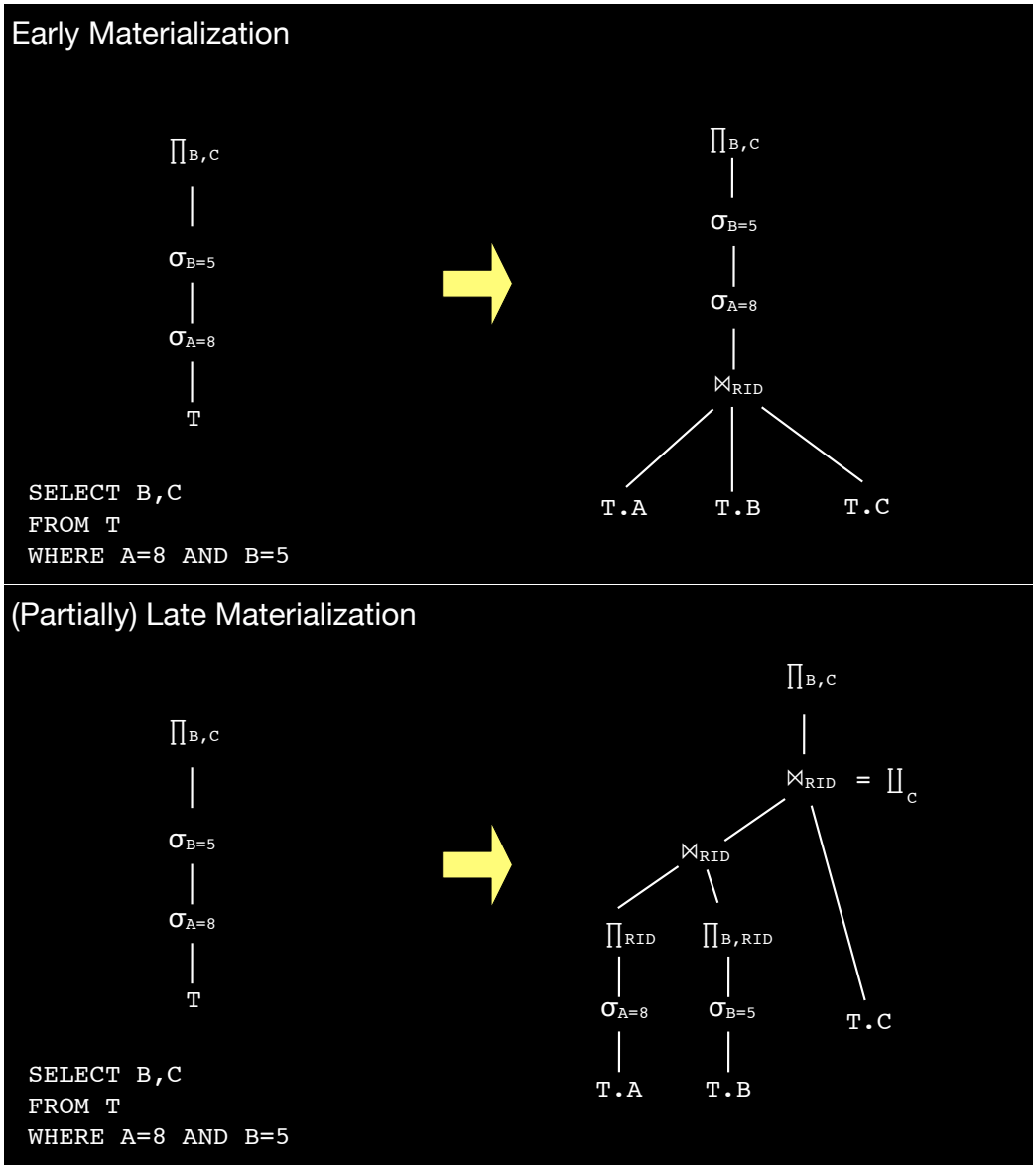


Figure 5.15: Early vs (partially) late materialization

Learning Goals and Content Summary

early materialization

What is early materialization?

In early materialization we read all attributes that are required to process a query as early as possible. This means for any plan requiring say attributes A, B, and C of table T, we replace T by the join over columns A, B, and C, as join key we use the RID. In other words, we read all three columns A, B, and C and keep them in a joint representation for further processing, typically a row layout, in this case a row of three attributes. See Figure 5.15 for an example.

column

What is the relationship of early materialization to column and vertically partitioned (column grouped) layouts?

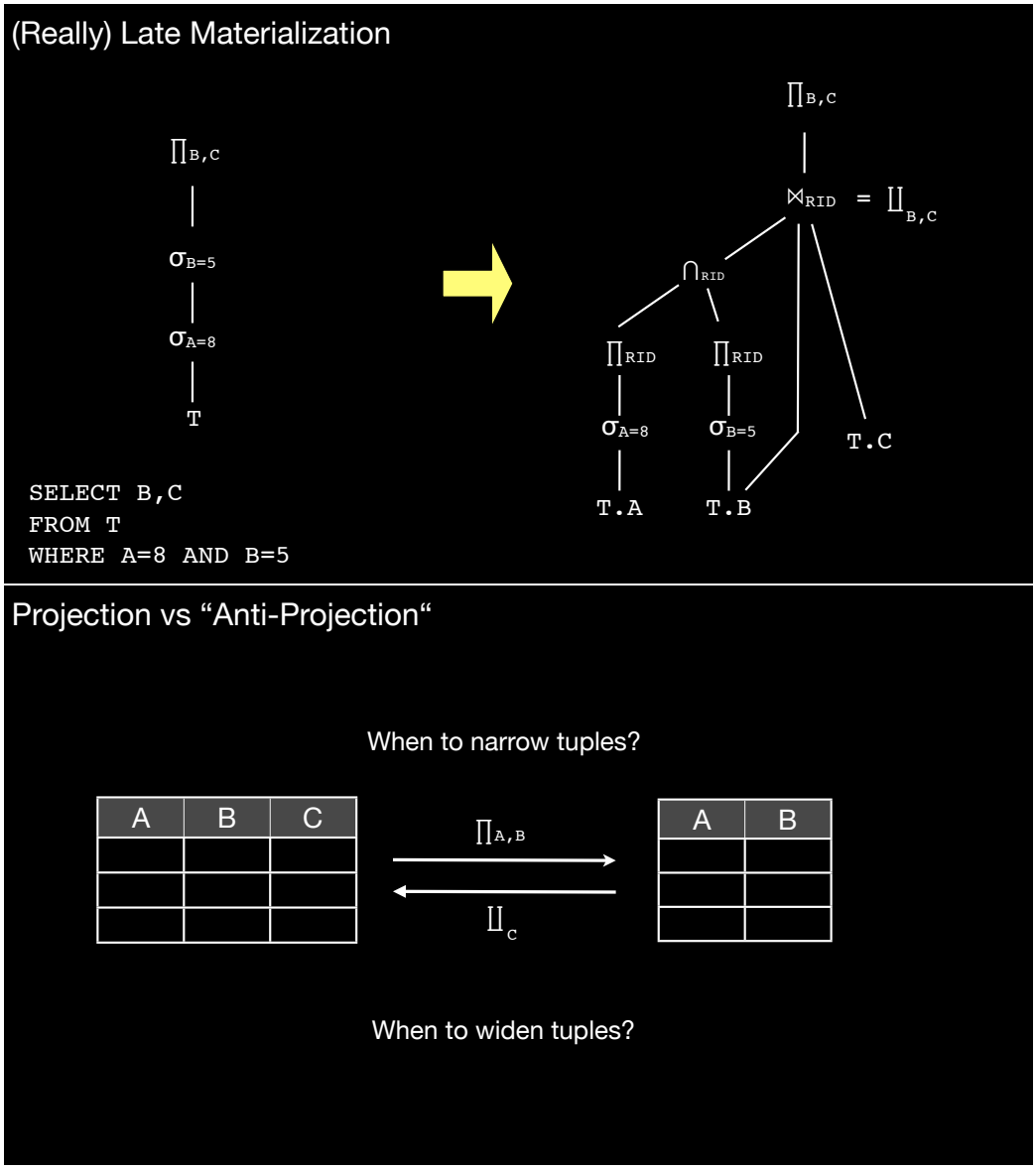


Figure 5.16: Late materialization and the relationship of projection and anti-projection

In column grouping, suitable columns are grouped *before* the query is executed. In contrast in early materialization, we group columns as the first step of query processing *at query time*. Another way of seeing this is that in column grouping the RID join over the different columns is already materialized in the store. You could consider the result of the RID join a materialized view already available in the store. Notice that as early materialization is not the best processing strategy in a column store, this relationship also gives you an intuition that vertically partitioning data in order to avoid the RID join is not necessarily a good strategy.

