CS213
Principles of Database Systems(H)

Chapter 10 Storage and Performance

Shiqi YU 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

10.1 Physical Storage System

Physical Storage System

The hardware where data is recorded

Classification of Physical Storage Media

Can differentiate storage into:

- Volatile storage
  - Loses contents when power is switched off
- Non-volatile storage:
  - Contents persist even when power is switched off
  - Includes secondary and tertiary storage, as well as battery-backed up main-memory

Factors affecting choice of storage media include

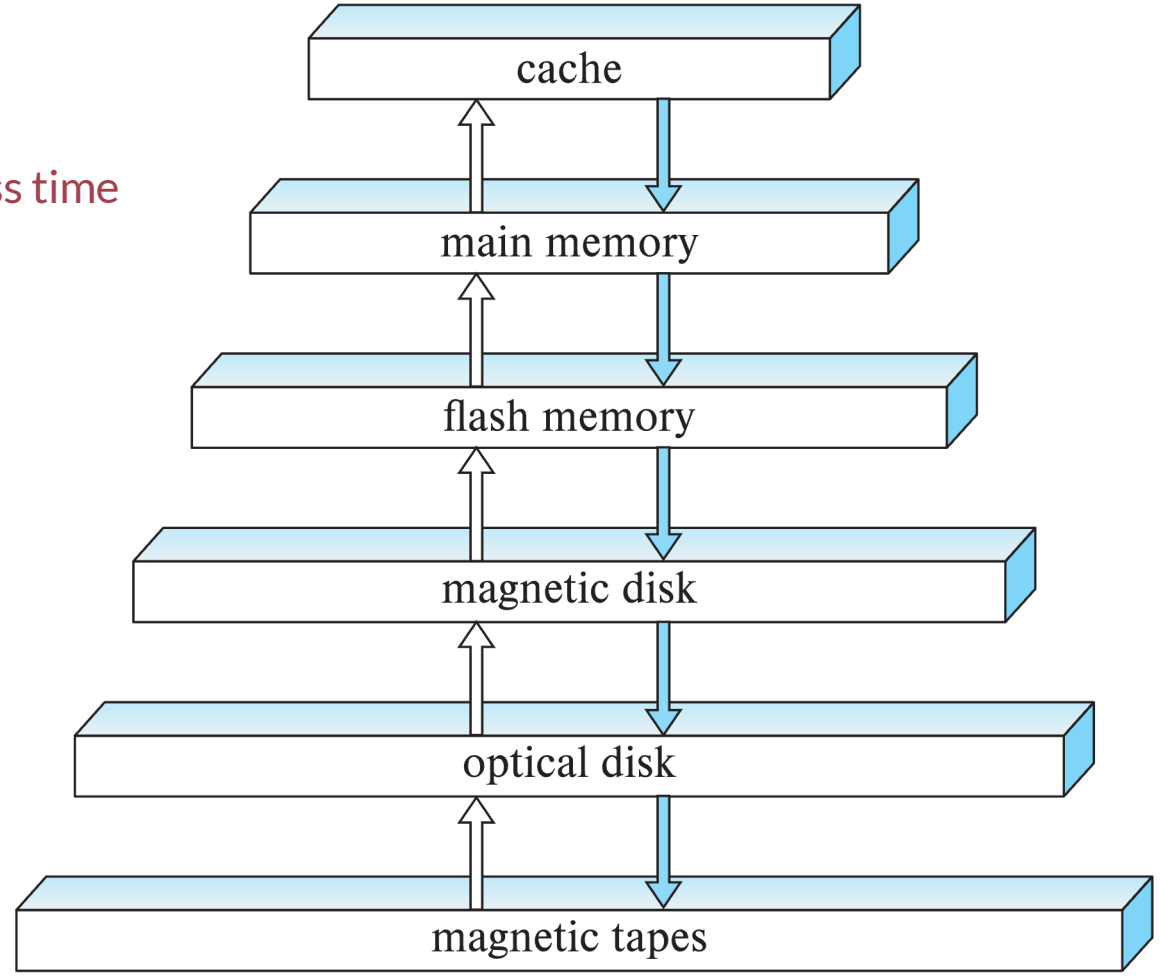
- Speed with which data can be accessed
- Cost per unit of data
- Reliability

Storage Hierarchy

Primary storage
- Fastest media but volatile (cache, main memory).

Secondary storage
- Next level in hierarchy, non-volatile, moderately fast access time
  - … also called on-line storage
- E.g., flash memory, magnetic disks

Tertiary storage
- Lowest level in hierarchy, non-volatile, slow access time
  - … also called off-line storage and used for archival storage
- E.g., magnetic tape, optical storage
  - Magnetic tape
    - Sequential access, 1 to 12 TB capacity
    - A few drives with many tapes
    - Juke boxes with petabytes (1000's of TB) of storage



Storage Interfaces

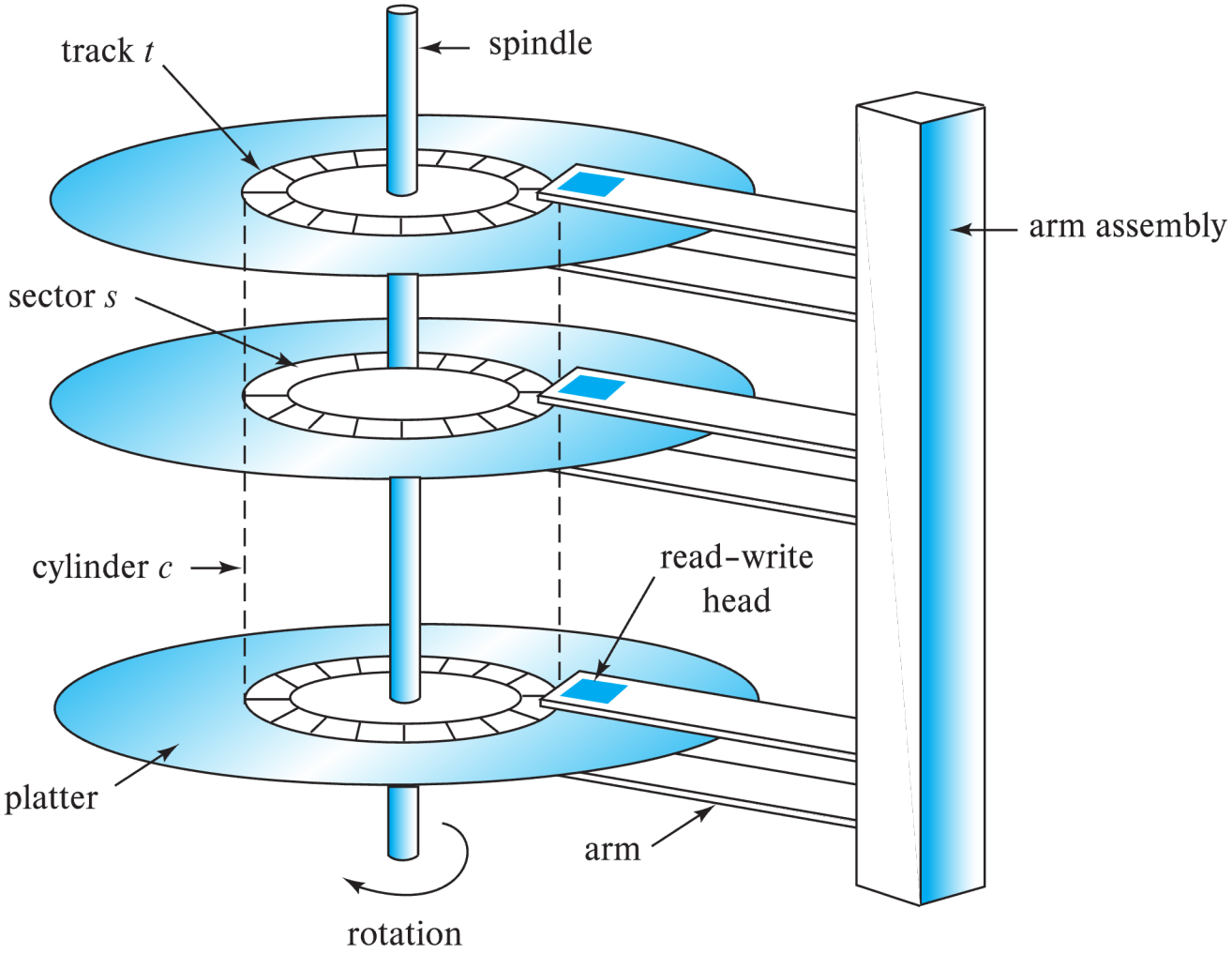
Disk interface standards families

- SATA (Serial ATA)
  - SATA 3 supports data transfer speeds of up to 6 gigabits/sec
- SAS (Serial Attached SCSI)
  - SAS Version 3 supports 12 gigabits/sec
- NVMe (Non-Volatile Memory Express) interface
  - Works with PCIe connectors to support lower latency and higher transfer rates
  - Supports data transfer rates of up to 24 gigabits/sec

Disks usually connected directly to computer system, however…

- In Storage Area Networks (SAN), a large number of disks are connected by a high-speed network to a number of servers
- In Network Attached Storage (NAS) networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface

Magnetic Hard Disk Mechanism



Schematic diagram of magnetic disk drive

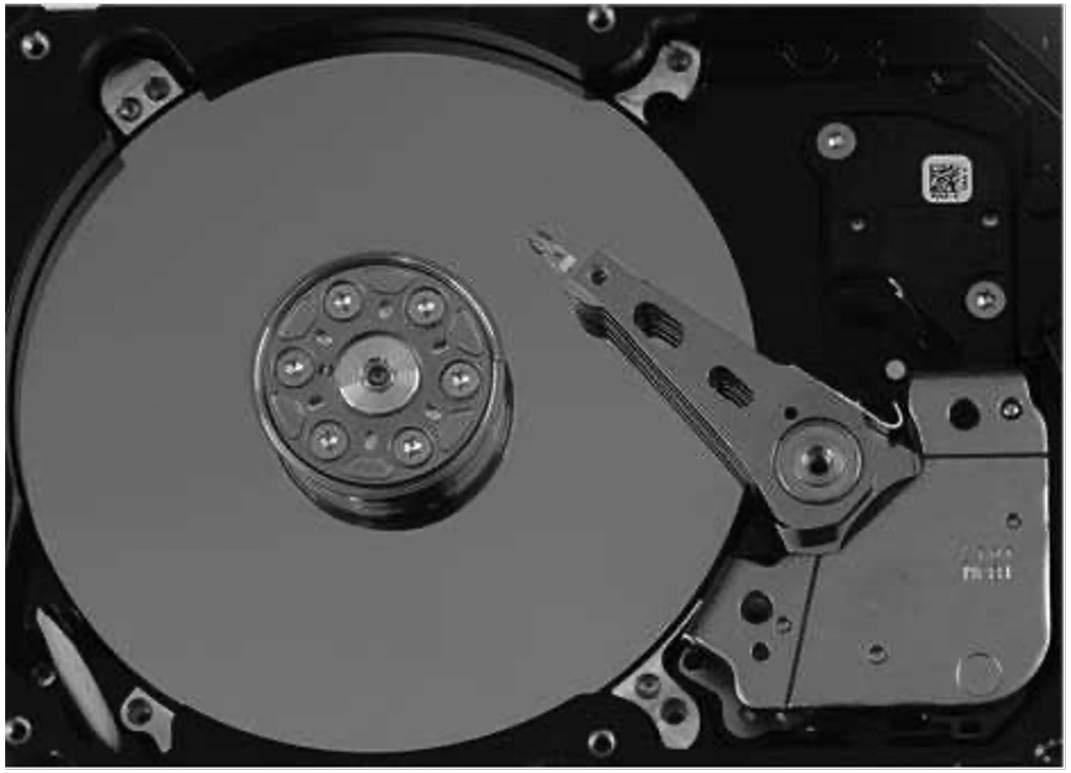


Photo of a magnetic disk drive

Magnetic Hard Disk Mechanism

Read-write head

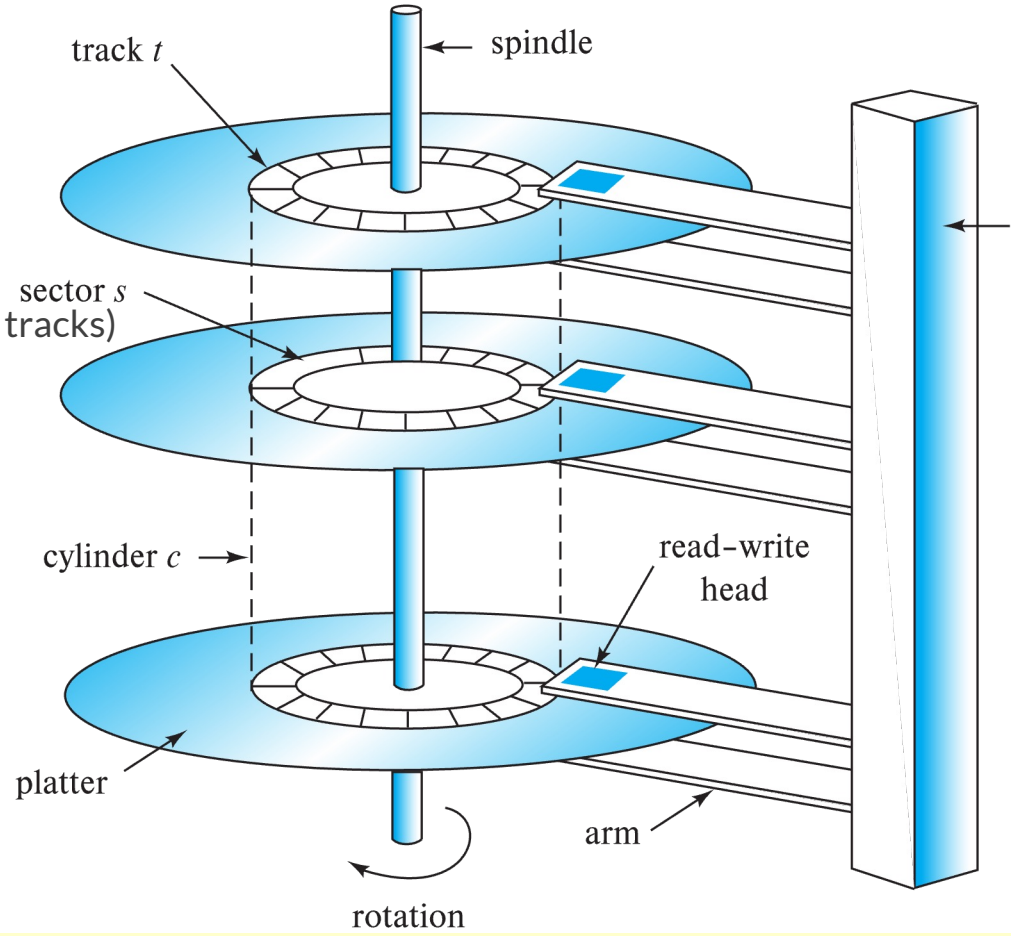
Surface of platter divided into circular tracks
- Over 50K-100K tracks per platter on typical hard disks

Each track is divided into sectors.
- A sector is the smallest unit of data that can be read or written.
- Sector size typically 512 bytes (modern OS requires 4KB)
- Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)

To read/write a sector
- Disk arm swings to position head on right track
- Platter spins continually; data is read/written as sector passes under head

Head-disk assemblies
- Multiple disk platters on a single spindle (1 to 5 usually)
- One head per platter, mounted on a common arm.

Cylinder i consists of ith track of all the platters



Magnetic Hard Disk Mechanism

Disk controller: An interface between the computer system and the disk drive hardware

- Accept high-level commands to read or write a sector
- Initiate actions such as moving the disk arm to the right track and reading or writing the data
- Compute and attach checksums to each sector to verify that data is read back correctly
  - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
- Ensure successful writing by reading back sector after writing it
- Perform remapping of bad sectors

Performance Measures of Disks

Access time: The time it takes from when a read or write request is issued to when data transfer begins. Consists of:

- Seek time – time it takes to reposition the arm over the correct track.
  - Average seek time is 1/2 the worst case seek time.
    - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
  - 4 to 10 milliseconds on typical disks
- Rotational latency – time it takes for the sector to be accessed to appear under the head.
  - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
  - Average latency is 1/2 of the above latency.
- Overall latency is 5 to 20 msec depending on disk model

Data-transfer rate: The rate at which data can be retrieved from or stored to the disk.

- 25 to 200 MB per second max rate, lower for inner tracks

Performance Measures of Disks

Disk block is a logical unit for storage allocation and retrieval

- 4 to 16 kilobytes typically
  - Smaller blocks: more transfers from disk
  - Larger blocks: more space wasted due to partially filled blocks

Sequential access pattern

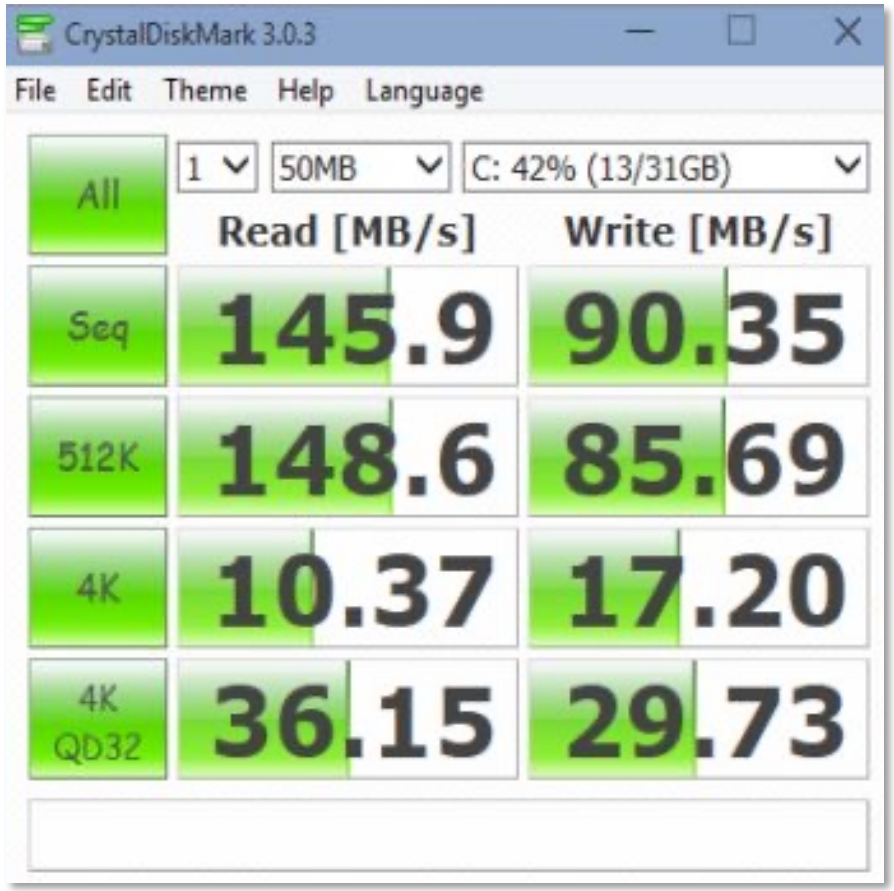
- Successive requests are for successive disk blocks
- Disk seek required only for first block

Random access pattern

- Successive requests are for blocks that can be anywhere on disk
- Each access requires a seek
- Transfer rates are low since a lot of time is wasted in seeks

I/O operations per second (IOPS)

- Number of random block reads that a disk can support per second
- 50 to 200 IOPS on current generation magnetic disks



Performance Measures of Disks

Mean time to failure (MTTF) – the average time the disk is expected to run continuously without any failure.

- Typically, 3 to 5 years
- Probability of failure of new disks is quite low, corresponding to a "theoretical MTTF" of 500,000 to 1,200,000 hours for a new disk
  - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
- MTTF decreases as disk ages

Flash Storage

NOR flash vs NAND flash

NAND flash

- Used widely for storage, cheaper than NOR flash
- Requires page-at-a-time read (page: 512 bytes to 4 KB)
  - 20 to 100 microseconds for a page read
  - Not much difference between sequential and random read
- Page can only be written once
  - Must be erased to allow rewrite

Solid state disks (SSD)

- Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
- Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe

Flash Storage

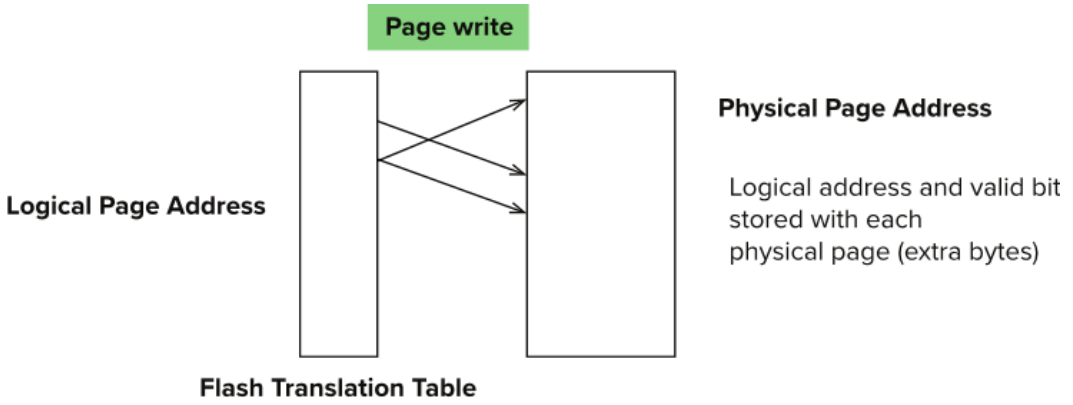
Erase happens in units of erase block

- Takes 2 to 5 millisecs
- Erase block typically 256 KB to 1 MB (128 to 256 pages)

Remapping of logical page addresses to physical page addresses avoids waiting for erase

Flash translation table tracks mapping

- Also stored in a label field of flash page
- Remapping carried out by flash translation layer



After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used

- Wear leveling

SSD Performance Metrics

Random reads/writes per second

- Typical 4KB reads: 10,000 reads per second (10,000 IOPS)
- Typical 4KB writes: 40,000 IOPS
- SSDs support parallel reads
  - Typical 4KB reads:
    - 100,000 IOPS with 32 requests in parallel (QD-32) on SATA
    - 350,000 IOPS with QD-32 on NVMe PCIe
  - Typical 4KB writes:
    - 100,000 IOPS with QD-32, even higher on some models

Data transfer rate for sequential reads/writes

- 400 MB/sec for SATA3, 2 to 3 GB/sec using NVMe PCIe

Hybrid disks: Combine small amount of flash cache with larger magnetic disk

Storage Class Memory

3D-XPoint memory technology pioneered by Intel

Available as Intel Optane

- SSD interface shipped from 2017
  - Allows lower latency than flash SSDs
- Non-volatile memory interface announced in 2018
  - Supports direct access to words, at speeds comparable to main-memory speeds

Magnetic Tapes

Hold large volumes of data and provide high transfer rates

- Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
- Transfer rates from few to 10s of MB/s

Tapes are cheap, but cost of drives is very high

Very slow access time in comparison to magnetic and optical disks

- limited to sequential access.
- Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity

Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.