What is late materialization?

In late materialization all tuples are read as late as possible. For instance, in Figure 5.16

this is pushed to the extreme: the RID join over columns A and B does not even keep the values of column B. Only afterwards, in the RID join with C column B is read again to retrieve the necessary values. Notice that for this particular query B=5 for all tuples in the result set anyways. Hence, let's hope for the best that the query optimizer will not read column B again anyways. However, this does not change the principle underlying problem: consider that we change the condition on column B to B>5. In that case, the query optimizer has to make a decision on when to fetch the attributes of B.

What is 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 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.

query planning

What is the impact on query planning?

As any anti-projection can be seen as a join of a relation with the attribute to anti-project, we could phrase tuple reconstruction as a join order problem. Recall Section 5.1.3 where we considered different orders of joining relations. In that section we still assumed that the inputs to the joins at the leaf-level of the join-tree are entire relations. However, if we assume that the input to a join is just one column of a relation, we could phrase anti-projections as a join order problem. All techniques that are suitable to the standard join order problem then also apply to tuple reconstruction. As the number of attributes of a schema is larger or equal than the number of its relations, enumerating all of these

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

options becomes quickly infeasible. See also the discussion in [SS16].

What is a join index?

join index

A join index materialized the pairs of tuples that belong to the join result. For instance for a join of two relations R and S, the join index stores a table with two columns R.RID and S.RID where each row in that table marks a result belonging to the join of R and S. Notice that in contrast to a materialized view, which may materialize an arbitrary query (including a join), a join index materializes the join keys and/or RIDs only.

Quizzes

1. Assume your are the developer of a DBMS, and assume you want to write code to perform the following query: SELECT A, B FROM C WHERE a1 < A < a2 AND b1 < B < b2. Does the exact same query processing algorithms perform equally well regardless of whether the DBMS is row-oriented or column-oriented?
   (a) No
   (b) Yes
2. Assume your are the developer of a column-oriented DBMS, and assume you want to write code to perform the following query: SELECT A, B FROM C WHERE a1 < A < a2 AND b1 < B < b2. Moreover, assume you already have the code for answering this query in a row-oriented DBMS. What paradigm helps you to reuse the existing code as much as possible?
   (a) Late materialization
   (b) Early materialization
3. Assume you perform the following query: SELECT D FROM T WHERE a1 <= A <= a2 AND b1 <= B <= b2 on a column-oriented DBMS. Which of the following paradigms project initially to row ID?
   (a) Early materialization
   (b) Late materialization
4. Could both early and late materialization be used in join processing as well, or only in selections?
   (a) Yes, they both can be used in join processing as well.
   (b) No, only late materialization can be extended to join processing.
   (c) No, only early materialization can be extended to join processing.
5. Which of the following is NOT an advantage of late materialization?
   (a) Lazy construction of tuples (only when necessary)
   (b) No re-accessing of the data
   (c) Potential minimization of intermediate results

6. Which of the following makes tuples wider?
   (a) Projection
   (b) Anti-projection
7. When performing an equi-join using late materialization, what does get initially projected in the join index?
   (a) Column data
   (b) Row IDs

Exercise

Assume you have a main-memory column-store database with implicit keys, and table `T` containing $1,000,000,000$ tuples.

(a) Provide two query plans: one using late-materialization and one using early-materialization for each of the following queries:

Q1:

```
SELECT a
FROM   T
WHERE  b < 4 AND d > 42;
```

Q2:

```
SELECT   a, AVG(b)
FROM     T
GROUP BY a
HAVING   AVG(b) > 100;
```

Q3:

```
SELECT   a, b, MIN(c)
FROM     T
WHERE    b > d
GROUP BY a, b
HAVING   MIN(d) > 100;
```

(b) Assume that the selectivities of the filter conditions in Q1 and Q3 are as follows:

$$sel(b < 4) = 0.02, \quad sel(d > 42) = 0.2, \quad sel(b > d) = 0.125,$$

and that these are independent from each other. Further, you have a column-scan operator which takes a bitlist of row-IDs as a parameter, and returns only the attributes stored at those positions. Our simplistic cost function considers only the

number of attribute values read from each column. Provide a cost-estimate for each of the query-plans you have created in (a).

(c) There is more than one possible plan using late-materialisation for Q1. List all possible plans and provide the cost-estimates for them using the previously introduced cost function.

Chapter 6

Recovery

6.1 Core Concepts

6.1.1 Crash Recovery, Error Scenarios, Recovery in any Software, Impact on ACID

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
[LÖ09], Logging and Recovery
[LÖ09], Logging/Recovery Subsystem
[LÖ09], Disaster Recovery
[LÖ09], Crash Recovery

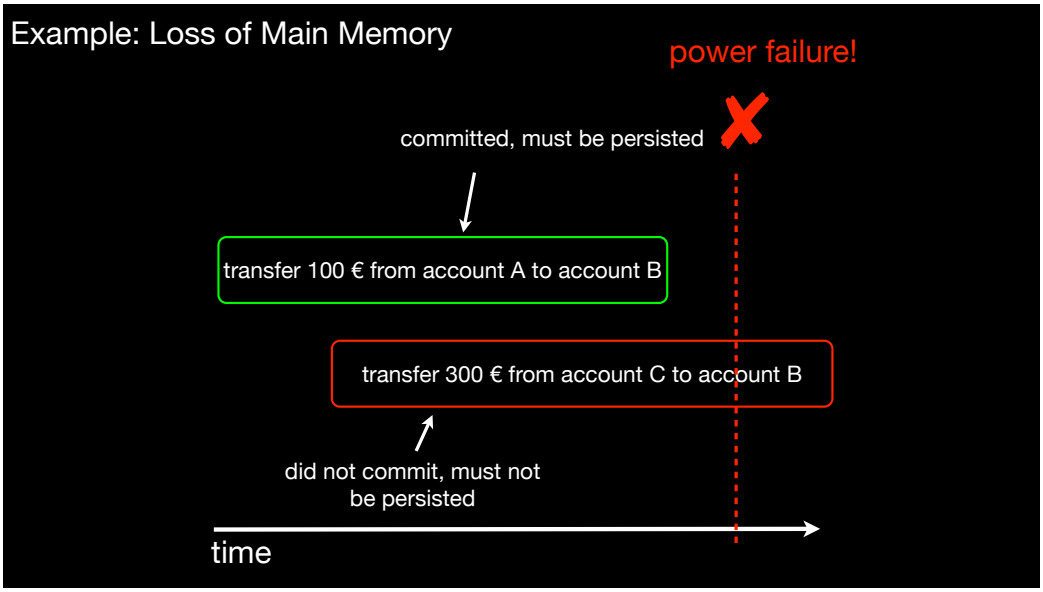


Figure 6.1: Possible effects when loosing main memory due to a power failure

Learning Goals and Content Summary

error scenario

What are possible error scenarios?