Tape jukeboxes used for very large capacity storage

- Multiple petabytes (10^{15} bytes)

RAID

RAID: Redundant Arrays of Independent Disks

- Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
  - high capacity and high speed by using multiple disks in parallel
  - high reliability by storing data redundantly, so that data can be recovered even if a disk fails

The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.

- E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
- Techniques for using redundancy to avoid data loss are critical with large numbers of disks

Improvement of Reliability via Redundancy

Redundancy – Store extra information that can be used to rebuild information lost in a disk failure

- E.g., Mirroring (or shadowing)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
    - Probability of combined event is very small
    - Except for dependent failure modes such as fire or building collapse or electrical power surges
- Mean time to data loss depends on mean time to failure, and mean time to repair
  - E.g., MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500*106 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

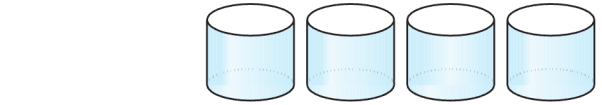
RAID Levels

Schemes to provide redundancy at lower cost by using disk striping combined with parity bits

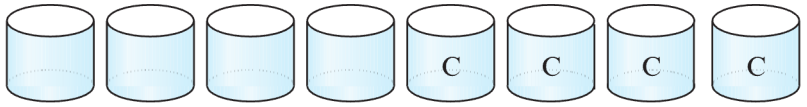
- Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics

- RAID 0: Block striping; non-redundant
- RAID 1: Mirrored disks with block striping
- RAID 10: Combination of striping and mirroring

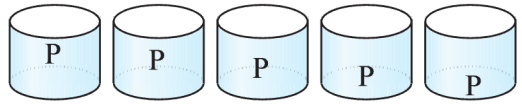
- RAID 5: Block-interleaved distributed parity
- RAID 6: P+Q Redundancy scheme



(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity

Optimization of Disk-Block Access

Buffering

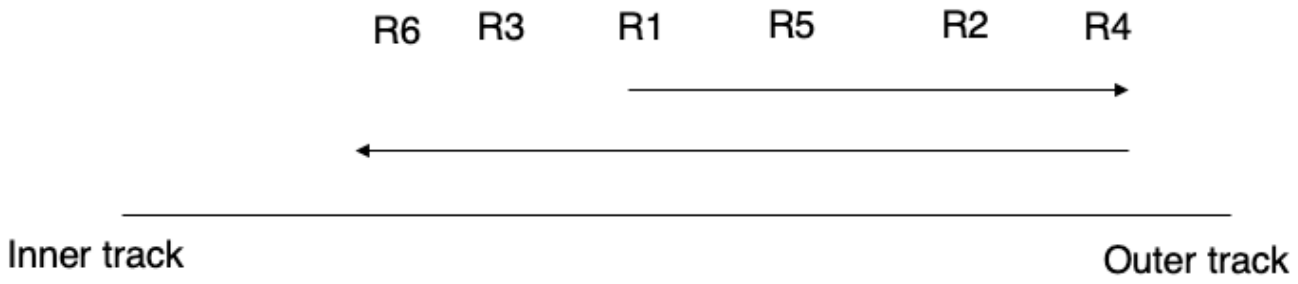
- In-memory buffer to cache disk blocks

Read-ahead

- Read extra blocks from a track in anticipation that they will be requested soon

Disk-arm-scheduling algorithms

- Re-order block requests so that disk arm movement is minimized
- E.g., elevator algorithm



Optimization of Disk-Block Access

File organization

- Allocate blocks of a file in as contiguous a manner as possible
- Allocation in units of extents
- Files may get fragmented
  - E.g., if free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
  - Sequential access to a fragmented file results in increased disk arm movement
  - Some systems have utilities to defragment the file system, in order to speed up file access

Non-volatile write buffers

- Temporarily store the written data
  - … and immediately notifies the OS that writing is completed without errors
- Write data into the disk when idle
  - … with some optimizations

10.2 (Logical) Data Storage Structure

File Organization

The database is stored as a collection of files

- Each file is a sequence of records
- A record is a sequence of fields.

One approach

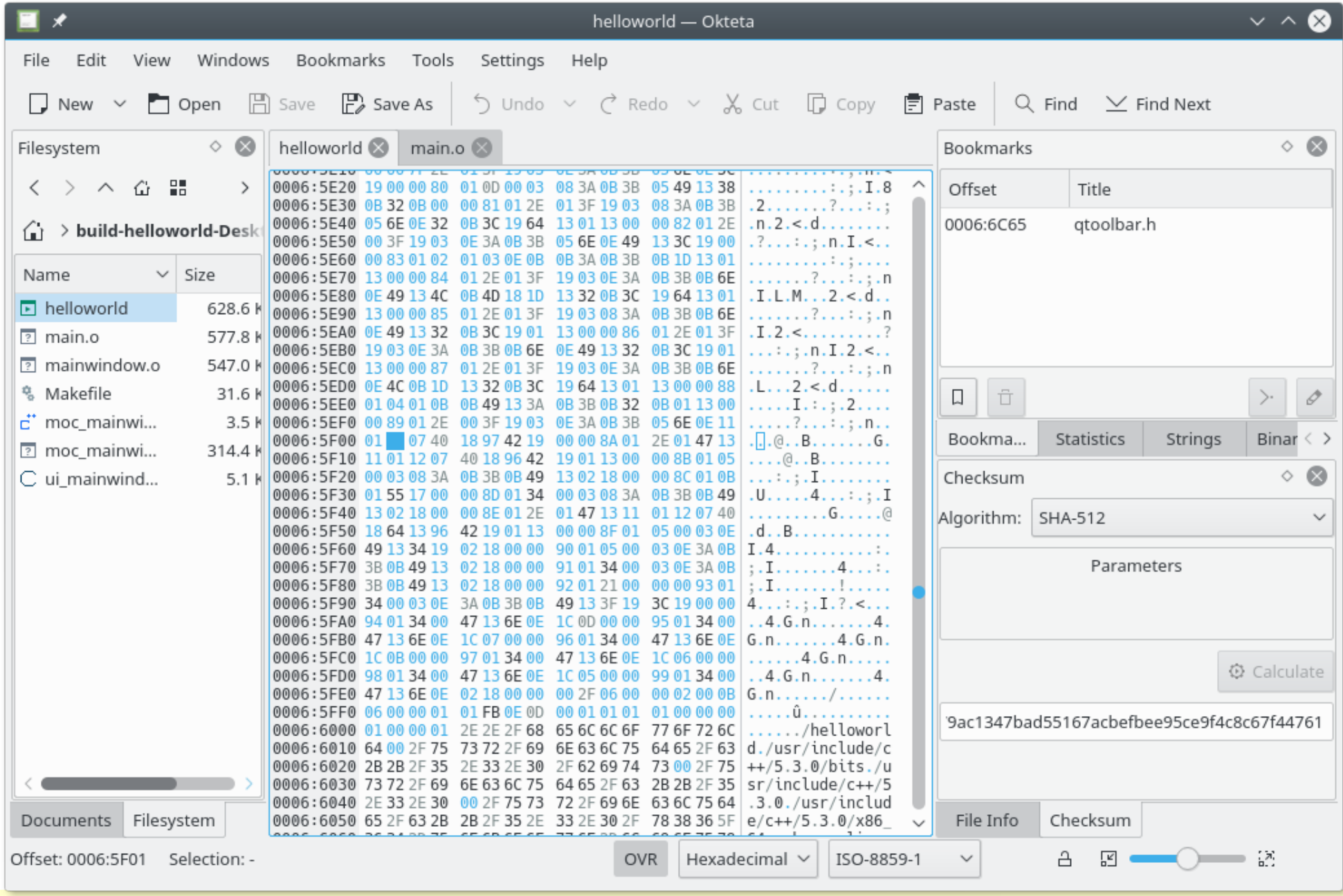
- Assume record size is fixed
- Each file has records of one particular type only
- Different files are used for different relations

* This case is easiest to implement; we will consider variable length records later

We assume that records are smaller than a disk block

File Organization

Bitmap of a file



File Organization

Goals: Time and Space

- Support CURD operations as fast as possible
- Save storage space as much as possible

- Also, to some extent, maintain data integrity

Fixed-Length Records

Simple approach:

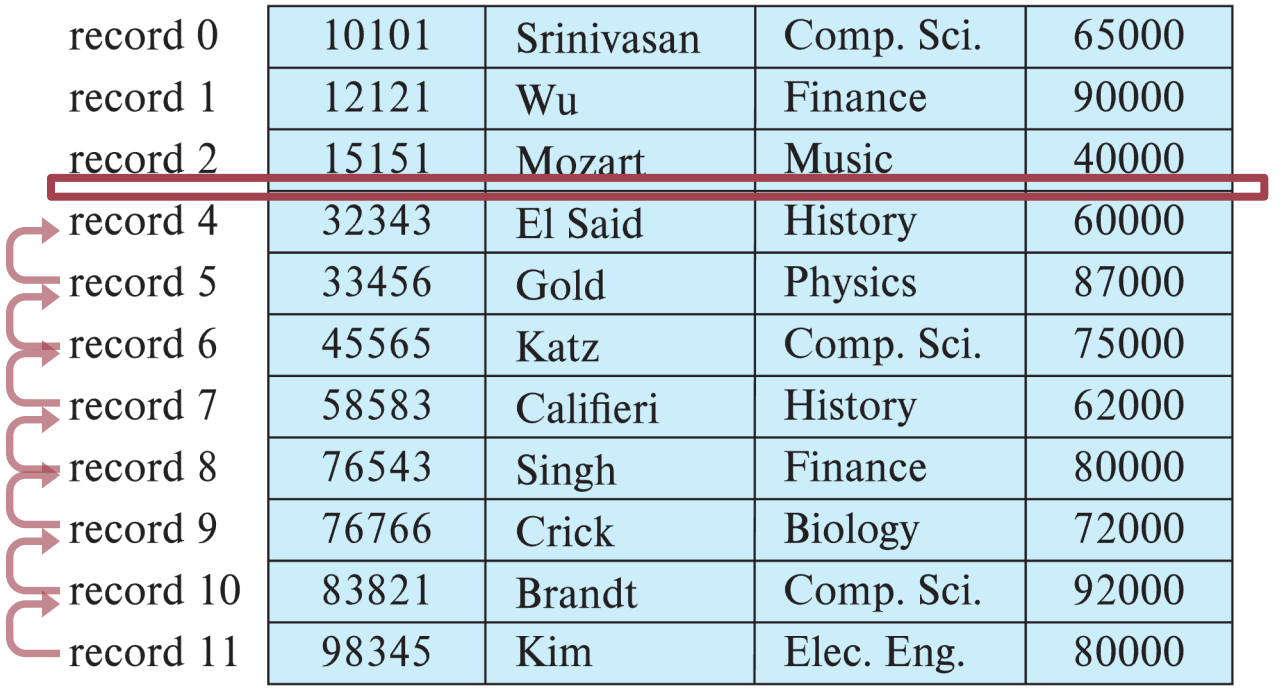
- Store record i starting from byte $n*(i-1)$, where n is the size of each record
- Record access is simple, but records may cross blocks
  - Modification: do not allow records to cross block boundaries

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Fixed-Length Records

Deletion of record i

- Way #1: move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$

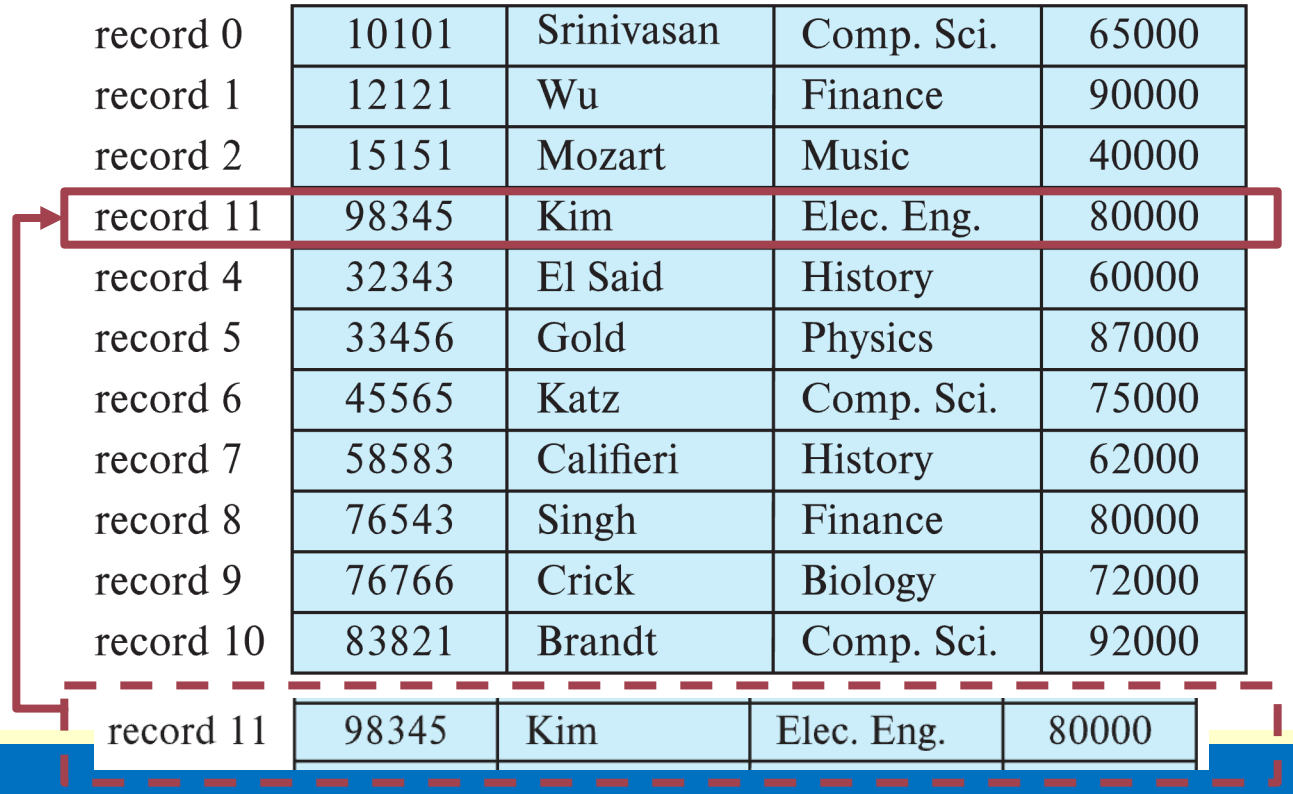


record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Fixed-Length Records

Deletion of record i

- Way #2: move record n to i
  - Record 3 is removed and replaced by record 11



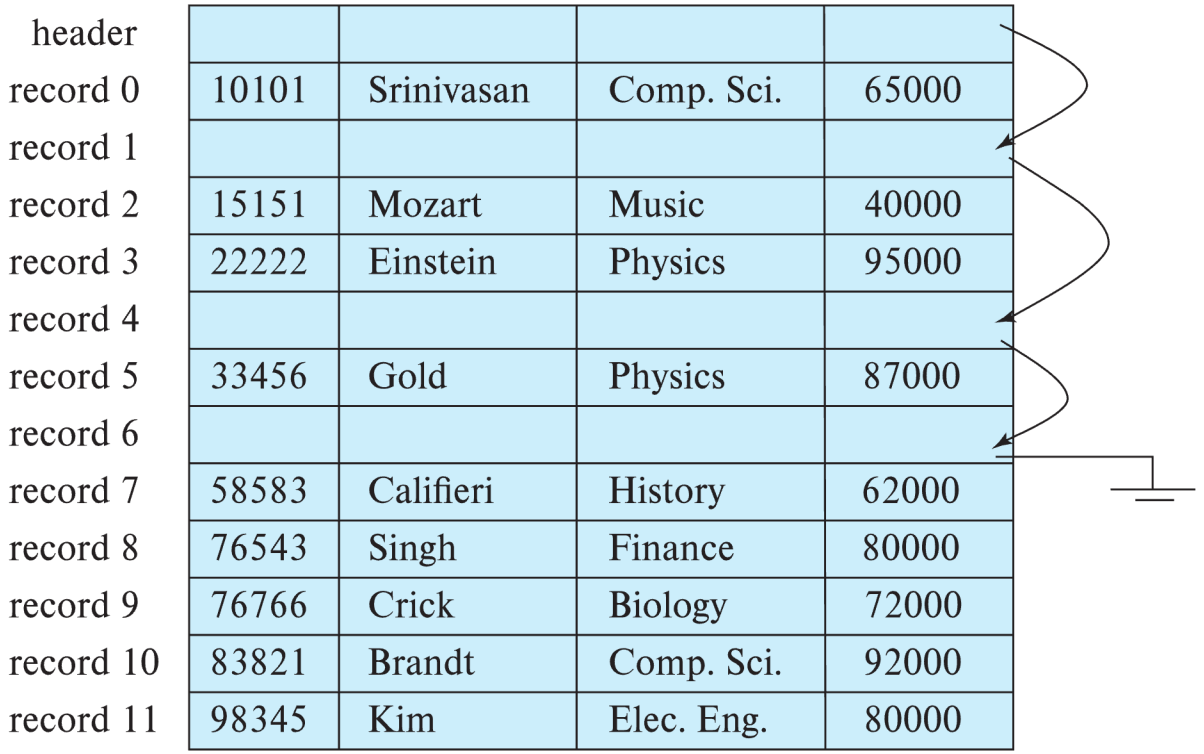
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

record 11	98345	Kim	Elec. Eng.	80000

Fixed-Length Records

Deletion of record i

- Way #3: Do not move records, but link all free records on a *free list*



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length Records

Variable-length records arise in database systems in several ways:

- Storage of multiple record types in a file.
- Record types that allow variable lengths for one or more fields such as strings (`varchar`)
- Record types that allow repeating fields (used in some older data models).

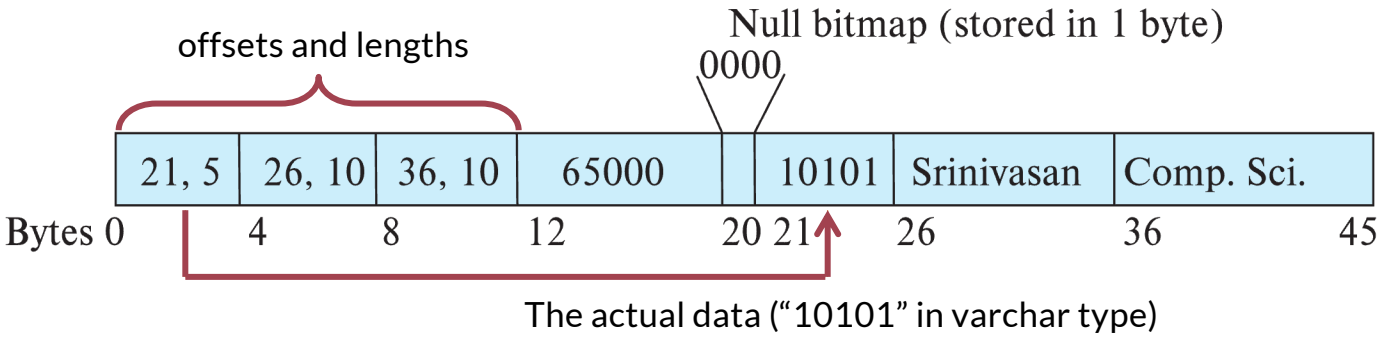
Problem with variable-length records

- How can we retrieve the data in an easy way without wasting too much space
  - `varchar(1000)`: do we really need to allocate 1000 bytes for this field, even if most of the actual data items only costs less than 10 bytes?

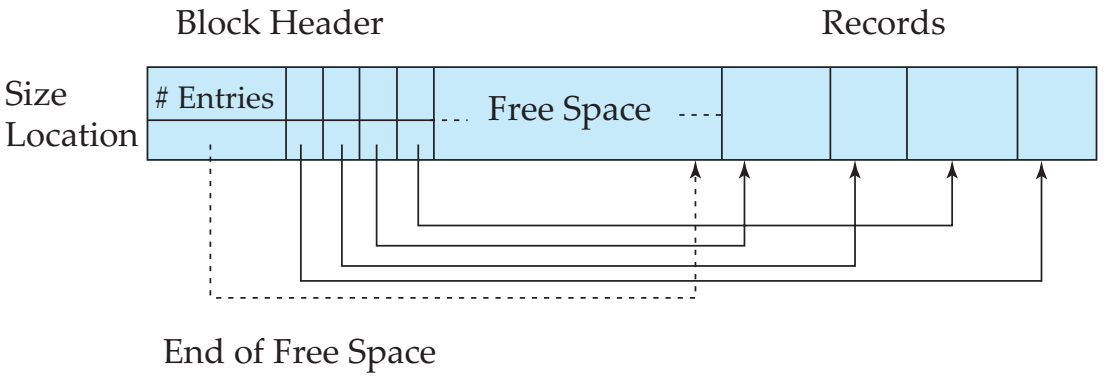
Variable-Length Records

Attributes are stored in order

- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Variable-Length Records



Slotted page header contains:

- number of record entries
- end of free space in the block
- location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated

Pointers should not point directly to records — instead, they should point to the entry for the record in header

Page size is usually aligned with the disk block size (4KB-8KB)

Storing Large Objects

E.g., BLOB/CLOB types

- BLOB: Binary Large OBject
- CLOB: Character Large OBject

Records must be smaller than pages

Alternatives:

- Store as files in file systems
- Store as files managed by databases
- Break into pieces and store in multiple tuples in separate relation
  - PostgreSQL TOAST

Organization of Records in Files

Heap – records can be placed anywhere in the file where there is space

Sequential – store records in sequential order, based on the value of the search key of each record

Multitable clustering file organization
- Records of several different relations can be stored in the same file
- Motivation: store related records on the same block to minimize I/O

B+-tree file organization
- Ordered storage even with inserts/deletes

Hashing – a hash function computed on search key; the result specifies in which block of the file the record should be placed

10.3 Index

Shiqi Yu 于仕琪

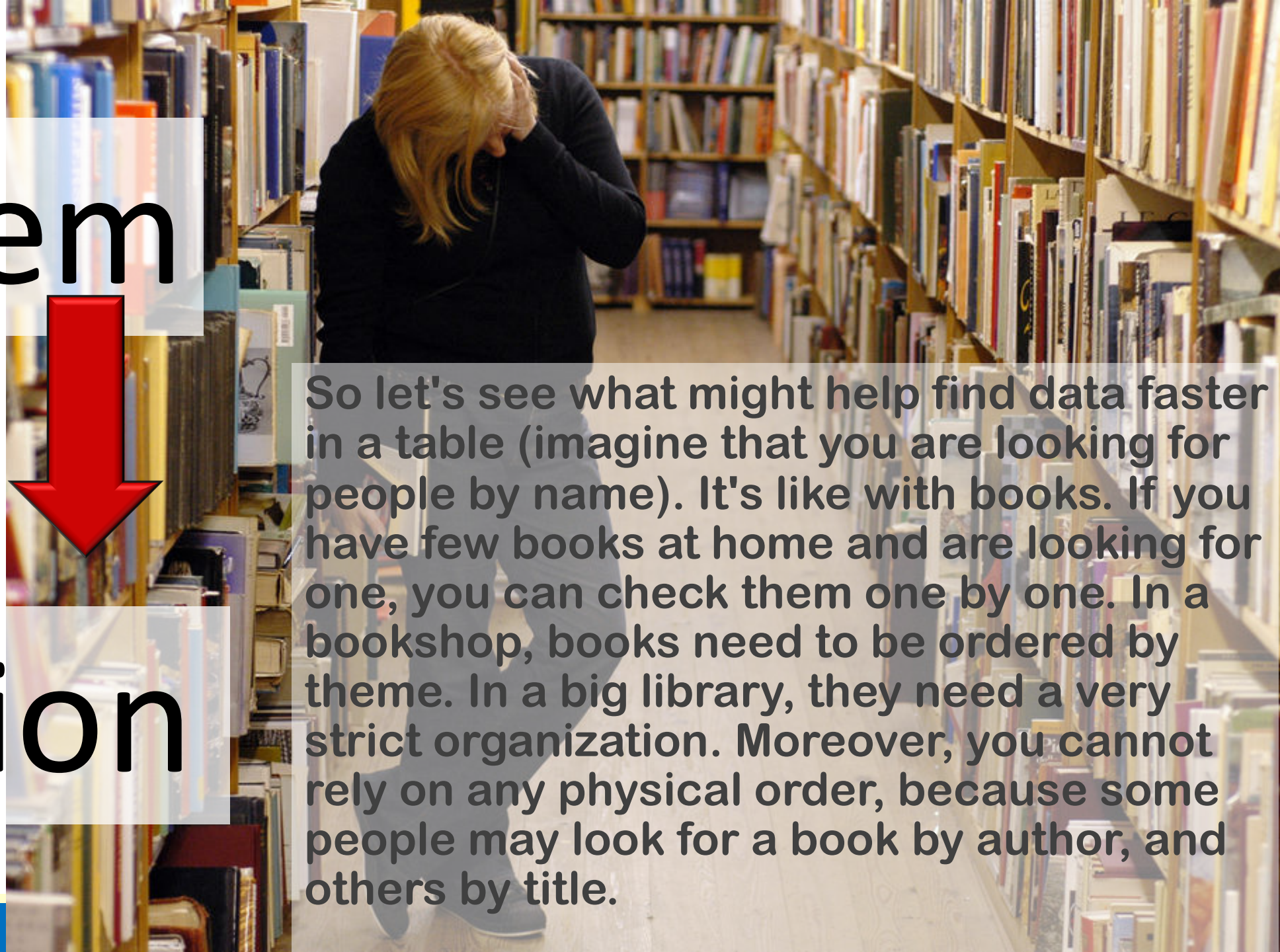
yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

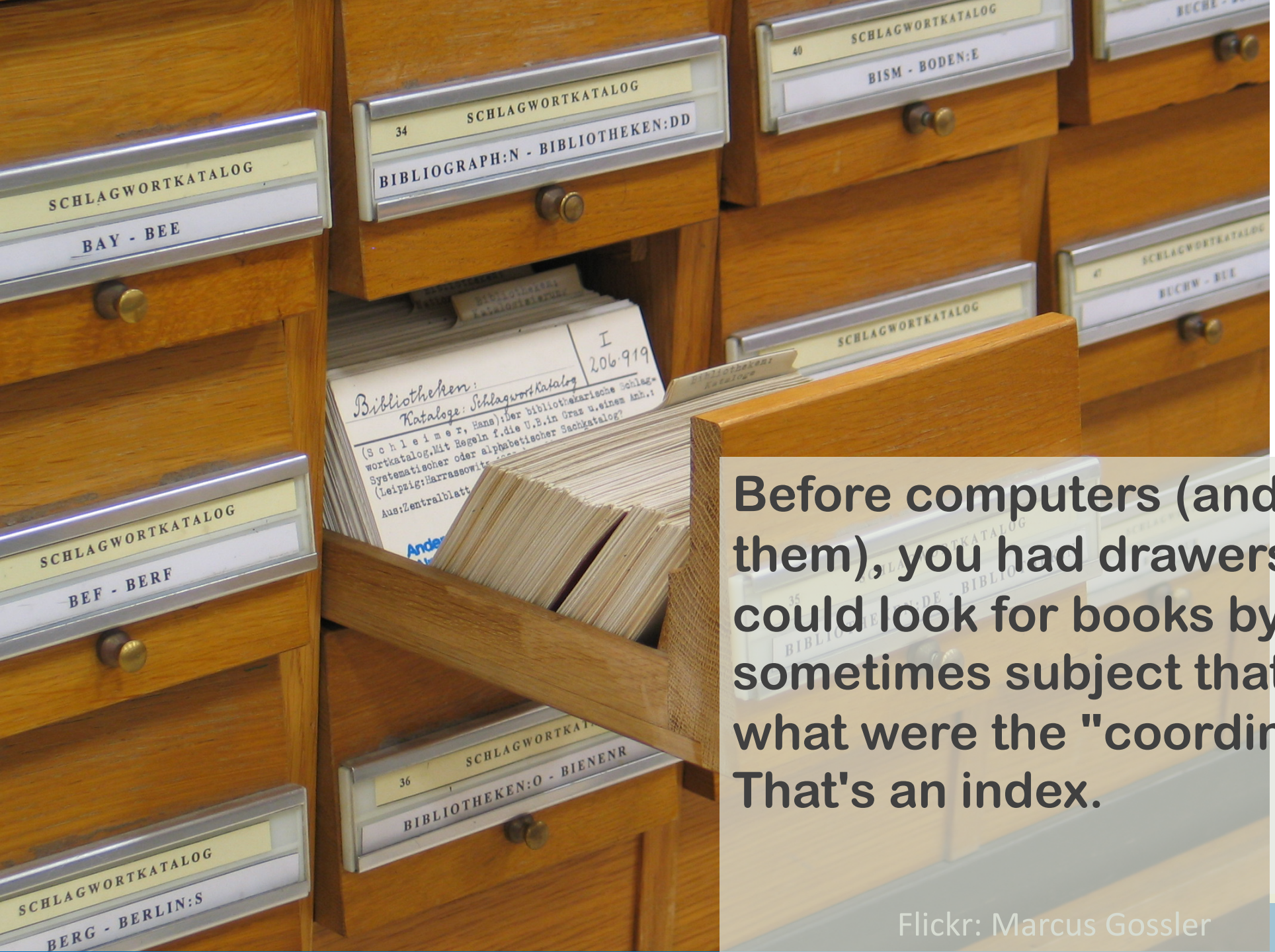
Problem

SQL

Solution



So let's see what might help find data faster in a table (imagine that you are looking for people by name). It's like with books. If you have few books at home and are looking for one, you can check them one by one. In a bookshop, books need to be ordered by theme. In a big library, they need a very strict organization. Moreover, you cannot rely on any physical order, because some people may look for a book by author, and others by title.



Before computers (and you can still find them), you had drawers where you could look for books by author, title or sometimes subject that were telling you what were the "coordinates" of a book. That's an index.

Flickr: Marcus Gossler

file 002

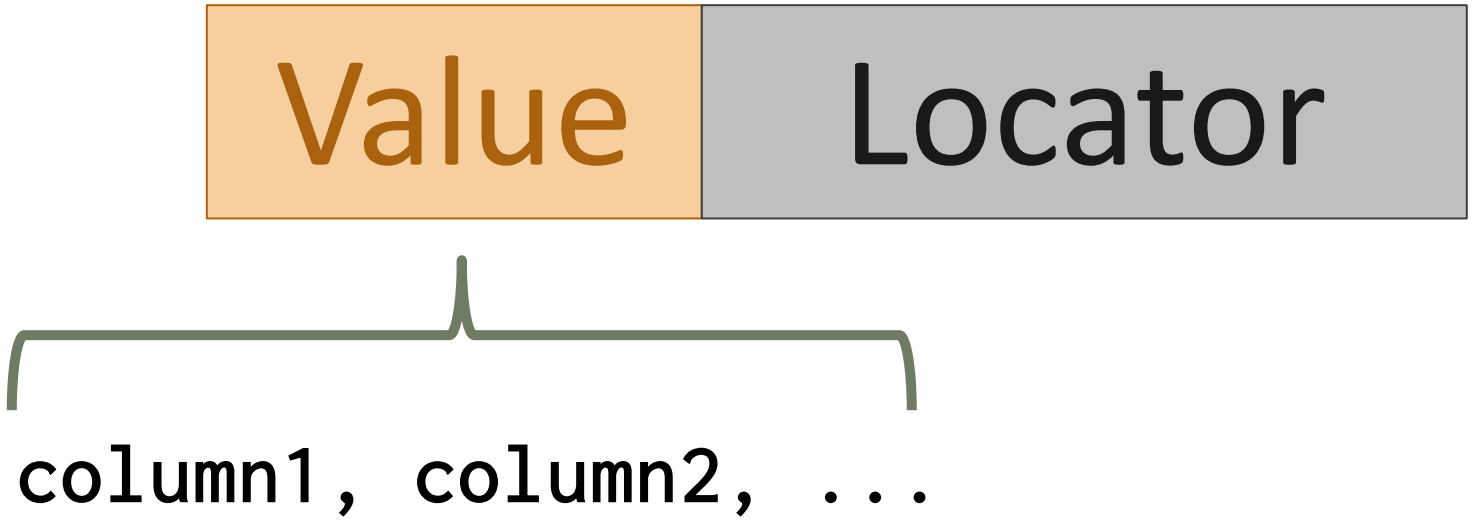
block #6

783 bytes from start

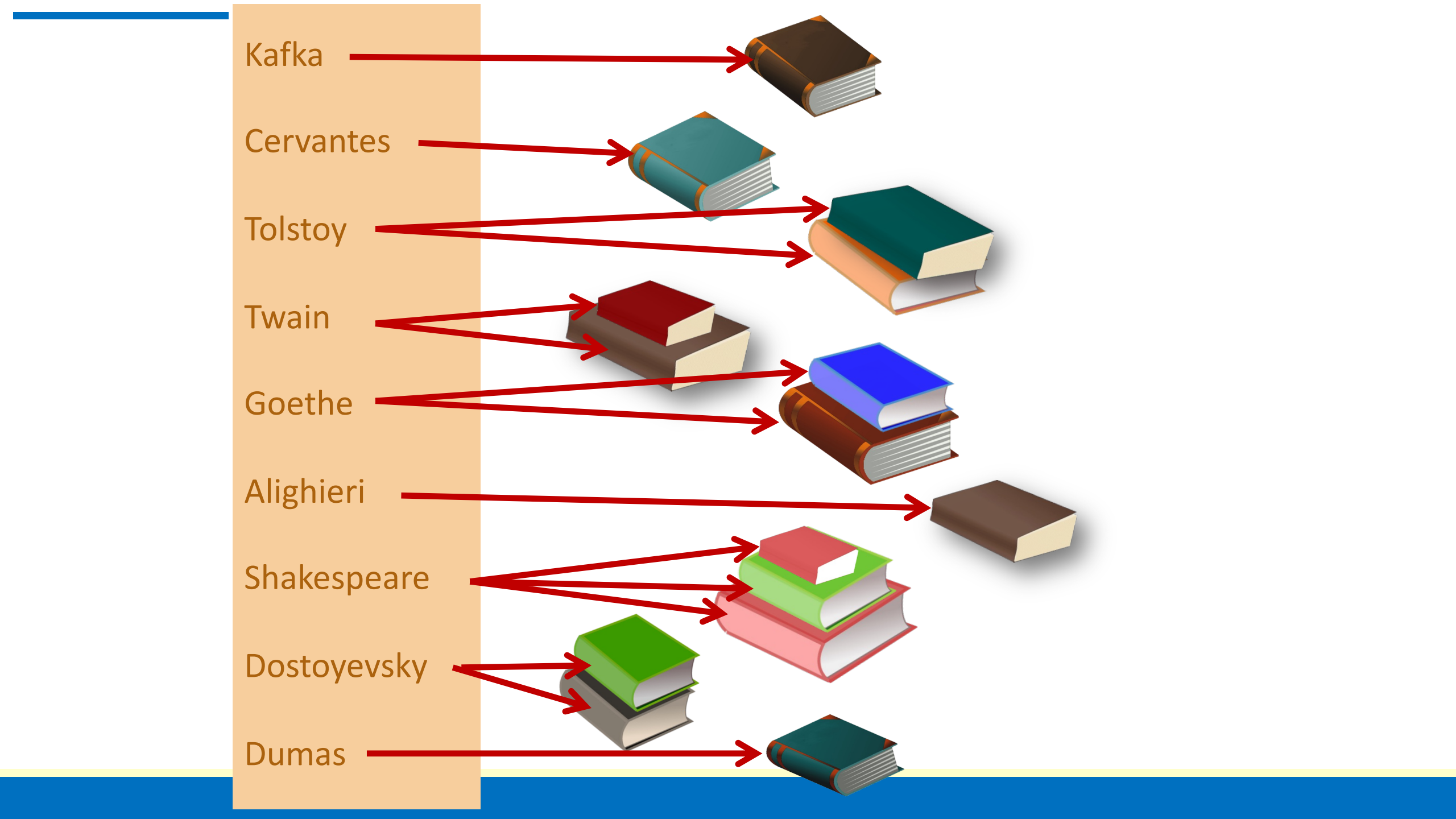
Row locator

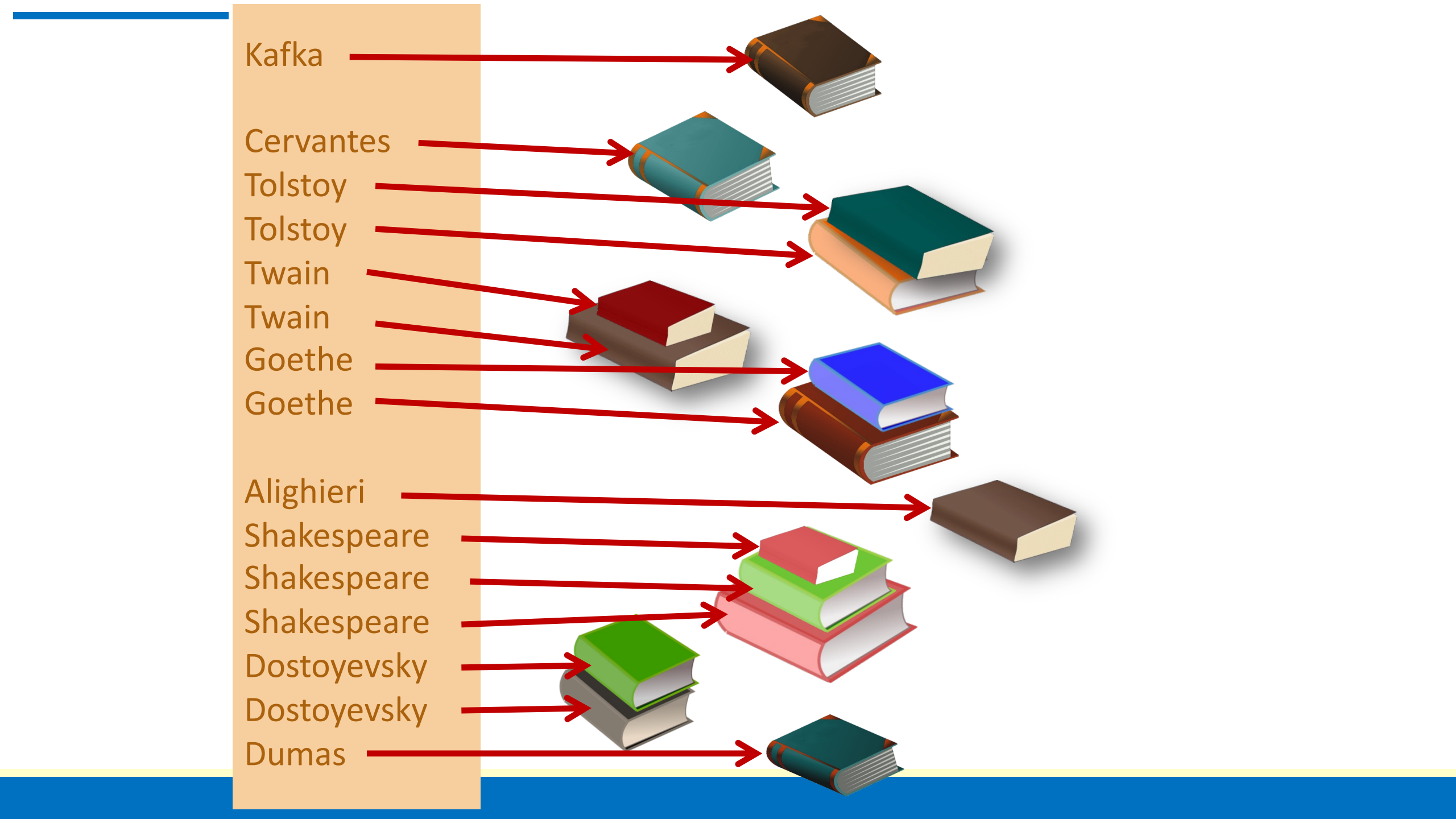
We'll see the structure in more detail later, but a database is made of files, themselves usually organized in equal-sized "pages" (or "blocks"). File, block and offset allow to locate very fast any row.

The whole idea of indexing consists in associating values in one or several columns of a table (we may look by groups of columns, such as FIRST_NAME and SURNAME in the PEOPLE table, and want to index them as a combination) to the locator(s) of the row(s) where they can be found.

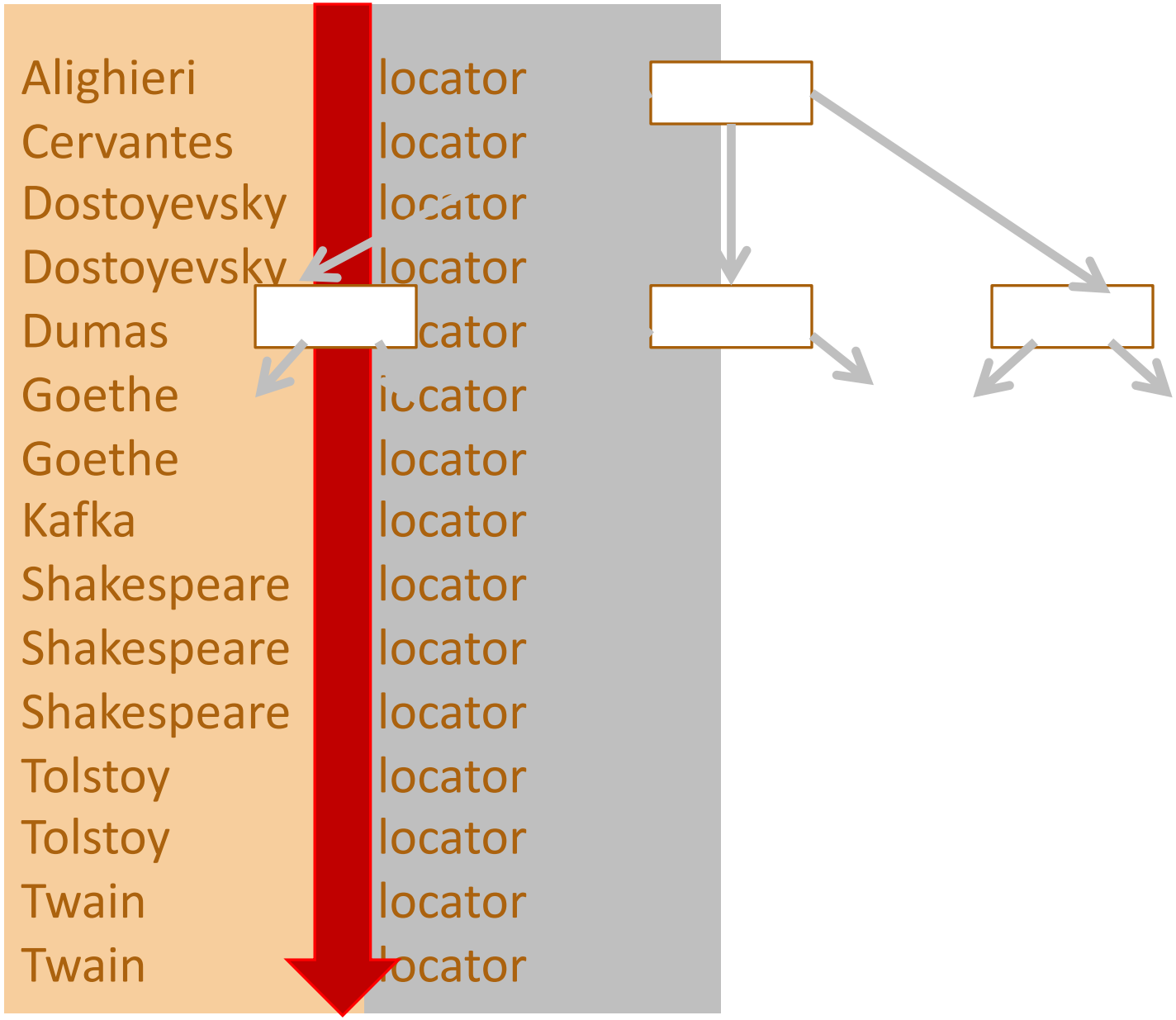


We build a sorted list of all the values with their locators.