Possible errors scenarios include: power failures, hardware failures (also due to outside occurrences like fires, floods), burglary, and software errors.

power failure

database buffer

What could be the impact of a power failure on the database buffer?

In the worst case, if you only lose the contents of the database buffer, all dirty pages are lost, i.e. all changes that were made in the buffer but were not persisted on disk (or any other persistent medium) yet.

recovery

What does the term 'recovery' mean?

By recovery we mean that we can get back to a consistent state of the database according to the ACID properties. Recall again, that all transactions that were running but not committed at the time of the crash have to be replayed in their entirety.

Is recovery just important for database systems?

No, recovery is important in many different situations in computer science. It makes software robust against failures. Recovery is strongly related to the undo-functionality you find in many software products, to versioning, to backups and archiving, to journaled file systems (which typically can recover the metadata like the file system's structure but not the file contents), and also embedded systems. In fact, any message like "do not switch off, system is performing XYZ" can be interpreted as a sign that recovery was not implemented.

ACID

What is the impact of error scenarios on ACID?

Error scenarios typically impact atomicity, i.e. only some of the actions of a transcation were executed, and durability, i.e. only some of the actions of a transaction were made

durable. These problems then also impact the consistency of the database.

What is a local error? **local error**

This is an error or ABORT in an application program, i.e. a program using the database through some interface (like JDBC). In addition, the DBMS may trigger a local error (a reset of one transaction) to resolve a deadlock situation in concurrency control.

What is redo? **redo**

Redo, in general, means that we have to reapply actions to the database state as those actions (for whatever reason) were not reflected in the database state.

What is undo? **undo**

Undo, in general, means that we have to remove effects of actions done to the database state as those actions were reflected in the database state, however, have to be removed as they belong to uncommitted (loser) transactions.

What does losing main memory mean?

Losing main memory implies that we lose all dirty pages (clean pages, by definition, are all persisted on disk anyway). This means, all changes applied to those dirty pages are lost.

What if we lose (some) external storage?

This depends on how much of external storage we lose. If we "just" lose the database, but still have the log file, we can reconstruct the database using that log. Recall, that the database is just an optimized materialized view of the (unpruned) log. See also Section 1.3.9. If we also lose the log file, then we may be in trouble. For these situations we can hopefully rely on a backup copy of the database and/or the log file. The particular setup, how the database and/or the log file is archived and how often should be considered with care.

What is the central idea of recovery?

The central idea of recovery is always the same: get back to some consistent (or inconsistent) old state of the database and then perform recovery to bring the database to a more recent *but definitely* consistent state.

Quizzes

1. Which of the ACID properties can potentially be violated if there is a power failure on an operating DBMS?
   (a) A
   (b) C
   (c) I
   (d) D
2. How could a software keep backup copies?

(a) There is the master (editable) file, and a single copy of this (master) file is made and kept updated concurrently for each and every change as the master file changes.

(b) There is the master (editable) file, and different snapshots of this (master) file are created and kept as the master file changes.

6.1.2 Log-Based Recovery, Stable Storage, Write-Ahead Logging (WAL)

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
see 6.1.1 and 6.2

Learning Goals and Content Summary

stable storage

What is 'stable storage'?

Stable storage refers to a special (typically additional device) that is used to persist the log file. "stable" is a relative term. Typically, we will argue in the following that a transaction can be considered committed once all log records created by that transaction are persisted on stable storage. This means, our notion of whether a database is considered committed highly depends on the definition of "stable". If we lose stable storage, we will lose transactions, previously considered committed.

database store

What is its relationship between stable storage and the database store?

You may consider stable storage to be part of the store, but may also consider it to be an outside, special, storage functionality. We stick with the latter notion in the following.

logging

What is the relationship between stable storage and logging?

It is a refinement of logging where the log file is assumed to be kept on an extra device.

write-ahead logging

WAL

What is write-ahead logging (WAL)?

Write-ahead logging introduces two constraints on the order of persisting data in the store and the log. First, before persisting any change done to the store, all log records reflecting that state must be persisted, i.e. the log is *written-ahead* in that sense that the log (on disk) is in a newer version than the database state (on disk). Second, a transaction may only be considered committed *after* all log records created by that transaction are persisted on stable storage.

What is and what is not allowed following WAL?

It is not allowed that any page in the database state is persisted without first persisting the corresponding log records. In other words: database pages *must not* be persisted

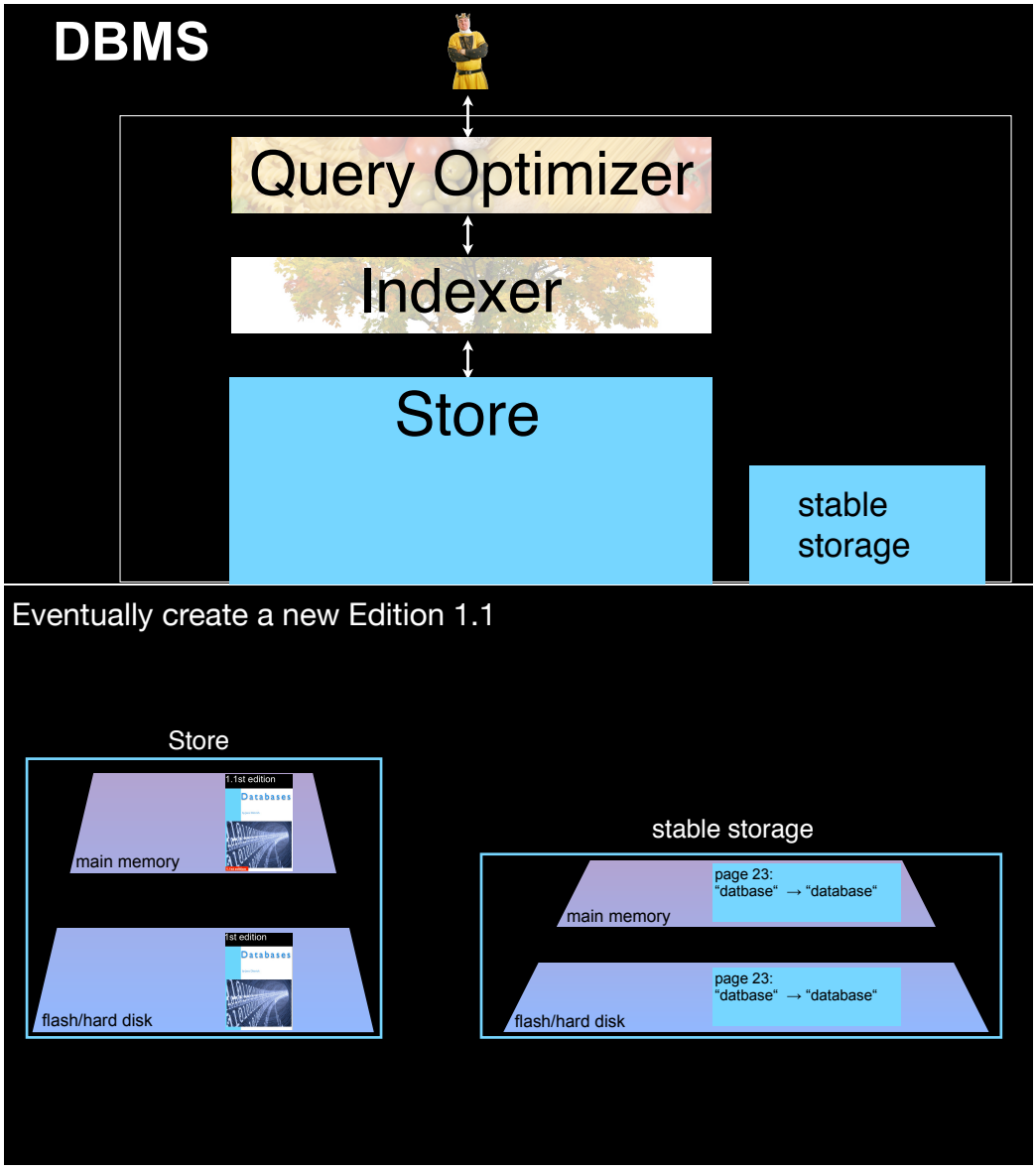


Figure 6.2: The relationship of the 'normal' database store and stable storage in terms of architecture and durability of updates

in newer versions than the corresponding log records. In addition, the database system is not allowed to tell an application program that a particular transaction committed successfully before all log records created by that transaction are persisted *in the log* — whatever the state of the database then is: it does not matter for this rule.