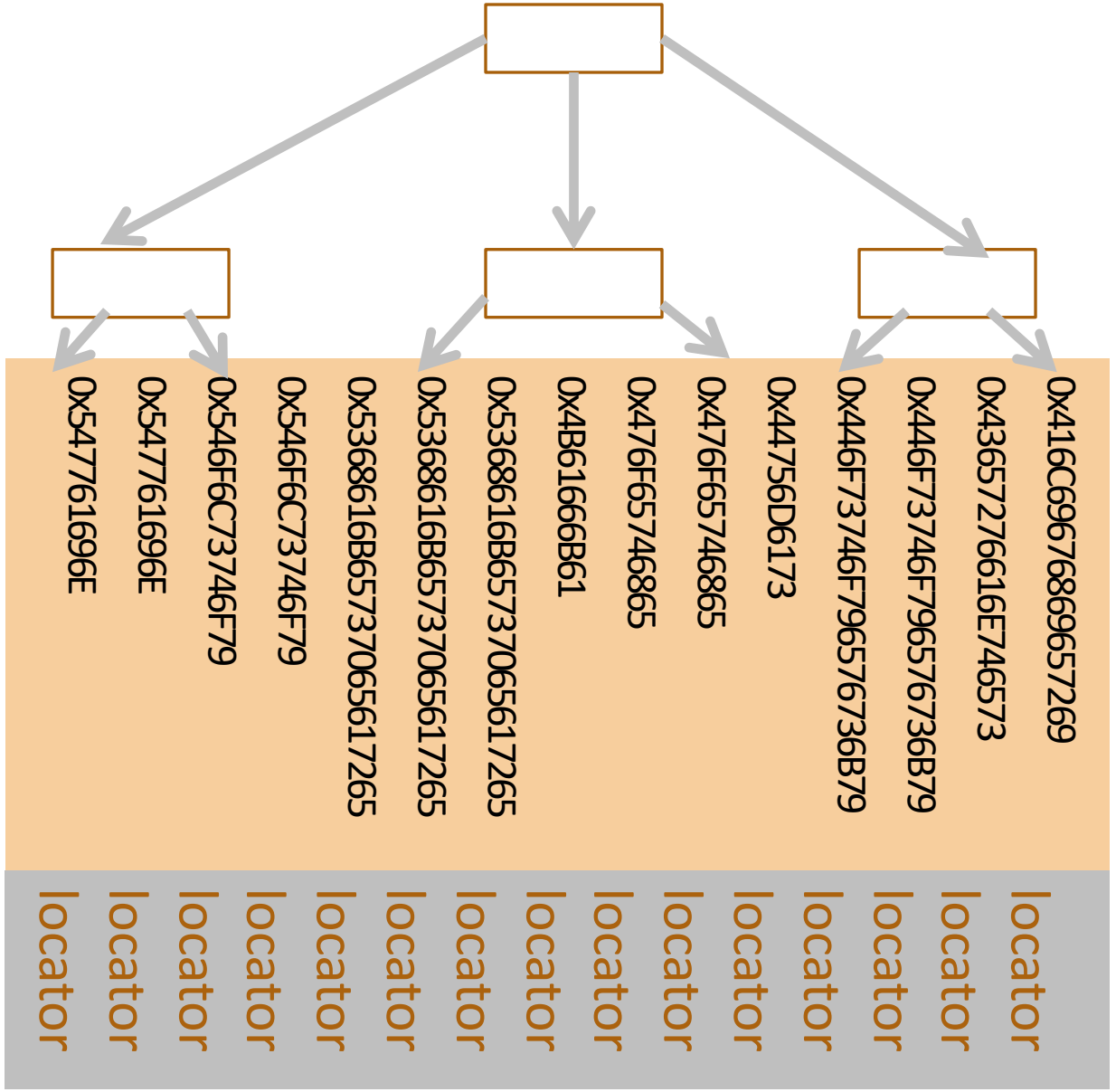

Kafka
Cervantes
Tolstoy
Twain
Goethe
Alighieri
Shakespeare
Dostoyevsky
Dumas


Kafka
Cervantes
Tolstoy
Tolstoy
Twain
Twain
Goethe
Goethe
Alighieri
Shakespeare
Shakespeare
Shakespeare
Dostoyevsky
Dostoyevsky
Dumas

Kafka	locator
Cervantes	locator
Tolstoy	locator
Tolstoy	locator
Twain	locator
Twain	locator
Goethe	locator
Goethe	locator
Alighieri	locator
Shakespeare	locator
Shakespeare	locator
Shakespeare	locator
Dostoyevsky	locator
Dostoyevsky	locator
Dumas	locator


Alighieri locator
Cervantes locator
Dostoyevsky locator
Dostoyevsky locator
Dumas locator
Goethe locator
Goethe locator
Kafka locator
Shakespeare locator
Shakespeare locator
Shakespeare locator
Tolstoy locator
Tolstoy locator
Twain locator
Twain locator

Remember that indexed values are binary values, that will be important!



To search the list easily, we plug a tree above it.

You create an index by giving it a name and specifying table name and column(s)

```
create index <index name>
on <table name>(<col1>, ... <coln>)
```

Example:

```
create index countries_cont_idx
on countries(continent)
```

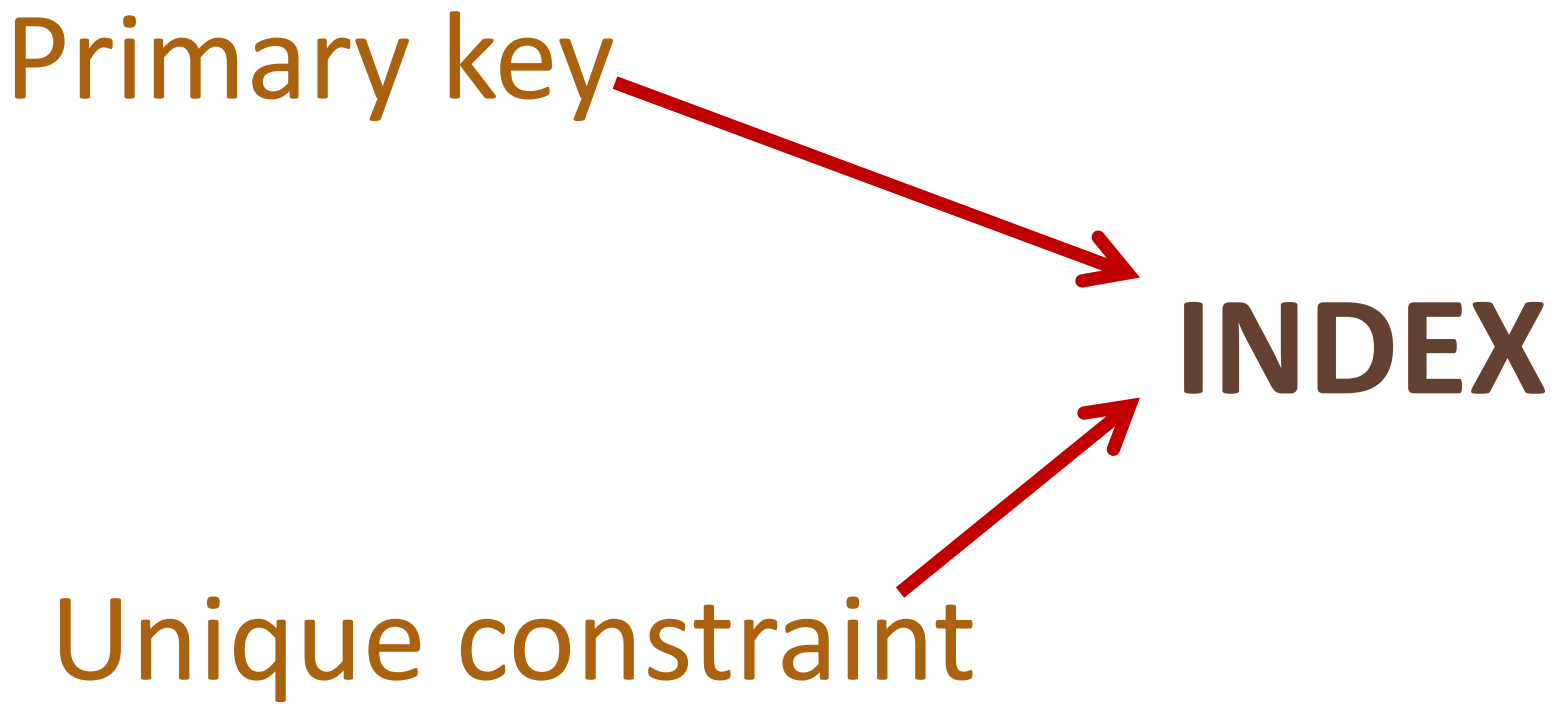
Two columns often queried together can be indexed together; what is indexed is concatenated values (NOT separate values)

```
create index people_surname_born_idx
on people(surname, born)
```

Composite index

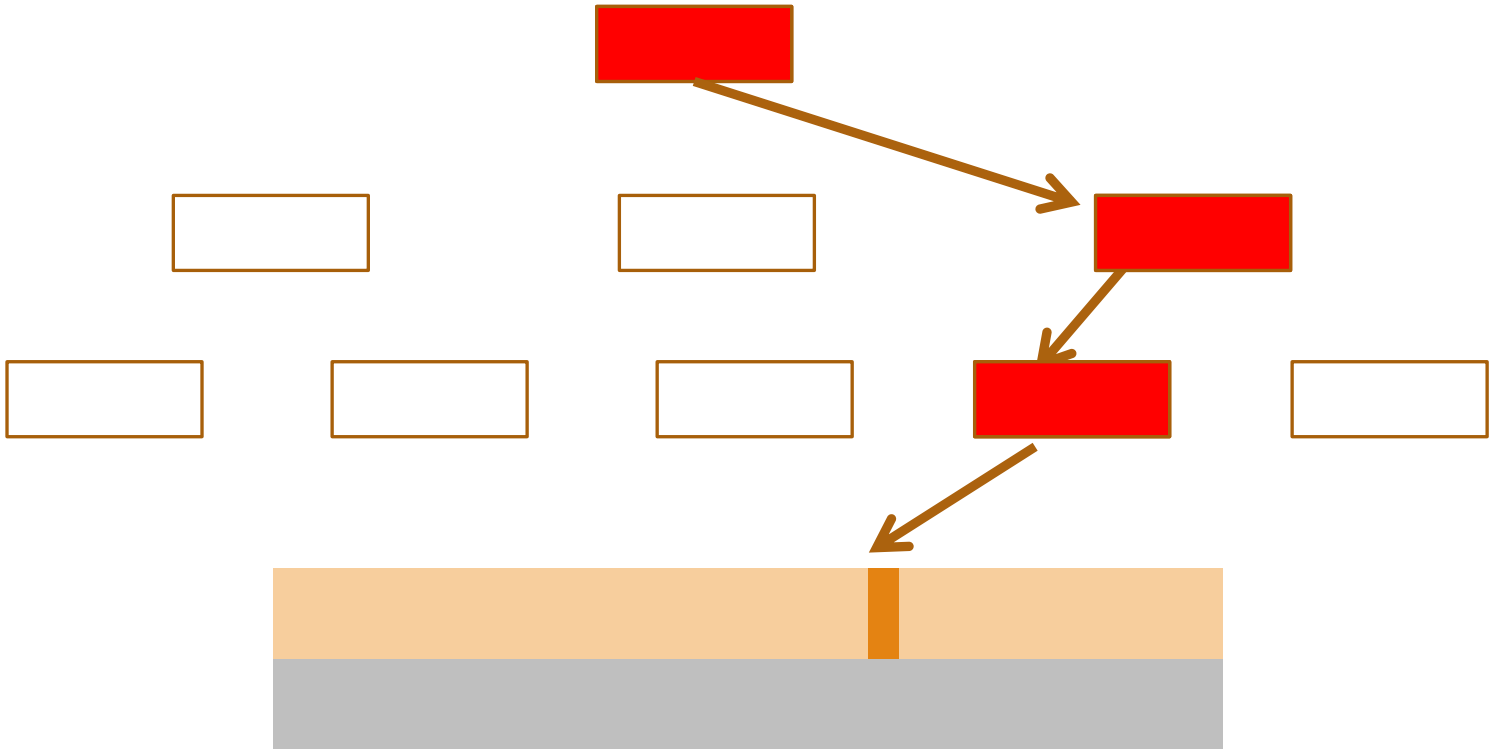
You have actually already created indexes without knowing it: whenever you declare a PRIMARY KEY or UNIQUE constraint, an index is created behind your back.

```
create table movies
(
    movieid       integer      not null
        constraint movies_pkey
            primary key,
    title         varchar(100) not null
        constraint "title length"
            check (length((title)::text) <= 100),
    country       char(2)      not null
        constraint movies_country_fkey
            references countries
        constraint "country length"
            check (length(country) <= 2),
    year_released integer      not null
        constraint "year_released numerical"
            check ((year_released + 0) = year_released),
    runtime       integer
        constraint "runtime numerical"
            check ((runtime + 0) = runtime),
    constraint movies_title_country_year_released_key
        unique (title, country, year_released)
);
```



It has nothing to do with constraints or the relational theory (with indexes, we are more talking engineering than theory), it's purely practical.

The only way to find quickly in a big table that a supposedly unique value is already recorded is to index the column



Oooops ….

Already there …

INSERT
DELETE

Then we should certainly index all columns? Why isn't it done by default? In fact, everything isn't rosy: insertion and deletion always require maintaining table AND indexes. Quite a lot of work.

Table

+ Index

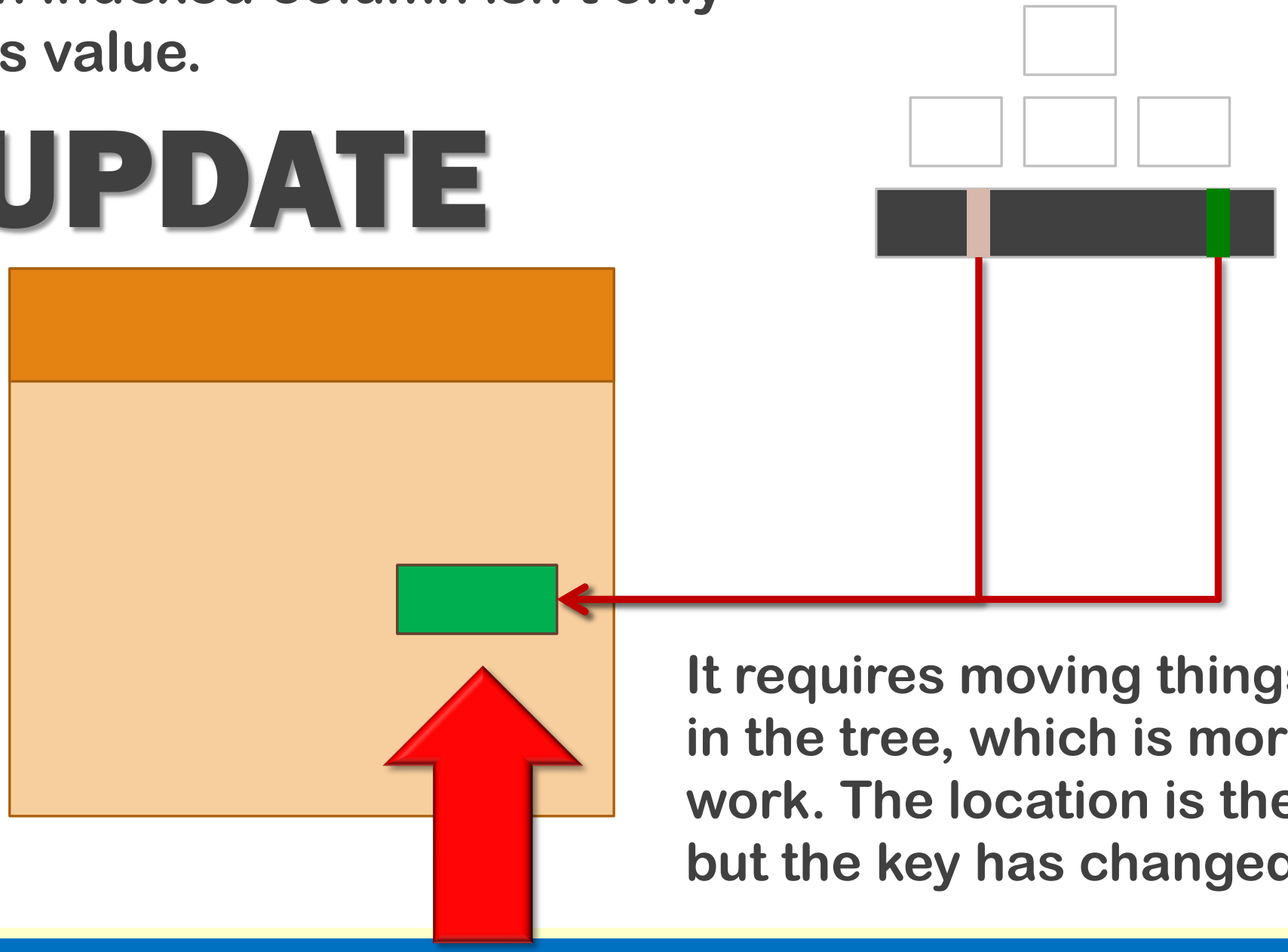
+ Index

+ Index

+ Index

Updating an indexed column isn't only changing its value.

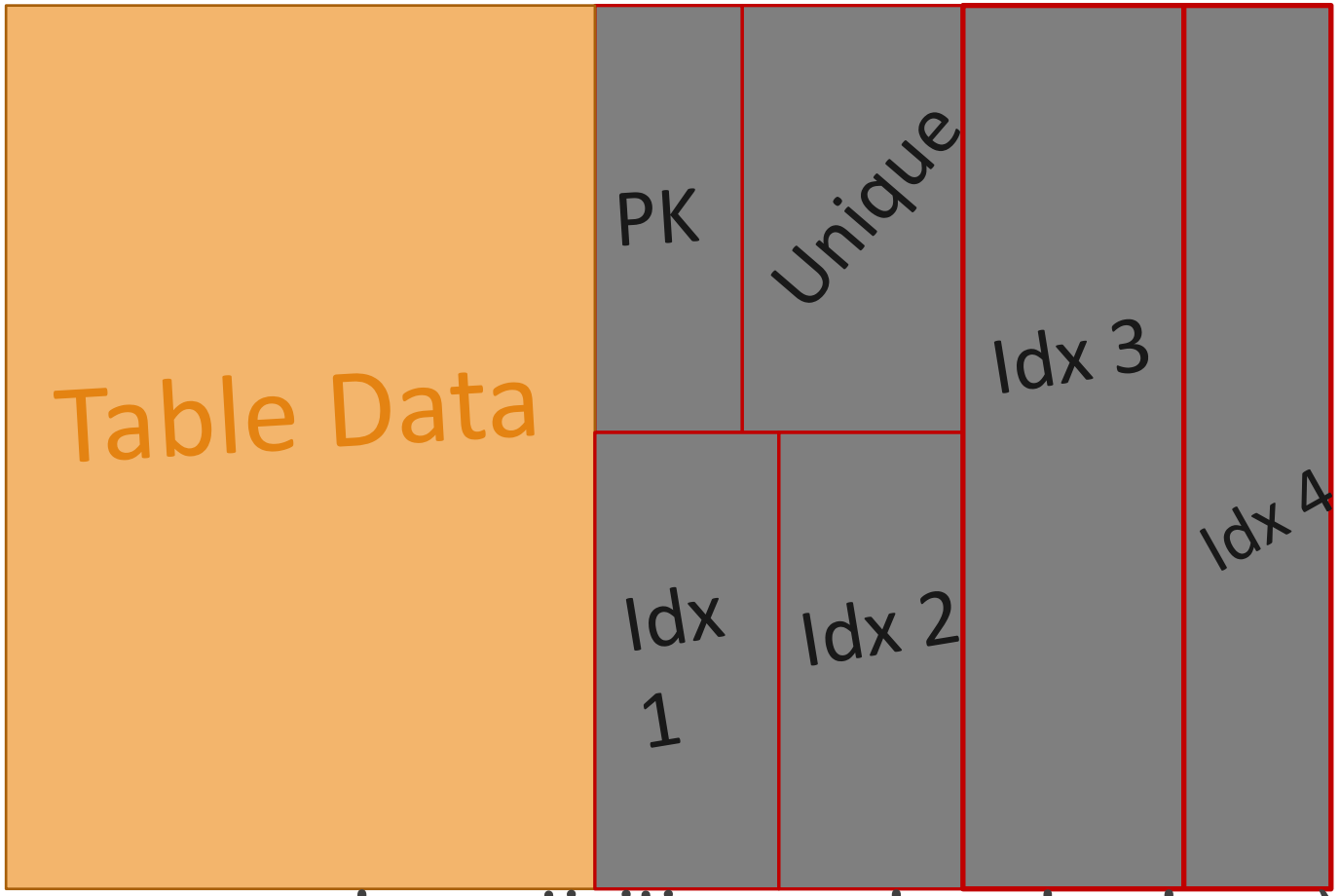
UPDATE



It requires moving things around in the tree, which is more painful work. The location is the same but the key has changed.