What does WAL imply for the database buffer and page eviction of dirty pages?

database buffer

page eviction

dirty page

Before writing a dirty page to the database store (be it due to page eviction or a background thread regularly writing back dirty pages), we have to persist all log records reflecting changes done to that page in the log.

Quizzes

1. What is the main difference between the store and the stable storage on a DBMS?
   (a) The stable storage works as a secondary store in case the main store runs out of space.
   (b) The stable storage is only considered for redundantly logging changes occurring in the DBMS, as to facilitate recovering from a possible failure.
2. What does write-ahead logging (WAL) in a disk-based database system mean? It means...
   (a) that every single change operation performed by a transaction is first flushed to the disk used by the log file, and then the change operation is flushed to the persistent disk used by the database store.
   (b) that every single operation performed by a transaction is first flushed to the disk used by the log file, and then the operation is flushed to the persistent disk used by the database store.
   (c) that every single change operation performed by a transaction is first flushed to the disk used by the log file, and only afterwards the change operation is applied on the database store (including the non-volatile DB buffer in main memory and hard disks).
3. Assume you have a DBMS with multiple storage layers, for the store as well as for the stable storage. Let these storage layers be main memory (RAM) and disk (HDD) in both, the store and the stable storage. Which of the following is NOT WAL?
   (a) First log the transactions on the two storage layers of stable storage (from RAM to HDD), and then perform the transaction on the DBMS and write out to the store (from RAM to HDD).
   (b) First log the transactions on the RAM of the stable storage, then perform the transaction on the DBMS and write out to the store (from RAM to HDD), and finally log the transactions on the HDD of the stable storage.
   (c) First log the transactions on the RAM of the stable storage, then perform the transaction on the DBMS (RAM), then log on the HD of the stable storage, and finally write out to the store (HDD) of the DBMS.

Exercise

Assume a disk-based DBMS **not using** WAL. The transaction execution example we look at is performing two interleaved transactions, transferring 50 Euro from account A to account B by transaction 1, while transaction 2 transfers 10 Euro from account A to account C. You can assume that dirty pages are only flushed to disk if explicitly stated in the following action log:

(1) T_1: Read the balance of account A into variable bal_{a1}

(2) T_1: $bal_{a1} := bal_{a1} - 50$

(3) T_2: Read the balance of account A into variable bal_{a2}

(4) T_2: $bal_{a2} := bal_{a2} - 10$

(5) T_2: Set the balance of account A to bal_{a2}

(6) T_1: Set the balance of account A to bal_{a1}
[Flush buffer pages to disk]

(7) T_1: Read the balance of account B into variable bal_{b1}

(8) T_1: $bal_{b1} := bal_{b1} + 50$

(9) T_1: Set the balance of account B to bal_{b1}

(10) T_1 commit

(11) T_2: Read the balance of account C into variable bal_{c2}

(12) T_2: $bal_{c2} := bal_{c2} + 10$

(13) T_2: Set the balance of account C to bal_{c2}
[Flush buffer pages to disk]

(14) T_2 commit

Assume one of our consistency constraints is that the sum of the amounts of money from all accounts should be equal at all times.

Which of the ACID properties are violated in the above transaction execution example? Refer to the exact lines causing a violation in your justification. If the system crashes directly after the execution of command (12), will the committed transaction be durable?

6.1.3 What to log, Physical, Logical, and Physiological Logging, Trade-Offs, Main Memory versus Disk-based Systems

Material

Video:	Original Slides:	Inverted Slides:

Additional Material

Literature:
see 6.1.1 and 6.2

Learning Goals and Content Summary

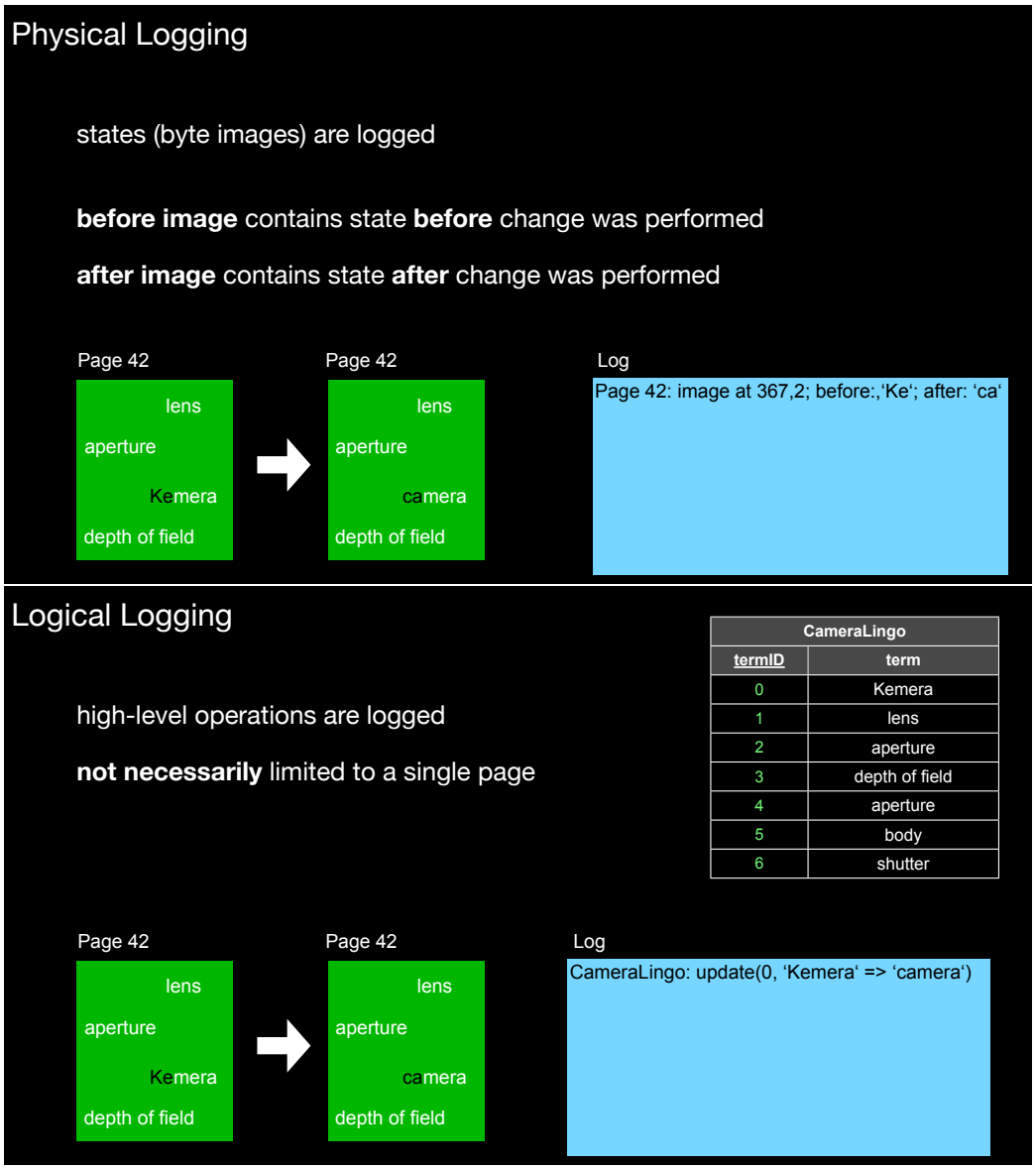


Figure 6.3: Physical vs logical logging

physical logging

What is physical logging?

In physical logging a snapshot of a subset of the database state is logged.

after image

What is an after image and a before image?

before image

The *before image* logs the state as of before applying a change. The *after image* logs the state as of after applying a change. Obviously, this is only efficient if the amount of data recorded in the before and after images is relatively small.

How big may before and after images become?

Well, in theory, if everything was changed by an operation, physical logging has to log

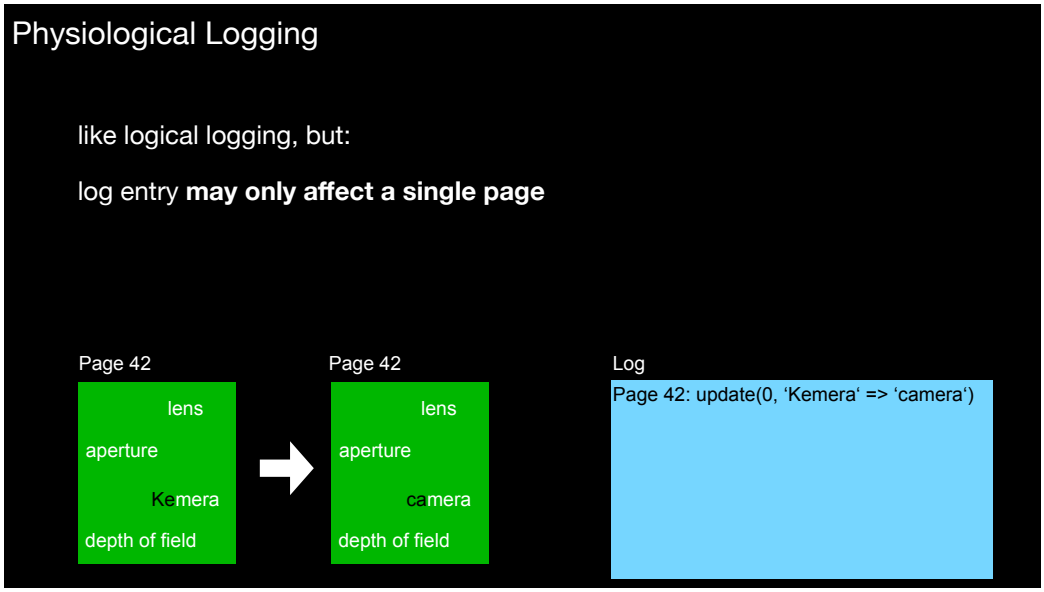


Figure 6.4: Physiological logging

the entire database.

What is logical logging?

logical logging

Logical logging logs the transitions between database states, i.e. we log the operation that is required to get from an older database state to a newer database state and vice versa. As transitions we use different granules. One extreme version is to simply log the SQL-command (in some byte-efficient representation). Another extreme is to simply log transitions operating on atomic data values, e.g. a=a+10.

How does logical logging differ from physiological logging?

physiological logging

Logical logging is not restricted to a particular page. This means, the transition recorded in a logical log record may effect multiple pages. This is not the case for physiological logging: even though these log records describe a transition, that transition may only effect one particular page.

How do log entries for logical, physical, and physiological logging typically look like?

Physical log records contain an after and a before image and (typically) a reference to a storage location where these images should be applied (on the level of pages). Like physical log records, physiological log records keep a reference to a storage location (typically a page), yet they do not store images but transitions. Logical log records do not keep a reference to a single page and keep transitions.

What are the performance trade-offs for the different logging variants in general?

We can identify two extremes: logical logging at the level of transactions (or their IDs) and physical logging using before and after images. An important consideration when choosing a particular method is the processing time triggered by a particular log record during recovery. For instance, a logical log record just storing the ID of a complex query

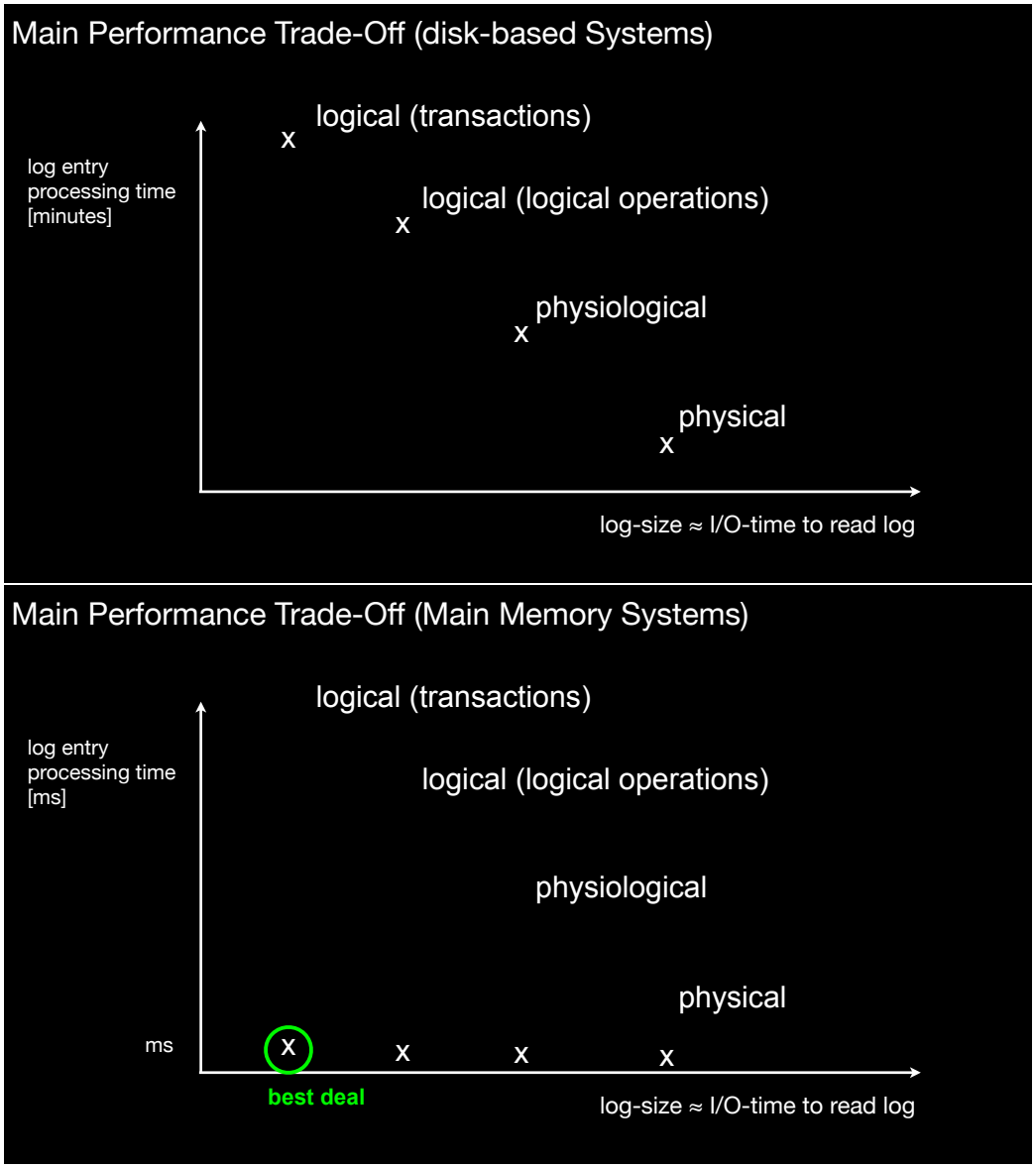


Figure 6.5: Logging trade-offs in different database systems: disks-based vs main-memory systems

may affect several pages, it may involve heavy join computations and so on and thus in total it may take a long time to be redone. In contrast, replaying a physical log record merely involves loading one particular page and copying the after image to that page. So there is a clear trade-off among the space required for the log records and the time it takes to process those log records.