Storage

Additionally, indexes use a lot of storage, sometimes more than data! It has a huge impact on operations.



(By "operations" I mean regular activities such as backups)

You can also declare an index to be unique.

```
create unique index <index name>
on <table name>(<col1>, ... <coln>)
```

Enforces unique constraint like a constraint definition

If both are equivalent, then which one should we use?

Unique index

Unique constraint

Constraints

Logic

Rules

Constraints, without hesitation. They refer to design.

Indexes

Implementation

However (there is always a "however") there are some rare cases when some uniqueness (such as case-insensitive uniqueness in Oracle in a column in which data is in mixed case) cannot be enforced through declarative constraints but can be with the dirty trick of unique indexes.

For naming conventions, some people advocate prefixing an index name with something special.

```
IDX_MYTABLE_SOMECOL
```

If you have the choice, a suffix is probably a better option (with triggers too). It's possible to list all objects in a database, and, by sorting by name, suffixes allow to see all related objects grouped together.

```
MYTABLE_SOMECOL_IDX
```

In any case, follow standards that are in place.

FOLLOW NAMING STANDARDS

What matters isn't so much the rule than the fact that everybody is following it and that a name tells you immediately what an object is.

10.4 Performance

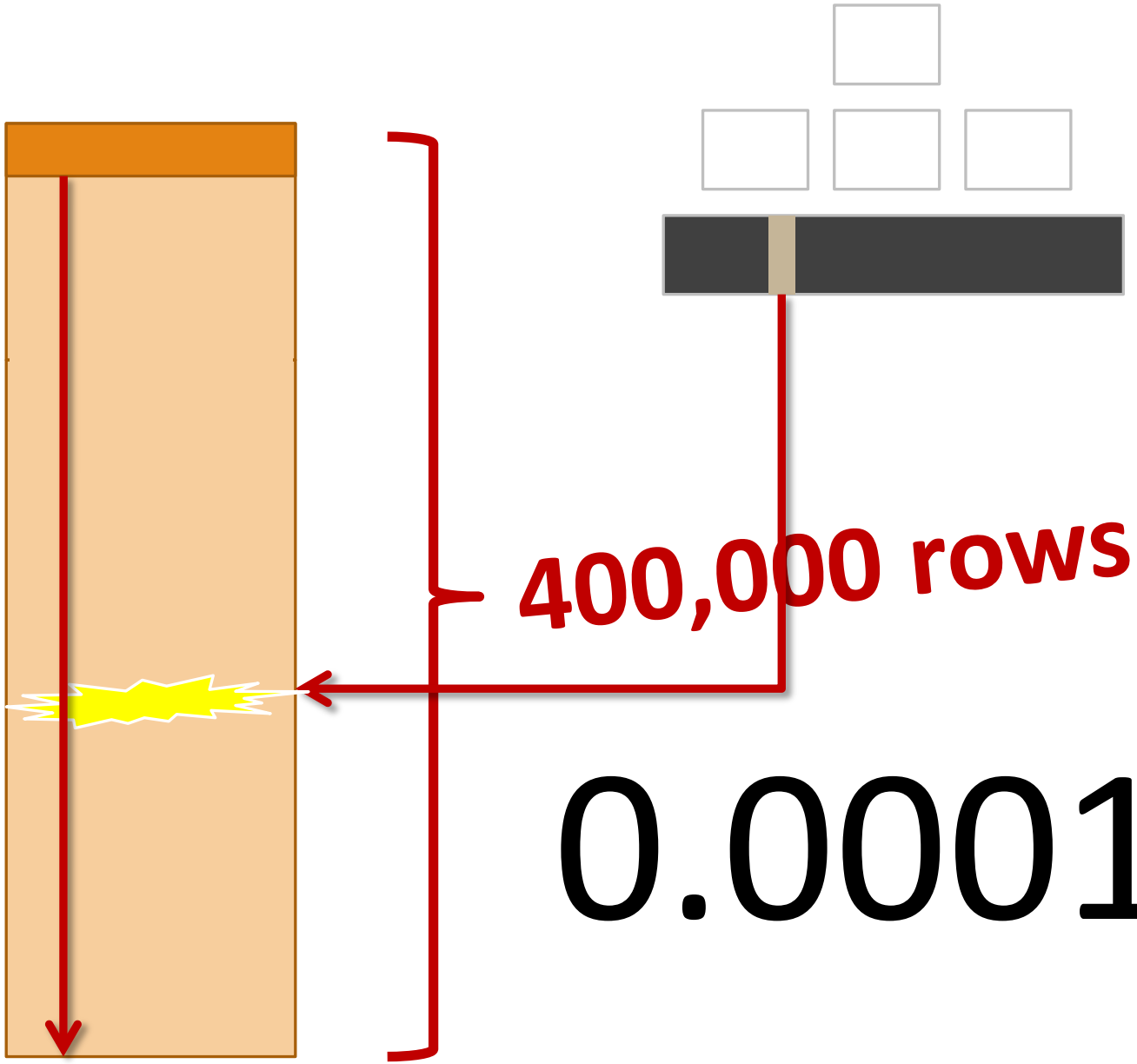
Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

0.4

Indexes massively improve performance. Scanning a half-million row table for a unique value may take a fraction of a second, but it will be instant with an index.

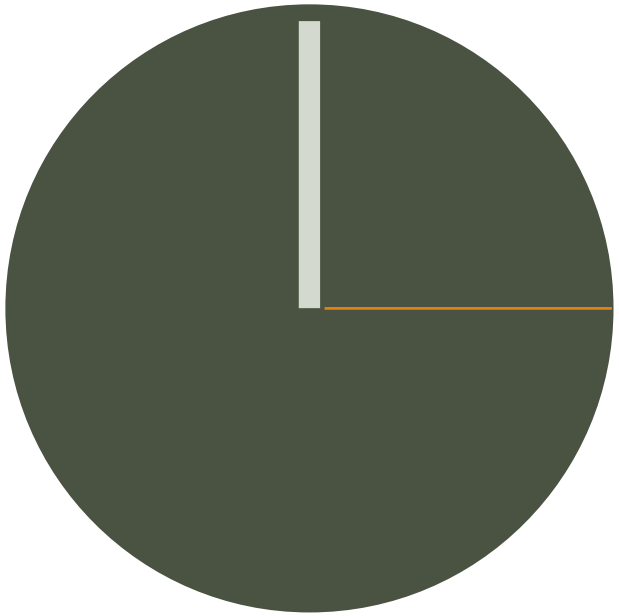


0.0001

An end-user may not notice much of a difference, because a sub-second response time is quite acceptable. It's when the same action is repeated a large number of times that it makes a difference.

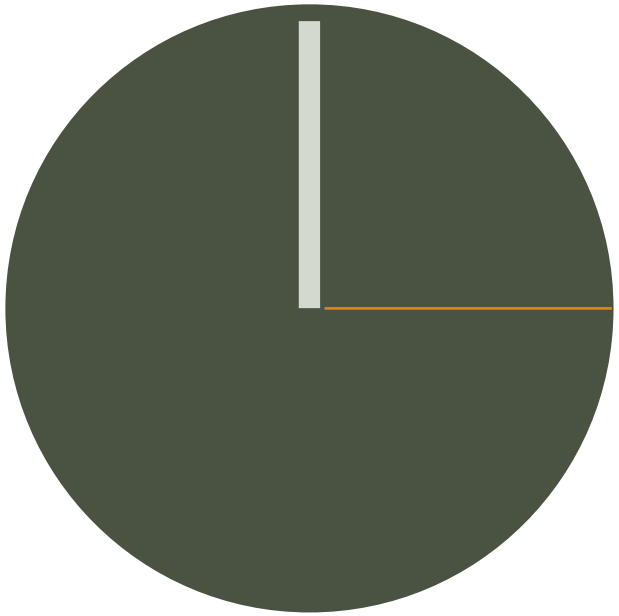
x100

0.01 40



1 2 3

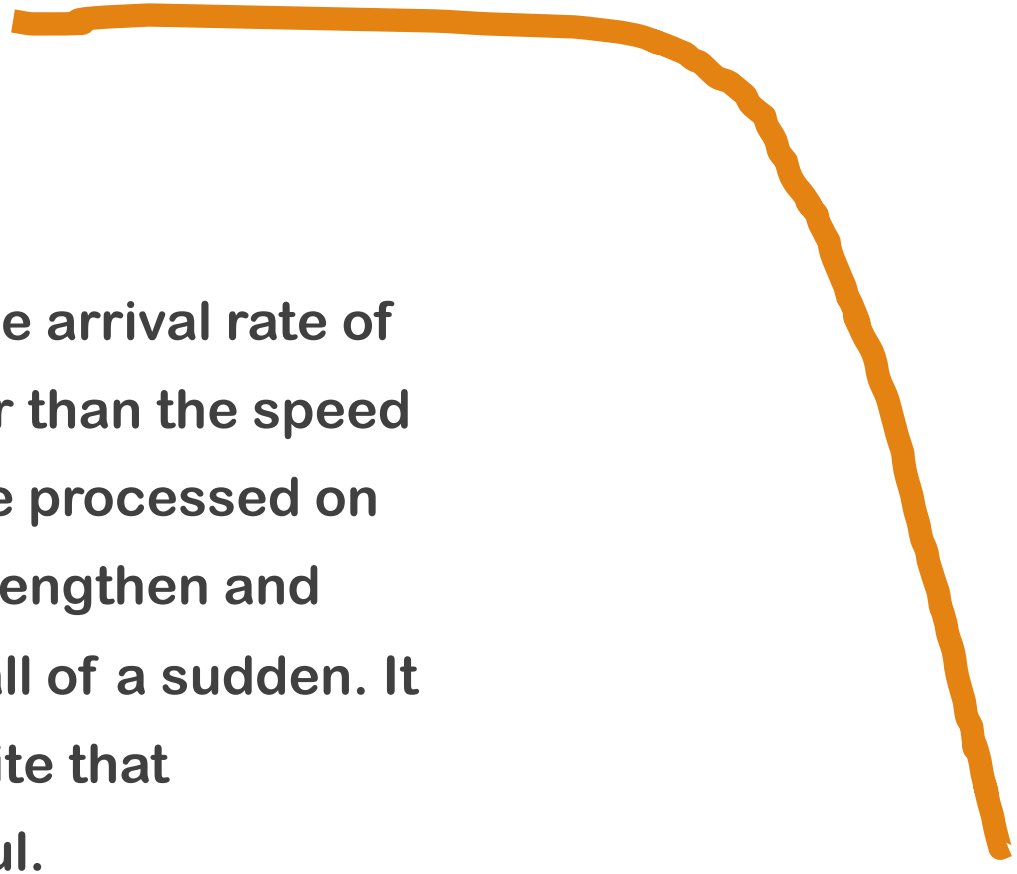
We are here right in the topic of scalability. Most computer systems are queueing systems. With few users, a mediocre response time is usually OK.



1 2 3 5 6 7

With a lot of queries, later queries will have to wait for earlier queries to be processed before they can be handled. For end-users, the wait time is part of the response time.

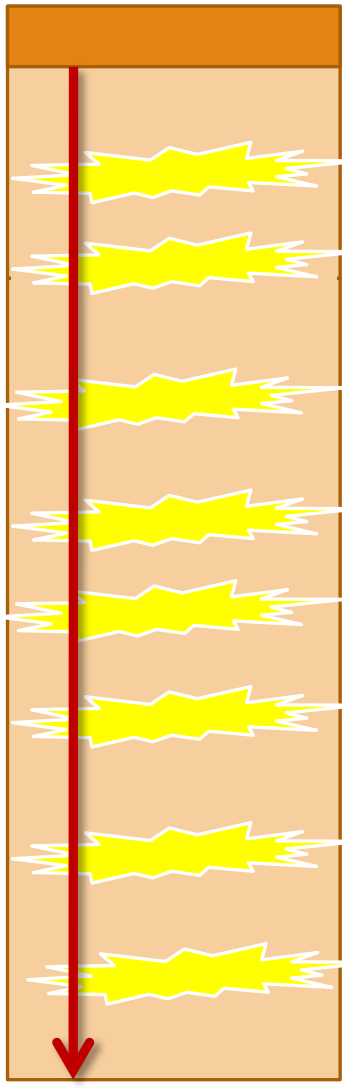
Performance



Problems start when the arrival rate of queries becomes faster than the speed at which queries can be processed on average. Then queues lengthen and performance crashes all of a sudden. It may happen on a website that becomes too successful.

The faster the better.

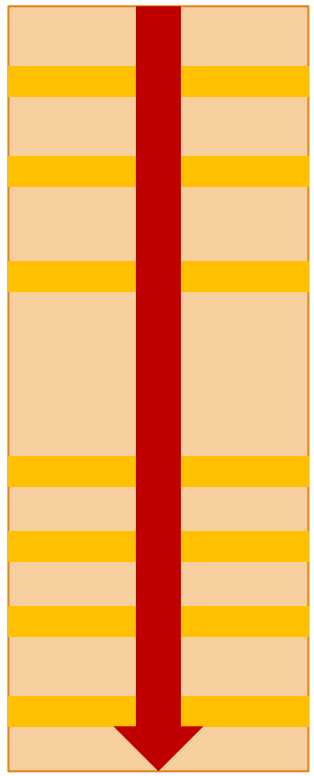
0.4

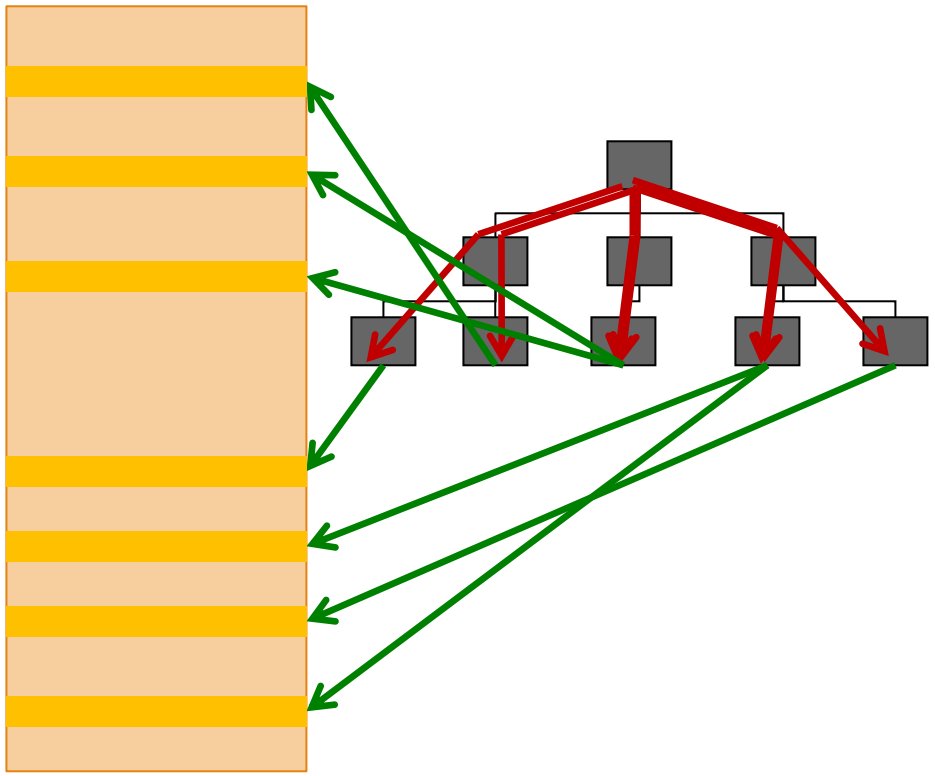


Does the super-performance of indexes mean that we should NEVER scan a table? That would be too easy! When we scan a table, the time to scan it is irrespective of how many rows we'll find to return: 1, 100, or several tens of thousands. An index search gives addresses of individual rows.

If we need to retrieve MANY rows, there will be a time when plainly scanning the table will be faster.

```
where column_name in
      (select …)
```

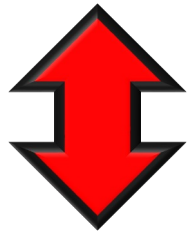




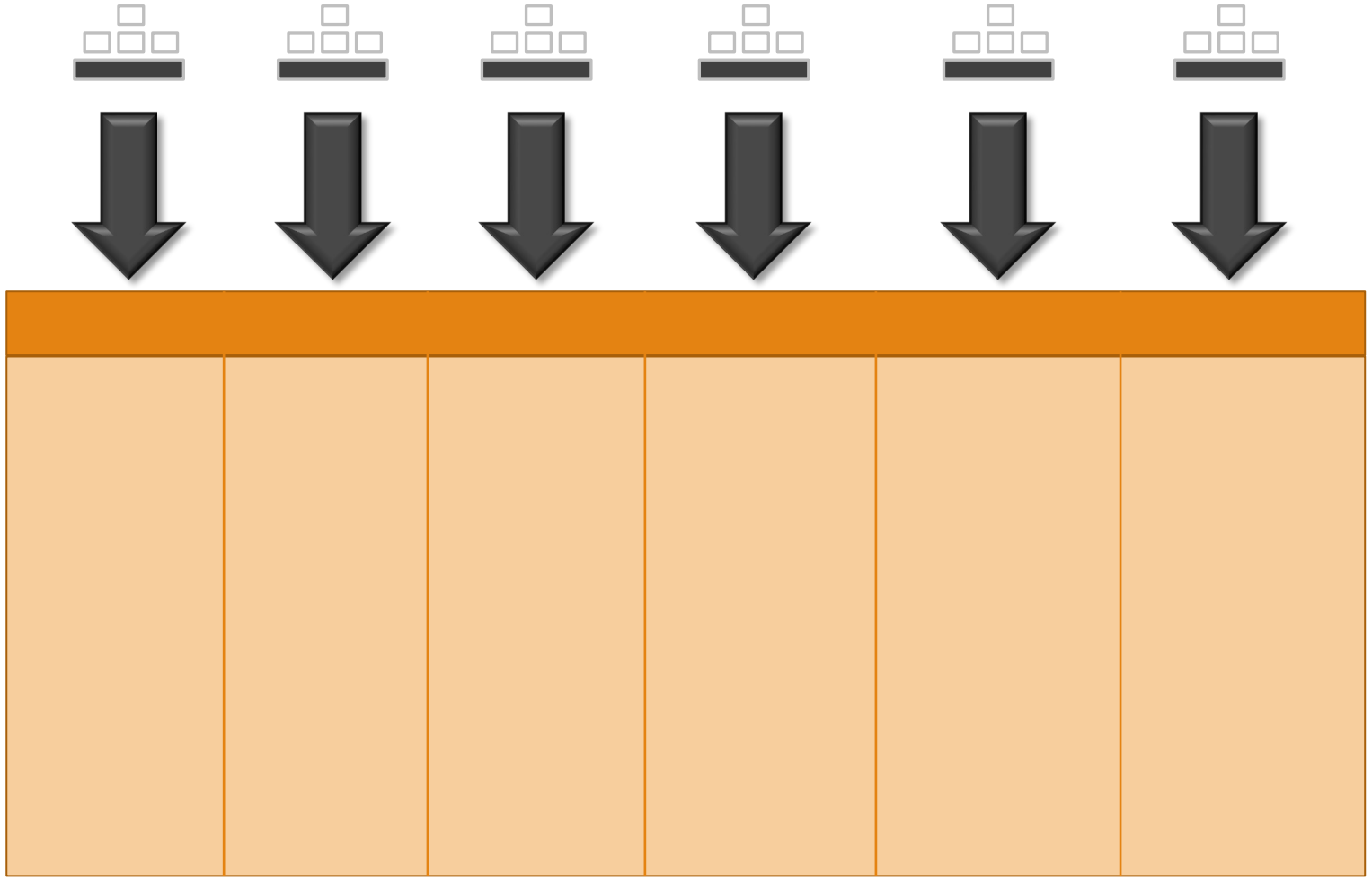
It arrives earlier than you might think.

What you are using and how you are doing it actually depends on how much data you are retrieving. What is known as "OLTP" (OnLine Transaction Processing) usually makes heavy use of indexes. By contrast, massive batch processes scan a lot, and it's how it should be.