What are the performance trade-offs for the different logging variants in a disk-based system?

In particular in a disk-based system the processing time for a particular log record may be huge. Hence, the I/O-time to read the log is not the bottleneck, but rather the time to apply the log records. It makes sense to store log records on a more physical level. If

logical log records are used, it may make sense to consider these log records "transactions" and to translate them into physical log records (as done in the full-blown ARIES algorithm 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?

Storing queryIDs rather than queries is an application of dictionary compression.

Why do the trade-offs differ so much in the two types of systems?

Again, due to the very different timescales used to change data in a disk-based store versus a main-memory store.

Quizzes

1. What is physical logging?
   (a) At a page level, explicit before and after changes are recorded.
   (b) Changes are kept at a high level, say at a table level, where the information of what entries are changed is kept. Changes here are not necessarily bounded to be kept at a page level.
   (c) We record high level information but restricted to a particular page.
2. What is logical logging?
   (a) At a page level, explicit before and after changes are recorded.
   (b) Changes are kept at a high level, say at a table level, where the information of what entries are changed is kept. Changes here are not necessarily bounded to be kept at a page level.
   (c) We record high level information but strongly restricted to a particular page.
3. What is physiological logging?
   (a) At a page level, explicit before and after changes are recorded.
   (b) Changes are kept at a high level, say at a table level, where the information of what entries are changed is kept. Changes here are not necessarily bounded to be kept at a page level.
   (c) We record high level information but strongly restricted to a particular page.

4. Is there any disadvantage in logical logging?
   (a) No.
   (b) Yes, the size of the log increases significantly if there are many changes.
   (c) Yes, changes touching multiple pages must be performed in an atomic manner (all or none) when recovering from a failure, they (possibly) could not be performed one by one.
5. What logging technique is in general more space-efficient?
   (a) Physical logging.
   (b) Logical logging.
   (c) Physiological logging.

6.2 ARIES

Material

Video:	Original Slides:	Inverted Slides:

Literature:
[Fra97] (Section 3.2 only)

Additional Material

Literature:
[MHL+92]

Further Reading:
[Moh99]
[LÖ09], Multi-Level Recovery and the ARIES Algorithm

Learning Goals and Content Summary

ARIES

What are the three phases of ARIES?

1. Analysis, 2. Redo, and 3. Undo

What are the core tasks of the three phases of ARIES?

The three phases can technically be summarized as follows:

1. Analysis Phase: construct metadata on dirty pages and non-committed transactions (DPT and TT) as of the time of the crash; compute where to start the Redo phase (the minDirtyPageLSN)
2. Redo Phase: repeat all actions including those from loser transactions and changes undone previously (the CLRs); restore database to where it was at the time of the crash (repeating history)
3. Undo Phase: undo actions of all loser transactions; log those undos to special log records (called CLRs)

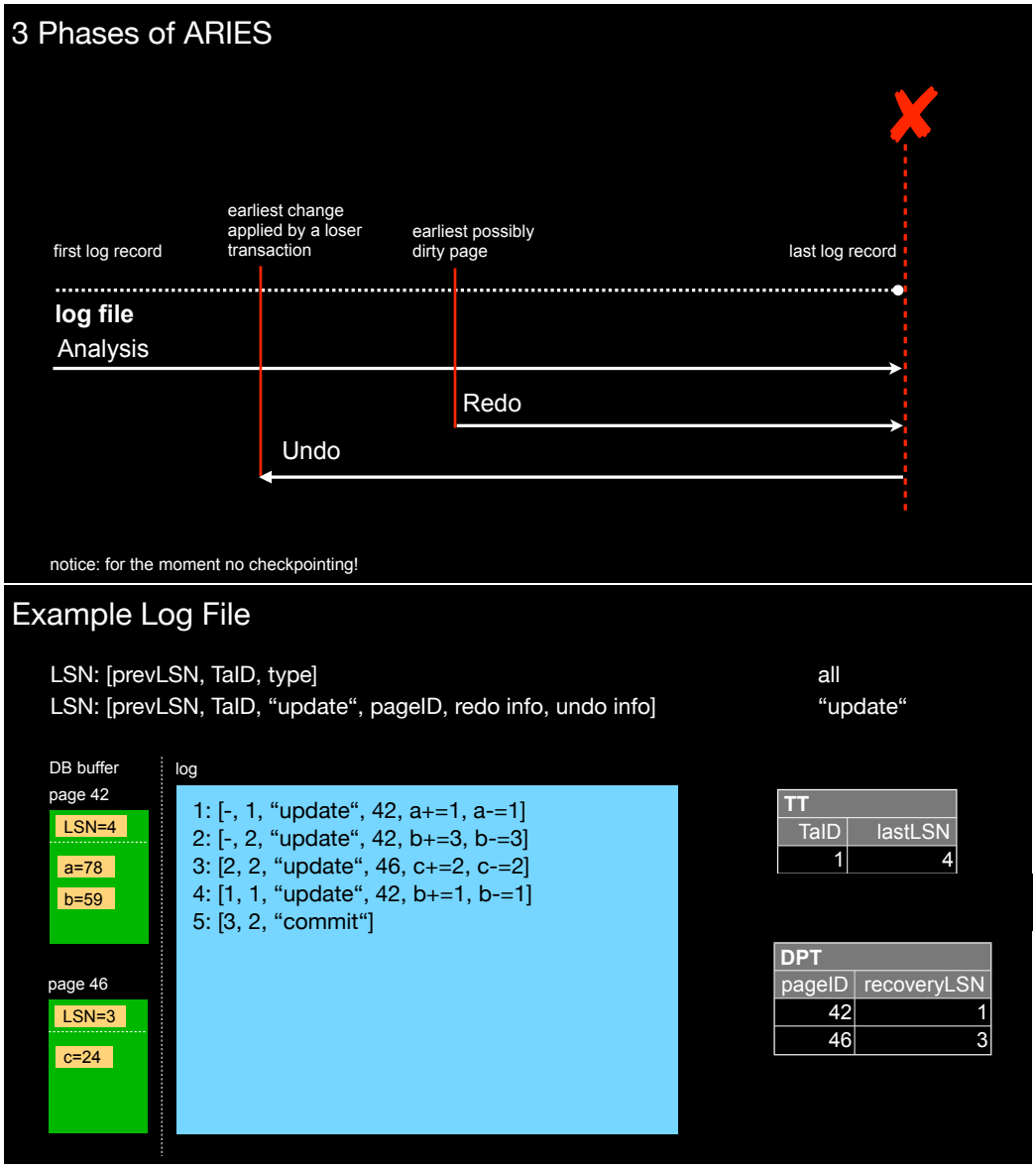


Figure 6.6: The three phases of ARIES and an example logging scenario showing the DB buffer, the log, the transaction table (TT) and the dirty page table (DPT)

We will explain the different concepts in the following.

What is an LSN?

LSN

In order to keep track of log records during recovery, we need a way of uniquely identifying log records. This is done by introducing log sequence numbers (LSNs). Log sequence numbers do not need to be stored explicitly in the log. The easiest implementation of LSNs is to assume that in the log file for each log record, its offset corresponds to its LSN.

log sequence number

What is the purpose of the transaction table (TT)?

transaction table

TT

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

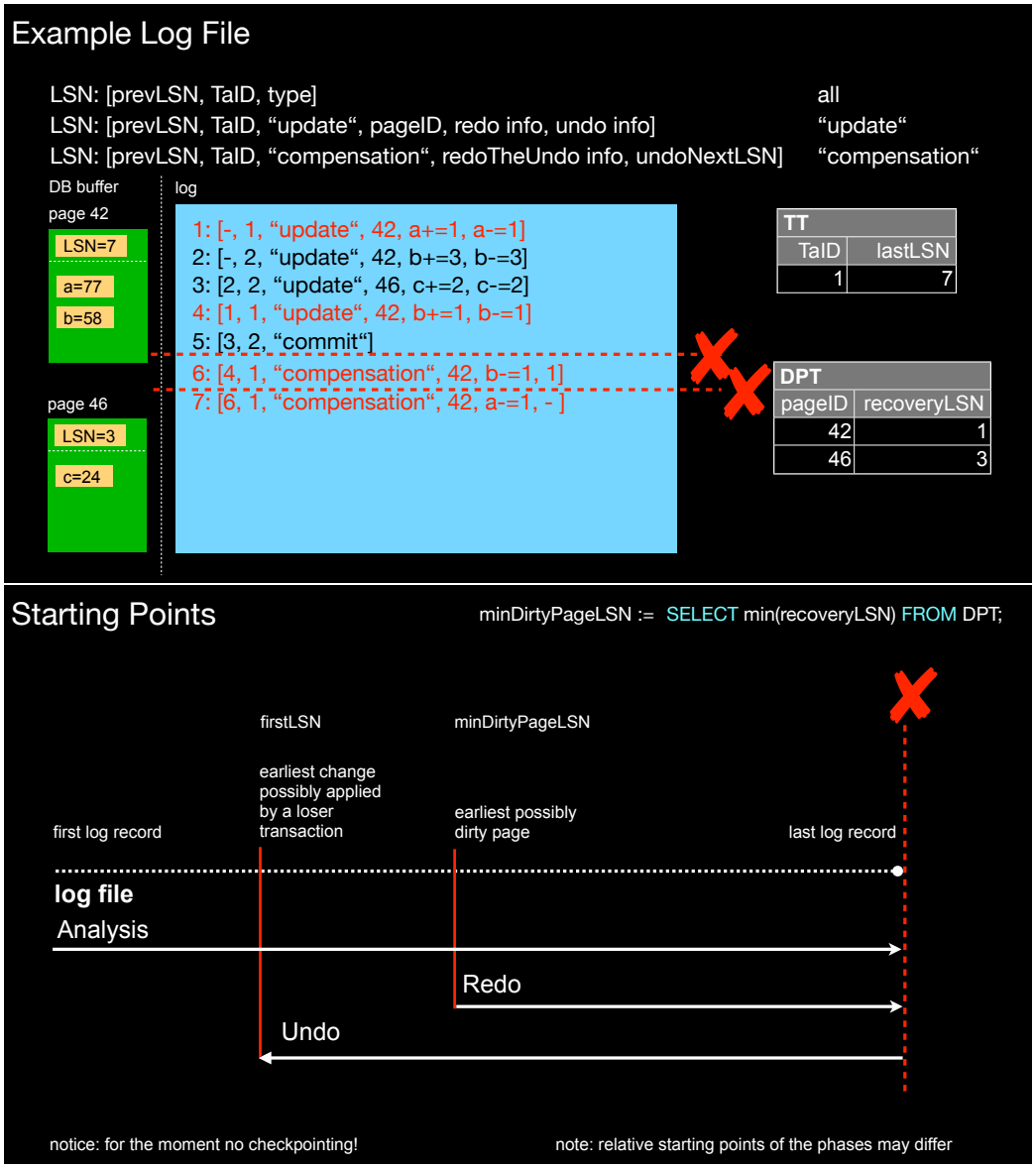


Figure 6.7: The logging scenario under multiple crashes and a schematic view on the different possible starting points for the three different phases in ARIES

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

dirty page table

DPT

What is the purpose of the 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

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.

What information is contained in an update log record?

update log record

The update log records in ARIES have a particular structure. The redo information is physical or physiological, however the undo-information may be logical. For simplicity, we constrain undo information to be physical or physiological as well. This implies that there is a field pageID recording the page that is affected by this update log record. If a change affects multiple pages there must be an update record for each of those pages. In addition, an update log record contains a field TaID, that is the ID of the transaction that performed this change. Moreover, there is a field prevLSN, that is the LSN of the previous action performed by transaction TaID. If this action is the first action of transaction TaID, prevLSN is set to empty (or NULL). For details on full logical undo see the article on multi-level recovery [REF above].

pageID

TaID

prevLSN

What is the general idea of a compensation log record (CLR)?

compensation log record

CLR

A compensation log record is written in the undo-phase of ARIES. A CLR reflects a change done to the database store while removing a database store modification performed by a loser transaction earlier. In other words, the CLR, just like a normal update log record, records a database store modification. In contrast to a normal update log record, that modification was not triggered by a transaction, but by recovery itself. Other than that, a CLR is treated just like any other update log record: in case of another crash it will be considered in the redo-phase just like any other update log record; that CLR must be ignored in any further undo-phases.

What is the redoTheUndo field contained in a CLR?

redoTheUndo

The redoTheUndo fields records the undo information applied by recovery during the undo-phase, i.e. semantically this field reflects undo information, yet it will be replayed in the redo phase. and *not* in the undo phase.

Where does undoNextLSN point to?

undoNextLSN

This attribute points to the LSN of the next log record to be undone by this transaction.

How do we shorten (prune) a log file?

prune

We can shorten a log file by introducing checkpoints. How exactly depends on relative order of the start points of the redo and undo phases. Be careful that this pruning may

change the value of LSNs if there are implemented as offsets in the log file.

How do we shorten the analysis phase?

We shorten the analysis phase by regularly writing fuzzy checkpoint. Then we do not have to read the entire log during analysis, but rather start from a well-defined position in the log, i.e. the checkpoint, and only read from that checkpoint until the end of the log.

fuzzy checkpoint

What is a fuzzy checkpoint?

A fuzzy checkpoint stores the contents of TT and DPT in the log file.

How exactly is a fuzzy checkpoint written to the log file?

We first write a begin-checkpoint record. It marks the version of this fuzzy checkpoint. Then we write an end-checkpoint. The end-checkpoint contains a copy of TT and DPT as of the time of the begin-checkpoint. Between the begin-checkpoint and the end-checkpoint records ongoing transactions may write other log records.

How do we shorten the redo phase?

We can shorten the redo phase by running a background thread regularly writing out dirty pages to disk. This thread is run in normal operation of the database, i.e. not only during or after recovery. That thread will write out dirty pages regularly. As the redo phase determines its starting point by computing the minimum of all recoveryLSNs in the DPT, this has a direct effect: that minimum is pushed forward to a later point in time. In other words, the background thread avoids that the version of pages on disk and in the DB-buffer do not differ too much (in the sense that only very recent log records may need to be replayed on those pages).

How do we shorten the undo phase?

One source for a long undo phase is long running loser transactions, i.e. a transaction started a long time ago but did not commit before the crash happened. The undo-phase has to go back until the earliest change performed by any loser transaction, i.e. the earliest LSN written by any loser transaction. Therefore, anything that allows us to push the starting point of the earliest loser transaction forward in time, i.e. to a more recent LSN, helps shortening the undo-phase. One reason why a transaction may be long running could be that a transaction got stalled by a user-interaction. For instance, consider that a transaction waits for a user to enter some input on a screen. Only after that the transaction will continue processing. These kinds of transactions may be considered uncommitted by the database system over a long period of time. One way to fix this is to write transactions such that they are only executed after all user input has been collected (if that is possible depends on the application). Another possibility is to enforce a timeout for transactions (again, how exactly depends on the application).

firstLSN

How do we determine firstLSN and where do we cut-off the log?