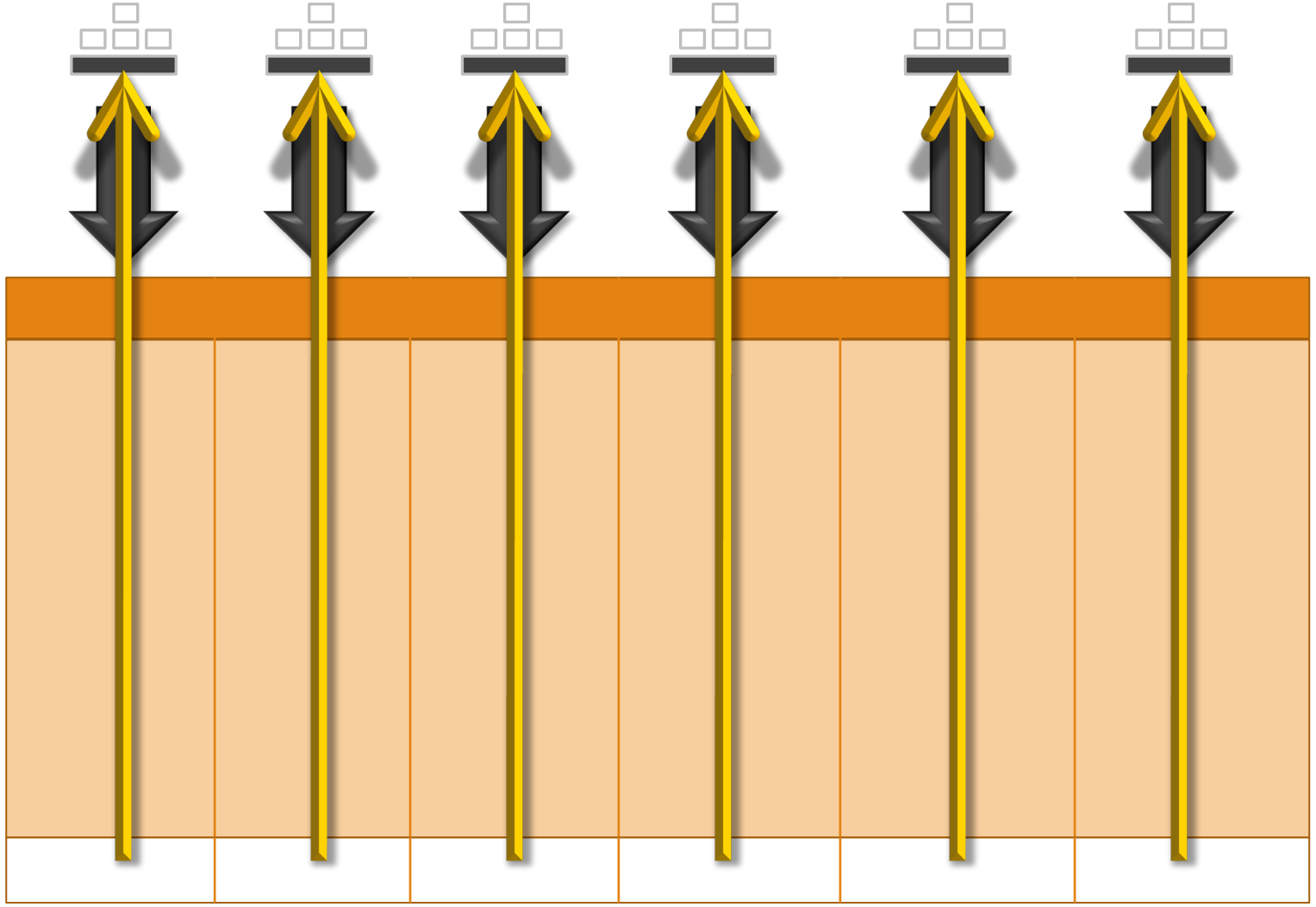
Right algorithm



Right volume

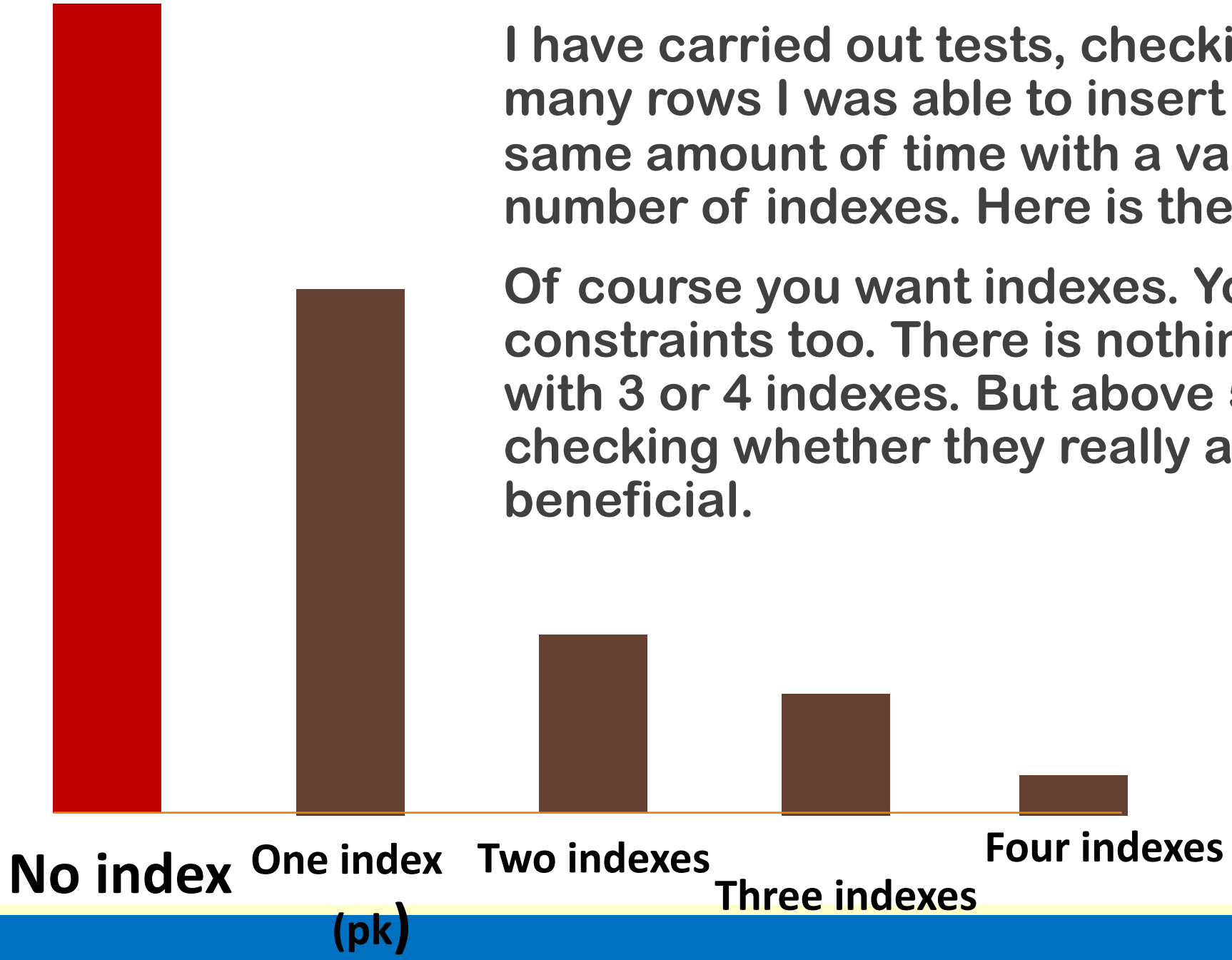


You sometimes find on forums people who are well-wishing but not very experienced who advise people who know hardly less than them to index all of their columns. Don't.



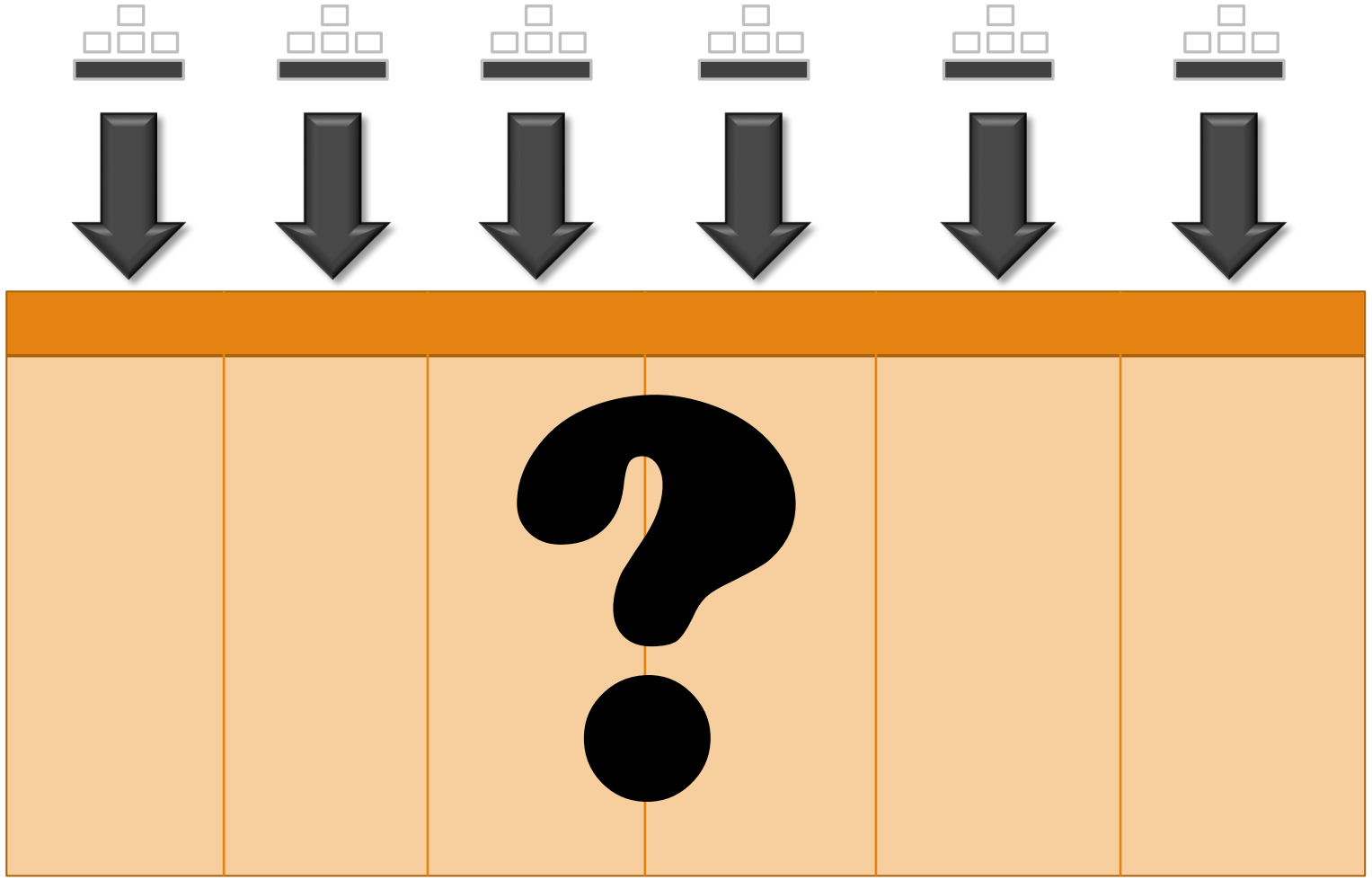
Indexing every column means, once again, that every INSERT will have to write not only into the table but into every index. Ouch.

You always have to compromise. Even if an index makes a search significantly faster, you have to put it in balance with the negative impact on inserts and deletes.



I have carried out tests, checking how many rows I was able to insert in the same amount of time with a variable number of indexes. Here is the result.

Of course you want indexes. You need constraints too. There is nothing wrong with 3 or 4 indexes. But above 5, I begin checking whether they really are beneficial.



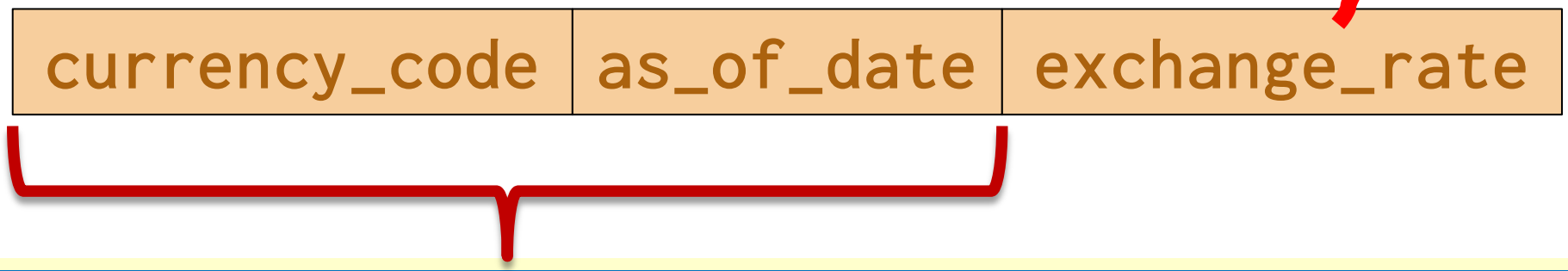
But if we don't want to index ALL columns, the big question becomes: which ones?

often used search criteria

There is little need to index a column that you don't use as a search criterion: indexes are primarily here to help you find values faster. In the same way, it doesn't make much sense to index a column especially for the yearly report if it must penalize your inserts all year long.

Typically, in table containing exchange rates, you would index by currency code and date, because one as little meaning without the other. Some people might want to add the exchange rate to the index to find it there without needing to access the table (we may talk more about storage tricks later if we have time)

```
where column_name = ...
```

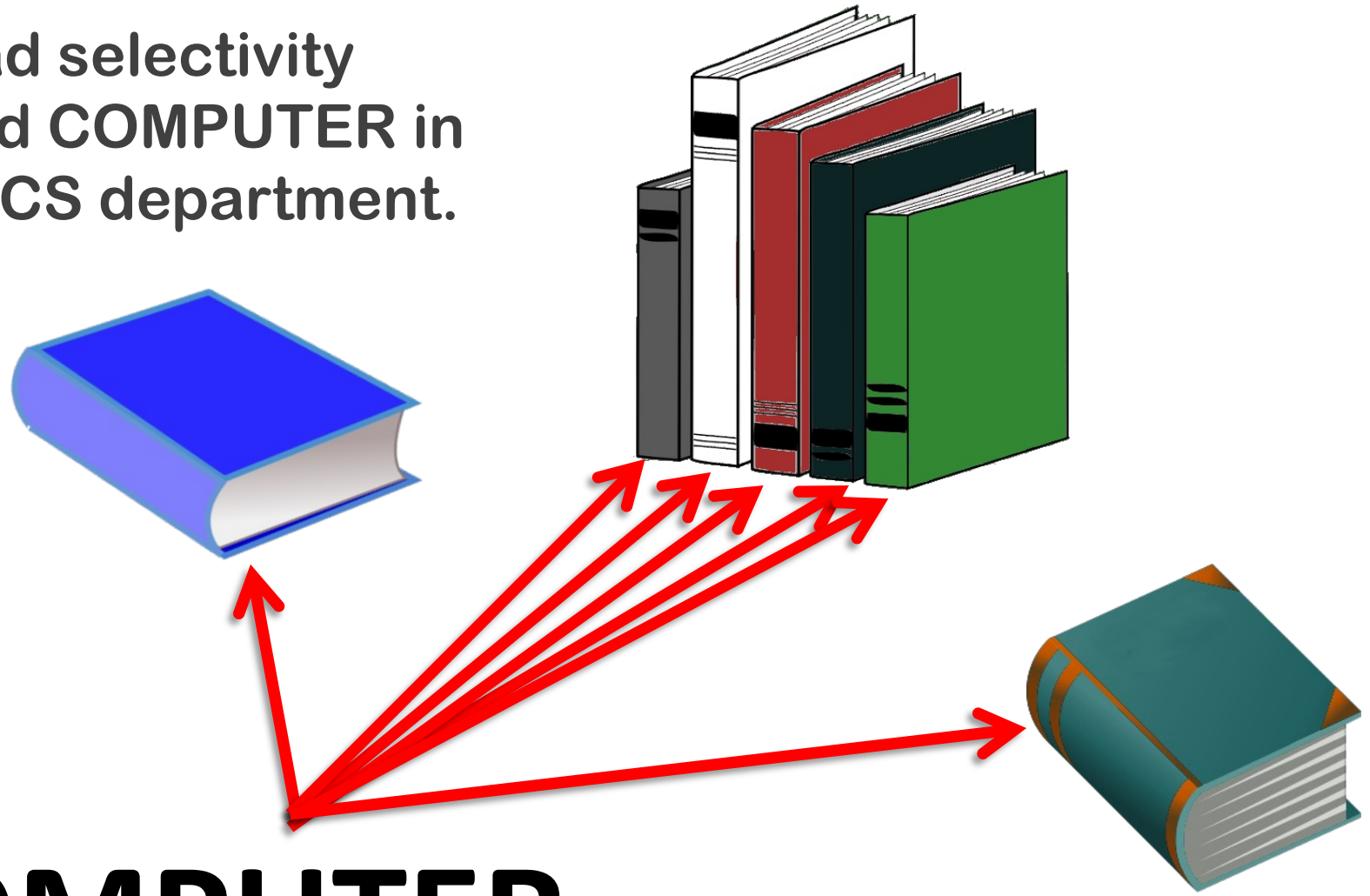


Another very important factor is that the column must be SELECTIVE, that means that the values it contains are rare and correspond to very few rows. Unique columns and PK are extremely selective by definition. Other columns are a mixed bag.

selective

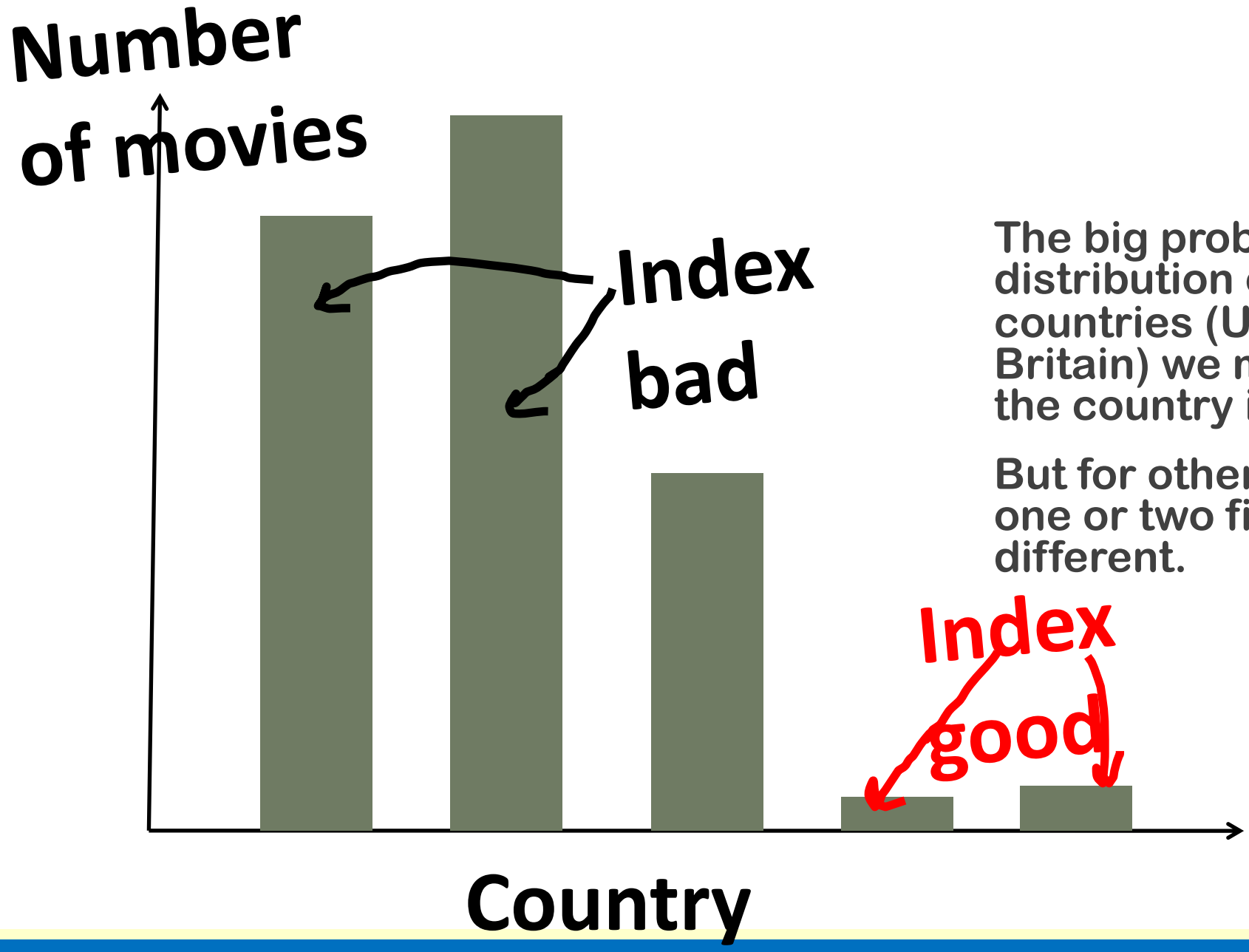
rare

An example of bad selectivity would be the word COMPUTER in the library of the CS department.



COMPUTER

It's probably in every book.



The big problem is with the distribution of values. For some countries (US, Japan, China, India, Britain) we may have many films, and the country isn't selective.

But for other countries we may have one or two films, and then it will be different.

10.5 Execution plan

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

What is the DBMS actually DOING?

If there is an index on a column such as COUNTRY, will the DBMS use it? Not necessarily. The optimizer may decide not to use an index. Is there a way to know if it will? Good news:

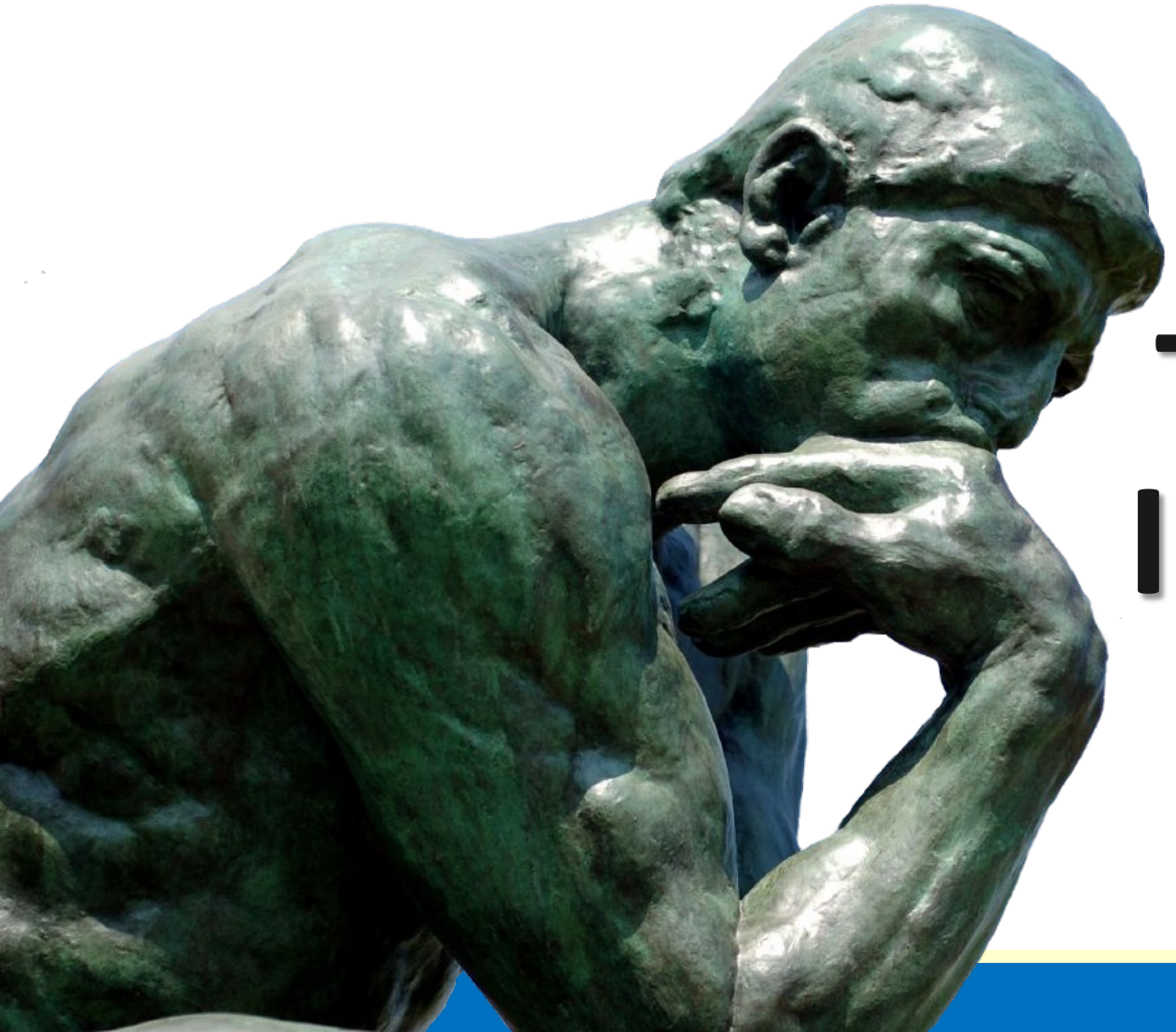
yes.

All DBMS products implement (with slight syntax differences, it won't surprise you) a way to display what is known as the "execution plan", what the optimizer would choose to do to run a query.

```
explain <select statement>
```


```
set showplan_all on
```

execution plan

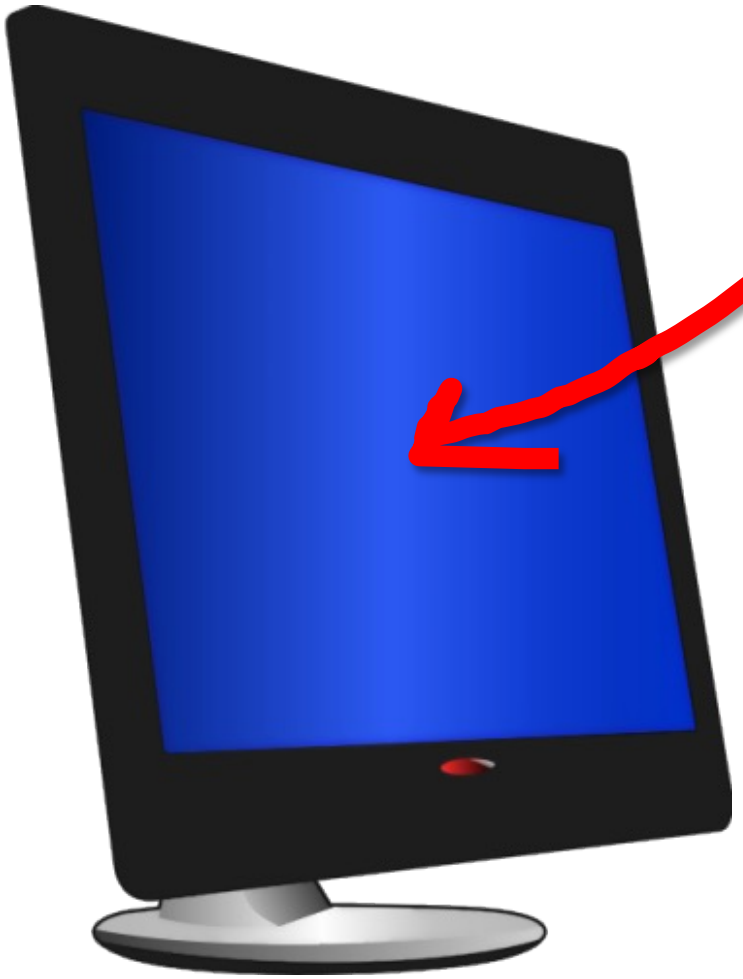


Tables
Indexes

In the execution plan, you see tables and indexes that are accessed.

Flickr: Brian Hillegas

explain



plan_table

Depending on the DBMS, the "explain" command will display the plan directly or populate a table that must be queried.

Many tools can also display graphically an execution plan on demand.

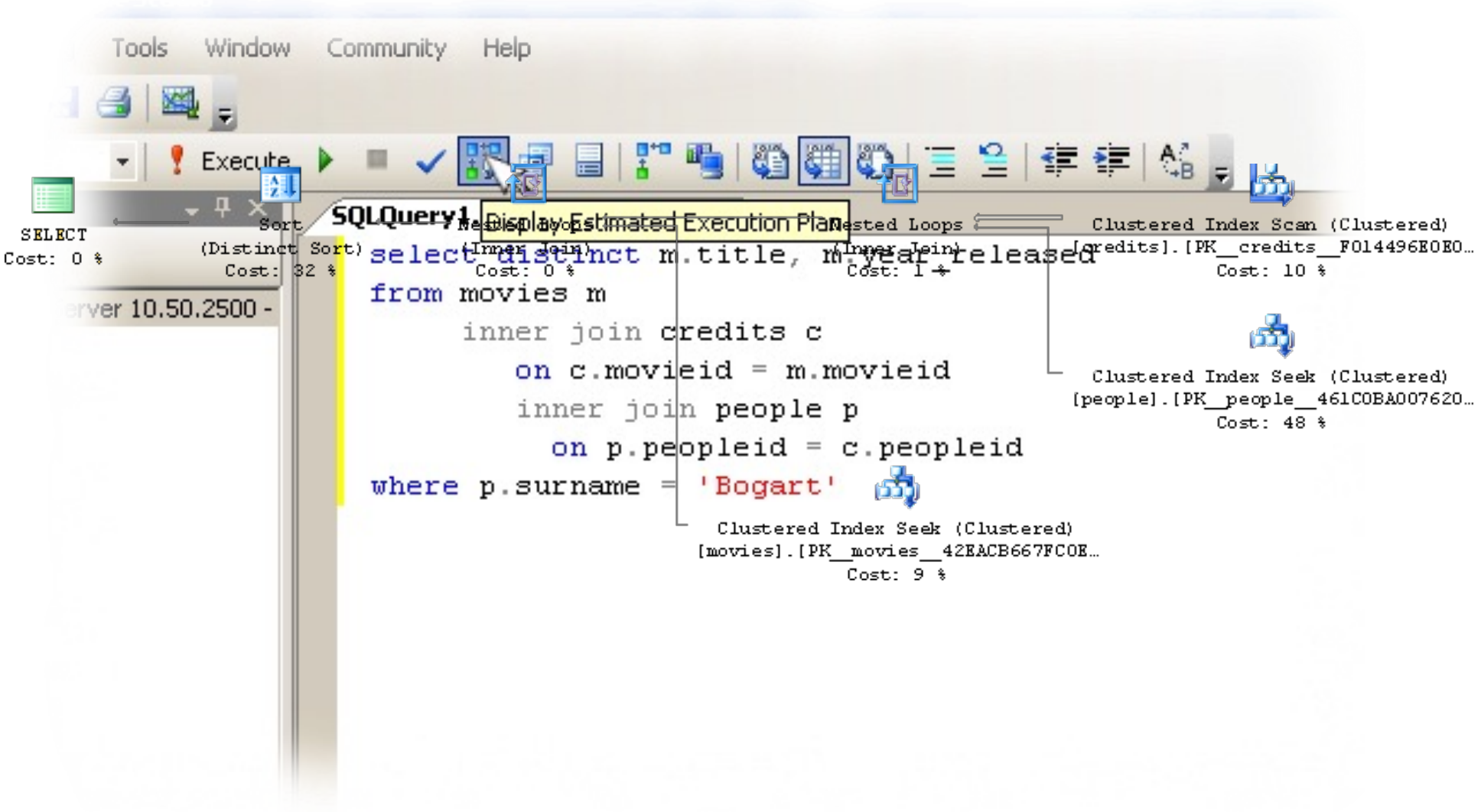
Graphical Environments

SQL Server Management Studio

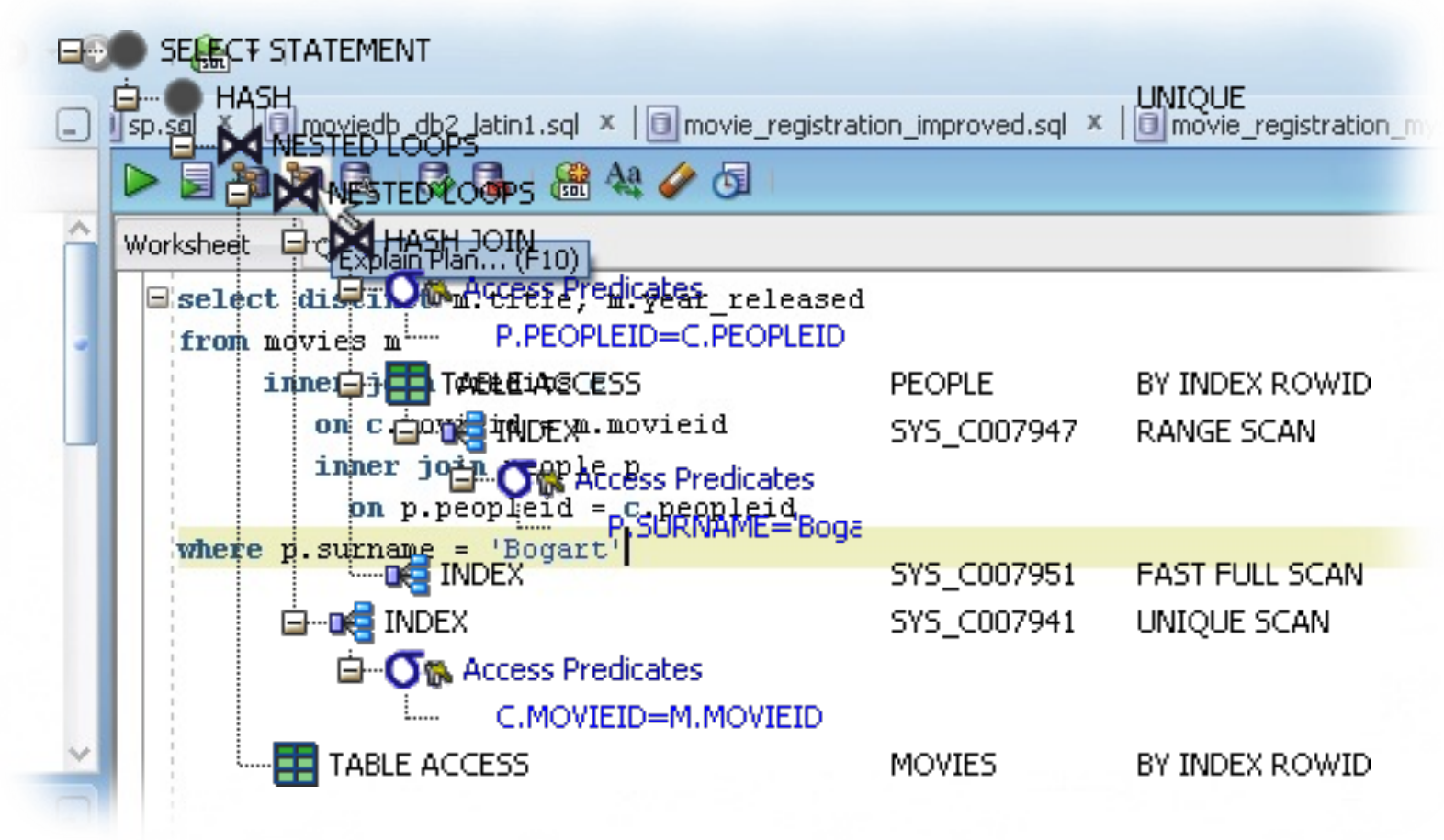
SQL Developer

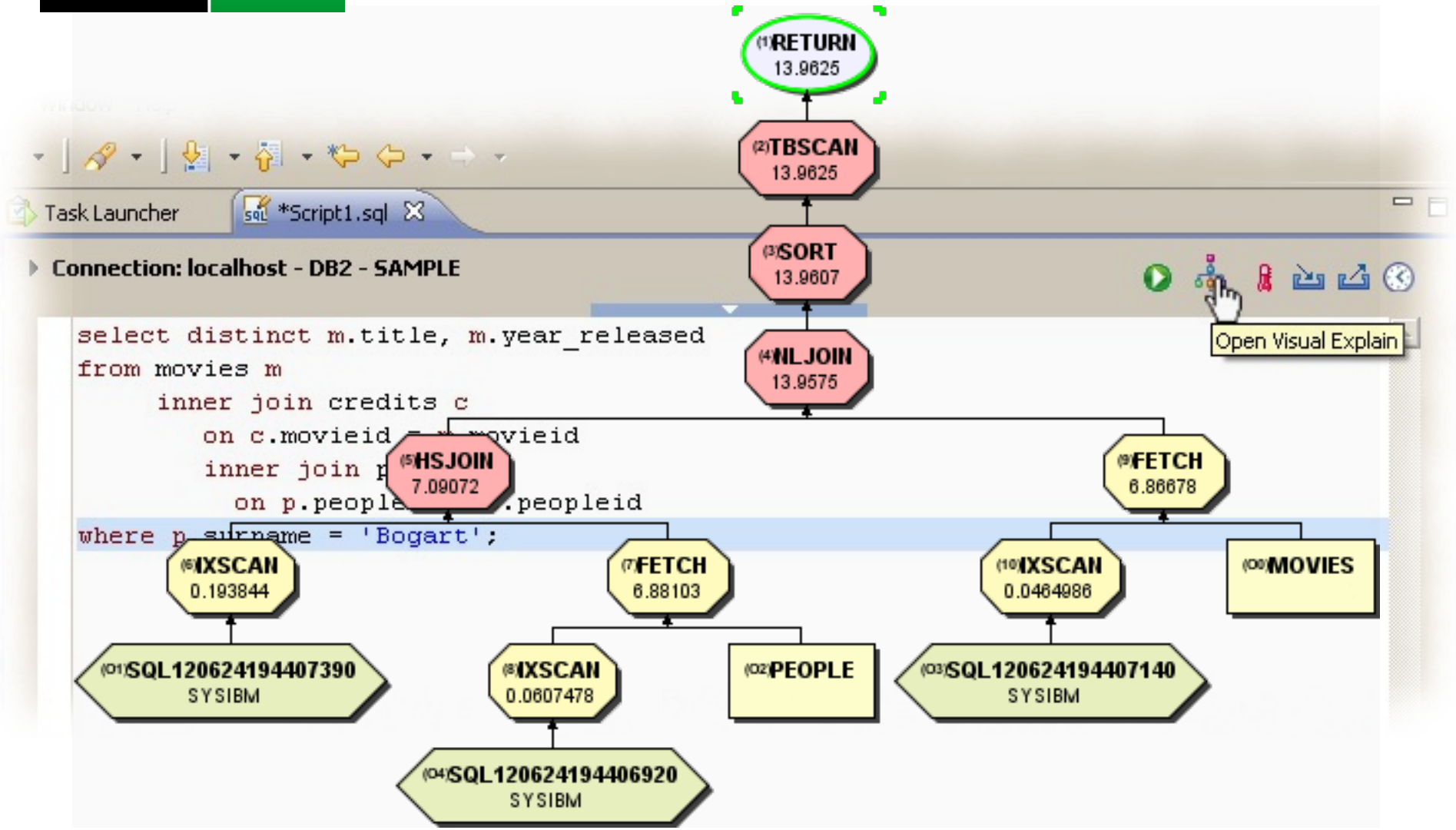
IBM Data Studio etc.

Microsoft® SQL Server®



ORACLE


SELECT STATEMENT
HASH
UNIQUE
NESTED LOOPS
NESTED LOOPS
HASH JOIN
Worksheet
Explain Plan... (F10)
Access Predicates
P.PEOPLEID=C.PEOPLEID
TABLE ACCESS
PEOPLE
BY INDEX ROWID
INDEX
SYS_C007947
RANGE SCAN
Access Predicates
P.SURNAME='Boga
INDEX
SYS_C007951
FAST FULL SCAN
INDEX
SYS_C007941
UNIQUE SCAN
Access Predicates
C.MOVIEID=M.MOVIEID
TABLE ACCESS
MOVIES
BY INDEX ROWID
select m.title, m.year_released
from movies m
on p.peopleid = c.peopleid
where p.surname = 'Bogart'


(1)RETURN
13.9625
(2)TBSCAN
13.9625
(3)SORT
13.9607
(4)NLJOIN
13.9575
(5)HSJOIN
7.09072
(9)FETCH
6.86678
(6)IXSCAN
0.193844
(7)FETCH
6.88103
(10)IXSCAN
0.0464986
(00)MOVIES
(01)SQL120624194407390
SYSIBM
(08)IXSCAN
0.0607478
(02)PEOPLE
(03)SQL120624194407140
SYSIBM
(04)SQL120624194406920
SYSIBM
Task Launcher
*Script1.sql
Connection: localhost - DB2 - SAMPLE
Open Visual Explain
select distinct m.title, m.year_released
from movies m
inner join credits c
on c.movieid
inner join p
on p.people .peopleid
where p.surname = 'Bogart';

```
mysql> explain select distinct m.title, m.year_released
    -> from movies m
    ->      inner join credits c
    ->         on c.movieid = m.movieid
    ->         inner join people p
    ->           on p.peopleid = c.peopleid
    -> where p.surname = 'Bogart';
+----+-------------+-------+----------+--------------------+---------------------------------+
| id | select_type | table | key      | ref                | Extra                           |
+----+-------------+-------+----------+--------------------+---------------------------------+
|  1 | SIMPLE      | p     | surname  | const              | Using where; Using index        |
|  1 | SIMPLE      | c     | peopleid | movies.p.peopleid | Using index                     |
|  1 | SIMPLE      | m     | PRIMARY  | movies.c.movieid  |                                 |
+----+-------------+-------+----------+--------------------+---------------------------------+
3 rows in set (0.00 sec)

mysql>
```


```
movies=# explain select distinct m.title, m.year_released
movies-# from movies m
movies-#      inner join credits c
movies-#         on c.movieid = m.movieid
movies-#         inner join people p
movies-#           on p.peopleid = c.peopleid
movies-# where p.surname = 'Bogart';
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 HashAggregate  (cost=5.29..5.30 rows=1 width=222)
   ->  Nested Loop  (cost=2.16..5.28 rows=1 width=222)
         ->  Hash Join  (cost=2.16..4.51 rows=1 width=4)
               Hash Cond: (c.peopleid = p.peopleid)
               ->  Seq Scan on credits c  (cost=0.00..1.97 rows=97 width=8)
               ->  Hash  (cost=2.15..2.15 rows=1 width=4)
                     ->  Seq Scan on people p  (cost=0.00..2.15 rows=1 width=4)
                           Filter: ((surname)::text = 'Bogart'::text)
         ->  Index Scan using movies_pkey on movies m  (cost=0.00..0.76 rows=1 width=226)
               Index Cond: (movieid = c.movieid)
(10 rows)

movies=#
```


```
sqlite> explain query plan
   ...> select distinct m.title, m.year_released
   ...> from movies m
   ...>      inner join credits c
   ...>         on c.movieid = m.movieid
   ...>         inner join people p
   ...>            on p.peopleid = c.peopleid
   ...> where p.surname = 'Bogart';
0|0|2|SEARCH TABLE people AS p USING COVERING INDEX sqlite_autoindex_people_1 (surname=?) (~10)
0|1|1|SEARCH TABLE credits AS c USING AUTOMATIC COVERING INDEX (peopleid=?) (~7 rows)
0|2|0|SEARCH TABLE movies AS m USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
0|0|0|USE TEMP B-TREE FOR DISTINCT
sqlite>
```

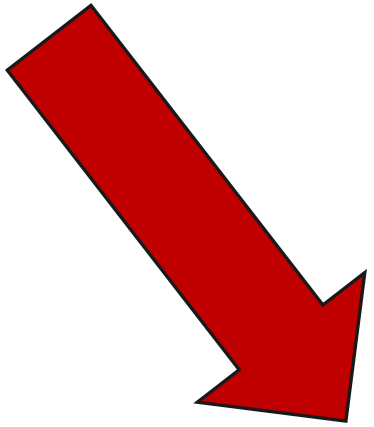
It's worth noting that the same query on the same tables with the same indexes and the same data in the tables may very well result in different execution plans with different DBMS products. Internal algorithms are different, internal data storage is different, some products may be better than others at processing data in a particular way, and of course optimizers are different and may choose different courses.