It is determined implicitly while processing the log backwards during the undo-phase. A better, and more reliable way would be to extend TT to record an extra field firstLSN for each transaction. Then, at all times, we are able to compute the minimum firstLSN of TT.

Then, the minimum over the firstLSN column, mindirtyPageLSN, and begin-checkpoint of the last successful log record determine the cut-off point where to safely prune the log file.

mindirtyPageLSN

What does 'repeating history' mean in this context?

repeating history

This means that during the redo phase all changes are replayed on the databases store — including the ones done by loser transactions.

Where do the different phases start and where do they end?

Figure 6.7 gives a principle overview on the starting points. Notice that the relative order of the starting points may be different.

1. Analysis Phase: starts at the beginning of the log file or (if it exists) the youngest fuzzy checkpoint successfully written (in which case it starts at the begin record of that checkpoint); ends at the last log record
2. Redo Phase: starts at the earliest possibly dirty page as determined by minDirtyPageLSN; ends at the last log record
3. Undo Phase: starts at the youngest log record of any loser transaction; ends at oldest log record possibly applied by a loser transaction (firstLSN)

Notice that the costs for the analysis phase may be shortened considerably by regularly writing fuzzy checkpoints. In addition, the costs of the redo phase may be shortened considerably by regularly running a background thread writing out dirty pages.

Quizzes

1. Which of the following is true?
   (a) ARIES uses logical redo and physical undo
   (b) ARIES uses logical redo and physiological undo
   (c) ARIES uses physiological redo and logical undo
   (d) ARIES uses physical redo and physical undo
2. Consider the following scenario: transaction A performs certain updates on page P. Transaction B kicks in before A finishes and also performs updates on page P, but in such a way that P is broken into more pages. Then B commits. Then A aborts. What kind of undo would work in this scenario?
   (a) Both physical undo and logical undo would succeed.
   (b) Physical undo would fail and logical undo would succeed.
3. Which of the following is correct?
   (a) ARIES works in two phases (after the crash). First redo and then undo.
   (b) ARIES works in three phases (after the crash). First Analysis, then redo, and finally undo.

4. What are the two most important data structures kept by ARIES?

   (a) Transaction Table and Checkpoint Table.

   (b) Transaction Table and Dirty Page Table.

   (c) Dirty Page Table and Checkpoint Table.

5. Which of the following is correct?

   (a) The analysis phase processes the log forward to find the newest checkpoint before the crash.

   (b) The analysis phase actually, besides scanning the log, reconstructs an up-to-date version of the Transaction and Dirty Page Tables that the redo and undo phases will use.

   (c) The redo phase undoes actions based on redo information present in compensation log records.

   (d) The undo phase undoes actions based on redo information present in compensation log records.

6. In the redo phase, does the system log a redo operation?

   (a) No.

   (b) Yes, with an update record.

   (c) Yes, with a compensation log record.

7. In the undo phase, does the system log an undo operation?

   (a) No.

   (b) Yes, with an update record.

   (c) Yes, with a compensation log record.

8. Does ARIES redo actions of loser transactions?

   (a) No.

   (b) Yes.

9. How can you reduce the costs of the redo phase in ARIES?

   (a) By running a background thread that periodically writes back dirty pages to the disk of the database store.

   (b) By running a background thread that periodically writes back dirty pages to stable storage.

   (c) By writing out more fuzzy checkpoints (without flushing dirty pages to the disk of the database store).

10. How can you reduce the costs of the analysis phase in ARIES?

(a) By running a background thread that periodically writes back dirty pages to the disk of the database store.

(b) By running a background thread that periodically writes back dirty pages to stable storage.

(c) By writing out more fuzzy checkpoints (without flushing dirty pages to the disk of the database store).

11. Assume ARIES crashes while being in one of the three recovery phases. Assume no change whatsoever has been done to the database store by ARIES yet. Now, ARIES starts a second time with the recovery process. Which of the three phases may potentially require less write I/O-operations

(a) Analysis

(b) redo

(c) undo

Exercise

Assume we are using a disk-based DBMS using ARIES-style recovery. Our buffer manager can keep 3 data pages in the buffer at any given time and uses LRU as a buffer replacement strategy. With this information you can decide which (possibly dirty) pages get flushed to disk. Checkpointing is performed after every 10^{th} log entry, storing the current TT and DPT.

Consider the following transaction execution example, where the concrete actions are not specified, only the affected page and the transaction performing that action are recorded. Assume empty buffer, TT and DPT in the beginning.

(a) Create a log from the transaction execution example. Add placeholders for undo- and redo information.

(b) Show the state of the buffer after each log entry and, where applicable, the page being flushed.

(c) Show the pageLSNs of the flushed pages (as they are available on disk), the TT and the DPT after the checkpoint and after the last operation.

Actions:

(1) T_1 updates P_1

(2) T_1 updates P_1

(3) T_1 updates P_3

(4) T_2 updates P_1

(5) T_2 updates P_4

(6) T_3 updates P_2

(7) T_1 updates P_1

(8) T_2 updates P_4

(9) T_3 updates P_4

(10) T_4 updates P_5

(11) T_1 updates P_6

(12) T_1 updates P_7

(13) T_2 updates P_3

(14) T_2 updates P_5

(15) T_3 updates P_1

Exercise

Consider the following log:

(1) (–, T_1, "update", P_1, redo_1, undo_1)

(2) (1, T_1, "update", P_2, redo_2, undo_2)

(3) (–, T_2, "update", P_3, redo_3, undo_3)

(4) (begin_checkpoint)

(5) (3, T_2, "update", P_1, redo_5, undo_5)

(6) (end_checkpoint):

Table 6.1: TT

TaID	LastLSN
T_1	2
T_2	3

(7) (5, T_2, "compensation", P_1, undo_5, 3)

(8) (2, T_1, "update", P_3, redo_8, undo_8)

(9) (8, T_1, "update", P_2, redo_9, undo_9)

Table 6.2: DPT

PageID	RecLSN
P_1	1

(10) (9, T_1, "commit")

(11) (–, T_3, "update", P_4, redo$_{11}$, undo$_{11}$)

(12) (11, T_3, "update", P_2, redo$_{12}$, undo$_{12}$)

The system has crashed after the last operation in the log. The pageLSNs of the pages on disk after the crash are:

PageID	**pageLSN**
P_1	–
P_2	9
P_3	3
P_4	–

Perform recovery using the ARIES algorithm, describing what you do according to the following:

1. Where does each phase of the ARIES algorithm begin and end? Be precise about the LSNs.
2. Show the TT and the DPT after the analysis phase.
3. Which operations (i.e. log records) are redone in the Redo phase? Provide a justification when skipping a log record.
4. Show the log after the Undo phase.

Exercise

Consider the log from the previous exercise up to and including LSN 13 which was written during a system recovery (LSN 13 is the last log record written in this scenario). Assume the system tried to recover and in that process flushed all dirty pages after LSN 8. Then it crashed while recovering after writing LSN 13. Show all the possible pageLSNs available on each page immediately after the crash.

Perform a second recovery pass using the ARIES algorithm (now assume that no dirty pages were written back after LSN 8). Describe what you do in the same way as in the previous exercise.

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 .

[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.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, pages 34–43, 1998.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[Cod82] E. F. Codd. Relational Database: A Practical Foundation for Productivity. *Commun. ACM*, 25(2):109–117, February 1982.
ACM 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.

[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.

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[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.

[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.

[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.

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?[retrieved 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: Martin Abegglen

https://www.flickr.com/photos/twicepix/5115400010/

https://creativecommons.org/licenses/by-sa/2.0/legalcode

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

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 second ever given to an Experiments&Analysis 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 reproducibility (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).