Beginners often assume that there are "good" execution plans (that only use indexes) and "bad" ones (which scan tables). In fact, it's impossible in most cases to say, by reading two different plans, which one is fastest.

execution plans

The main benefit of execution plans it to check whether the optimizer is more or less doing what you thought it would do.

```
explain
```



In particular whether an index is used. The optimizer may choose not to use it. It may also be unable to use it.

Index used?

```
filmdb=# explain select * from movies where movieid =100;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using movies_pkey on movies  (cost=0.29..8.30 rows=1 width=31)
   Index Cond: (movieid = 100)
(2 rows)
```

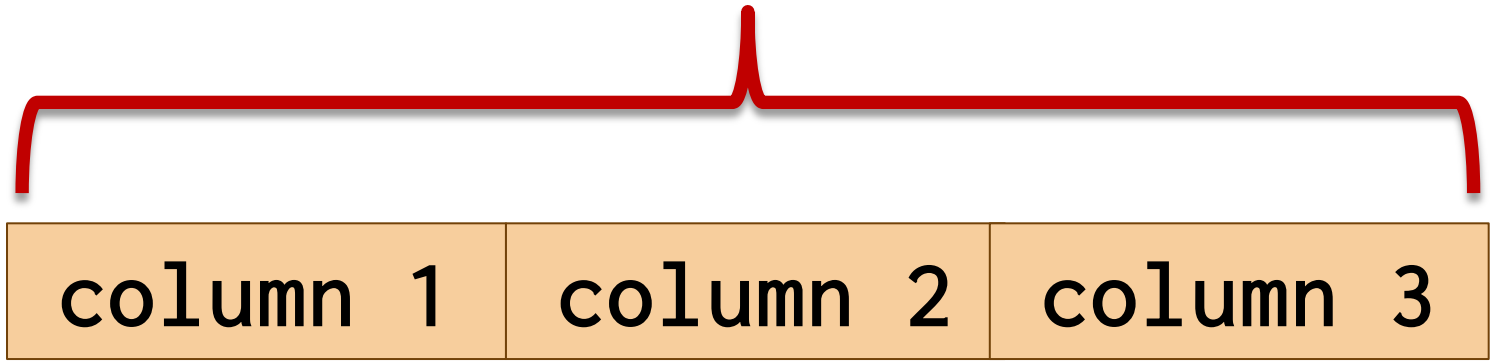
```
filmdb=# explain select * from movies where movieid >100;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on movies  (cost=0.00..187.05 rows=9105 width=31)
   Filter: (movieid > 100)
(2 rows)
```

10.6 Use Index or Not

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

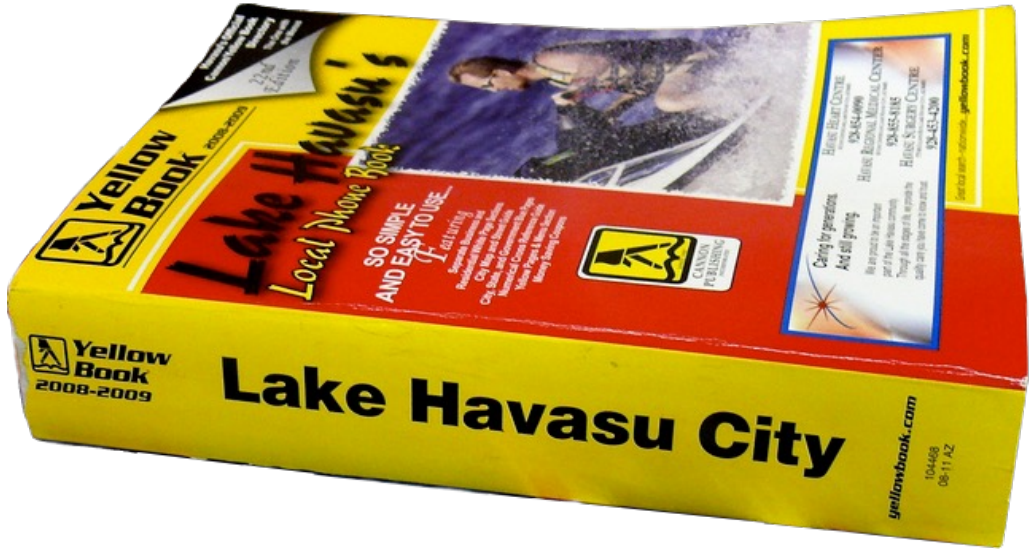
Most contents are from Stéphane Faroult's slides



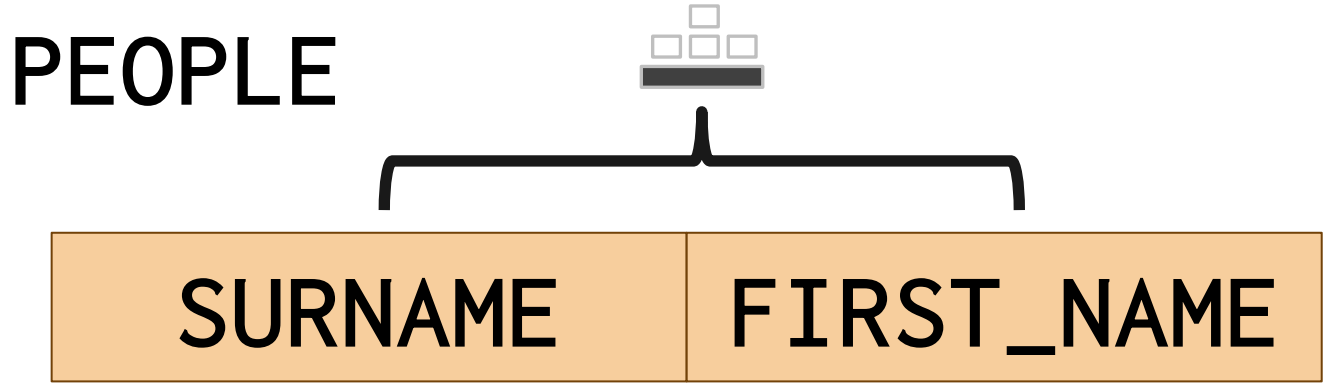
The impossibility of using an index (or of using it as intended) may come from several reasons. One is with composite indexes, the key of which is concatenated values from several columns. The index can only be used if the lead column(s) appear in the WHERE condition of the query.

The problem is the same as looking up for someone's number in a phone book when you don't know the surname. If you had the surname and address but not the first name, you could do it.

surname	firstname	column 3



Flickr: Larry Page




```
explain select * from people
 where surname = '...'
   and first_name = '...'
```

index can be used

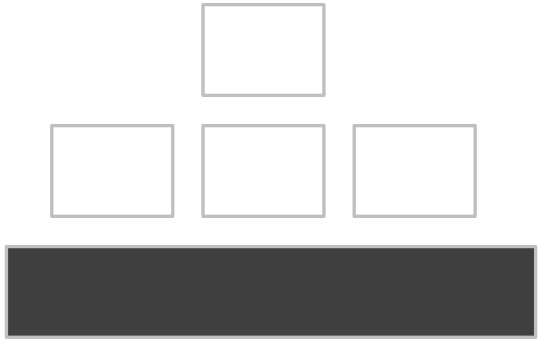
```
 where first_name = '...'
```

index can not be used

`like '%something%'`

Exactly the same problem happens when you are using a LIKE expression that STARTS with a %, or the SUBSTR() function. Without the leading part, no way you can walk the tree that takes you to row addresses.

function()



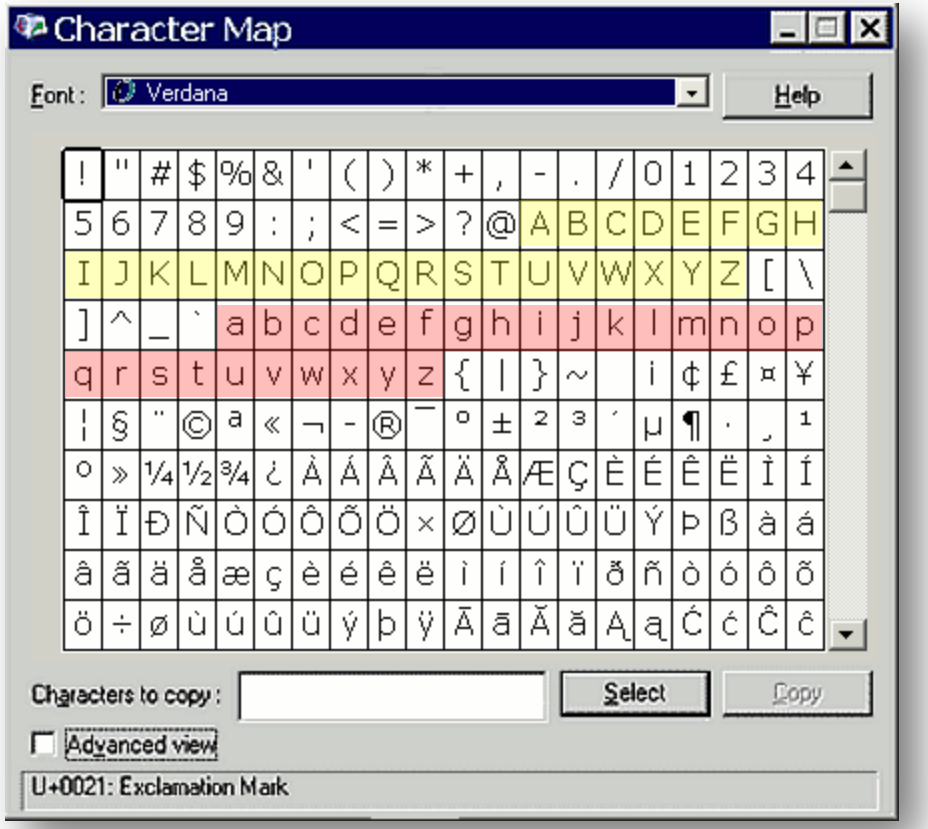
Actually, the problem is more general than that. Suppose that you have a tree built upon the values that you can find in column C. If you say that f(C) is equal to something, there is no way to find the corresponding value for C in the tree.

Case-sensitive DBMS:

```
where upper(surname) = upper('some input')
```

A typical, and common, problem is to perform, with a case-sensitive DBMS a case insentive search in a column when data is in mixed case.

A tree is based on the ordering of values. It's how values compare to the values in the node that you are visiting which tells you which subtree you should visit next.



The problem is that the internal codes of letters consider that "a" is greater than "Z" (and don't talk about accented letters)

```
where upper(surname) = 'MARVIN'
```

Where to search?

"smaller" values

MILES

O'brien

Stewart

marvin

wayne

"bigger" values

If you imagine that you have a binary tree, find 'Stewart' at the root, and look for something equal to MARVIN when set to uppercase without knowing in which case that something is written, you are toast.

It could actually be anywhere in the tree.

```
where upper(surname) = 'MARVIN'
```

Same story with functions that extract date parts.
Use them in a WHERE clause, your index is dead.
You should always express conditions on dates as range conditions.

```
where extract(month from date_column) = 6
  and extract(year from date_column) =
      extract(year from current_date)
```

Same story again with implicit conversions.

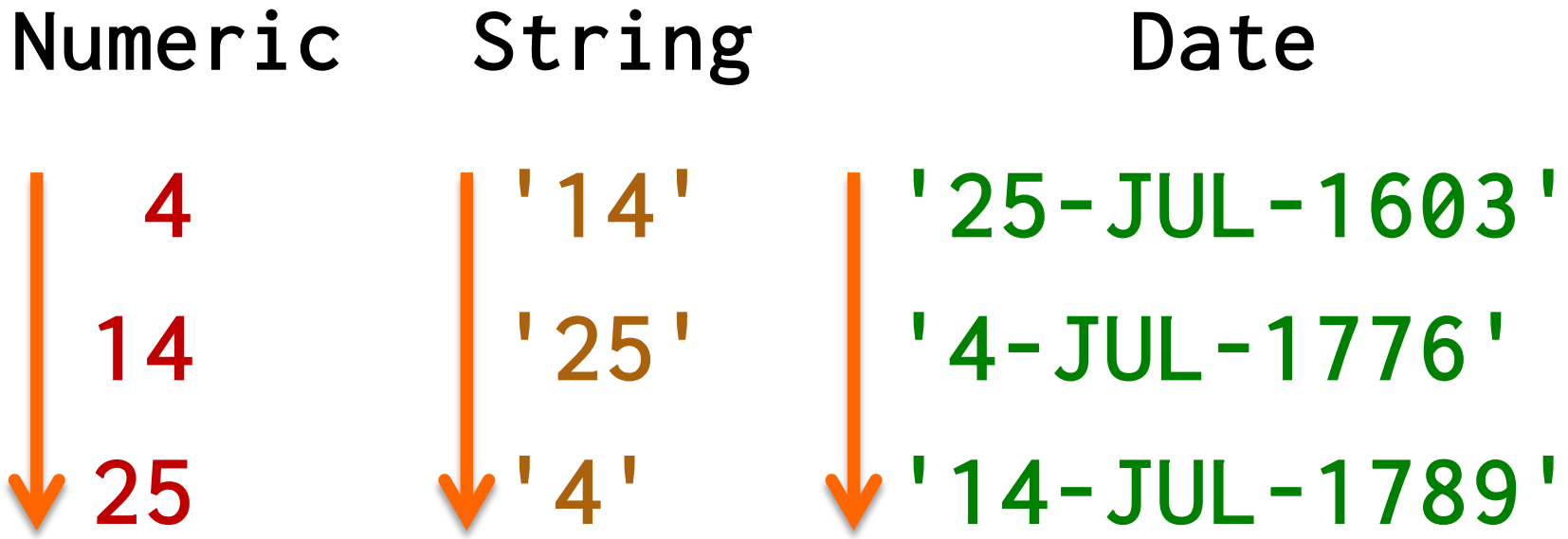
If they are performed
in the wrong direction,
wave farewell to the index.

```
and varchar_code = 12345678
```

```
select cast('12345678' as varbinary(max)) as string_12345678,
       cast(12345678 as varbinary(max)) as number_12345678;
```

string_12345678	number_12345678
0x31323334353637 38	0x00BC614E

Ooops ...



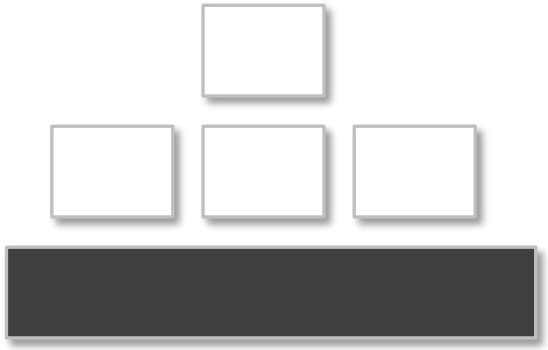
The reason is simple. Index search is based on ordering (inside the tree) and ordering is different with different datatypes. Convert datatypes, you break the ordering.

```
insert into table_name(column_name)
values(upper(<input>))
```

Some of these issues can be taken care of by only storing appropriate data in your tables, for instance not storing mixed case.

```
select *
from people
where soundex(surname) =
soundex('Stuart')
```

In some other cases, though, the search MAY require applying a function to the indexed column.



Is there a way to apply a function to a column and yet benefit of the quick access provided by an index? Some products actually allow indexing an expression (more on that coming soon). Otherwise the answer is yes, by cheating.

people

peopleid	first_name	surname	born	died	surname_soundex
1	James	Stewart	1908	1997	S363
2	Humphrey	Bogart	1899	1957	B263

What you can do for instance is add a column that stores the soundex. This happily violates the rules of good normalization (... it depends on another column that is a part of a key), but then you can index it and apply the search to this new redundant column.

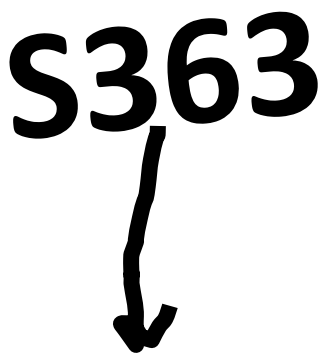
Trigger – before Insert / for each row

people

peopleid	first_name	surname	born	died	surname_soundex
1	James	Stewart	1908	1997	S363
2	Humphrey	Bogart	1899	1957	B263

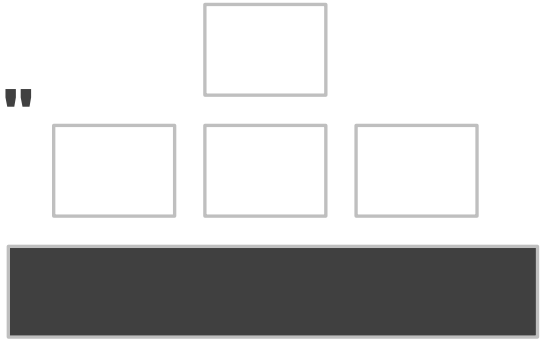
If you have full control of the program that inserts data into the table you can modify it so as to insert soundex(SURNAME) at the same time as you insert SURNAME. If not, there is (at the cost of far slower uploads) the option of a before insert/for each row trigger that does it on the fly.

```
select *
from people
where surname_soundex = soundex('Stuart')
```



Having the column and having indexed it, you no longer need to apply any function to the searched column, and everything is fine.

Most products actually allow you to do something cleaner by indexing an expression (sometimes called "generated" or "virtual" column), which can be the result of a function.

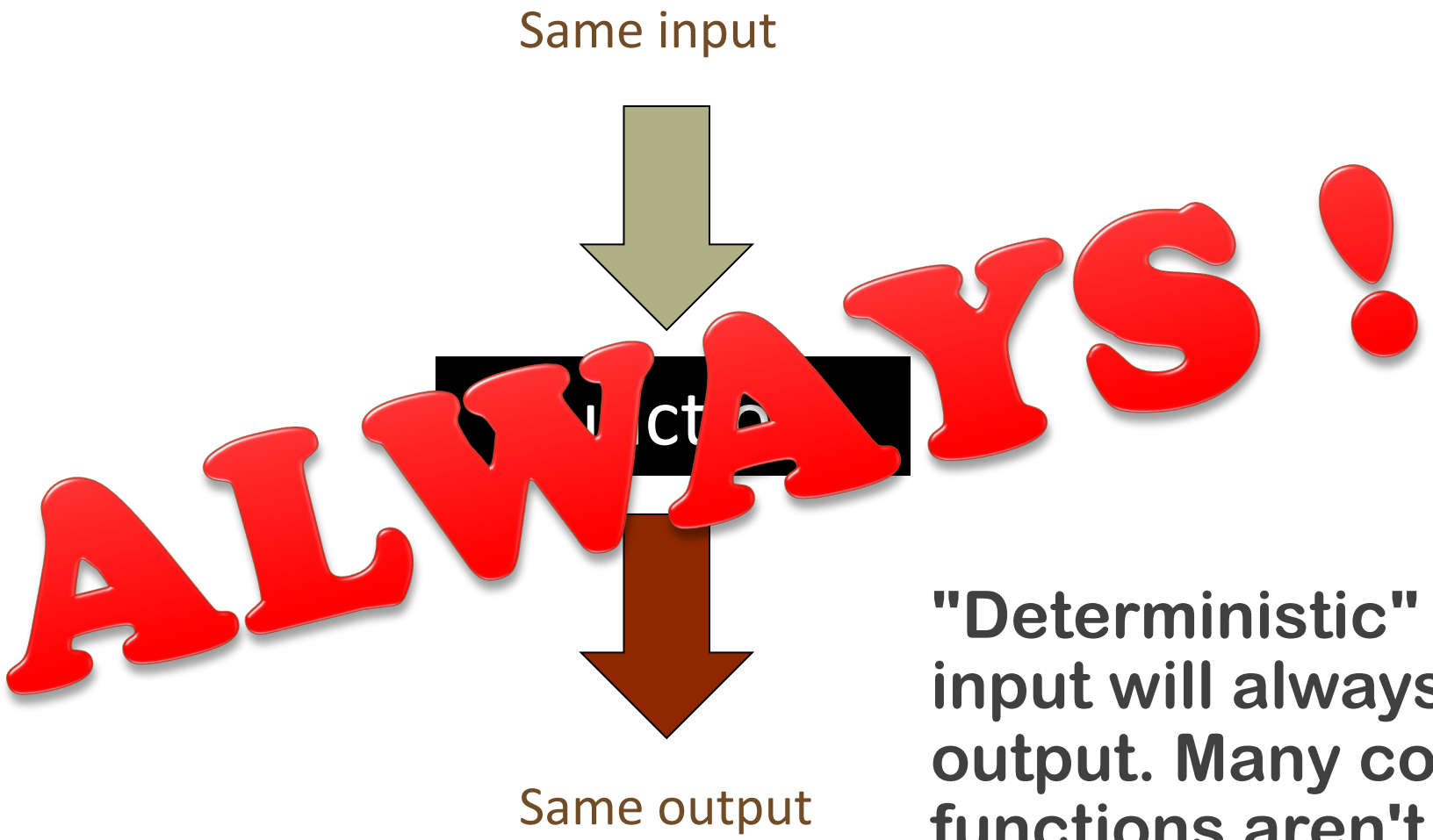


It only works if the expression or function is

Deterministic

$f()$

transformation



"Deterministic" means that the same input will always generate the same output. Many commonly used functions aren't deterministic because their result is affected by database settings (localization settings most often).

```
datename(month, '1970-01-01')
'Januar''Enero' 'Janeiro'
'January' '1月' 'Janvier'
```

This SQL Server function isn't deterministic, because if the DBA changes the language by default of the database, what will be returned will be different. You can imagine the scenario: language is English, you create an index, plenty of "January" stored in the index, the DBA switches the language to Spanish and you search the index for "Enero" ... SQL Server prevents you from indexing this function.

You might think that functions that return numbers are safer. Conventions for day numbering actually vary with countries.

upper(column_name)

Fortunately, there are still many functions that truly are deterministic! If your DBMS allows it, you can index UPPER(...) without any problem. Same story with SOUNDEX(...). When your statement will be analyzed the SQL engine will be able to notice your using the function and will use the index.