Static Indexes and Bitmaps

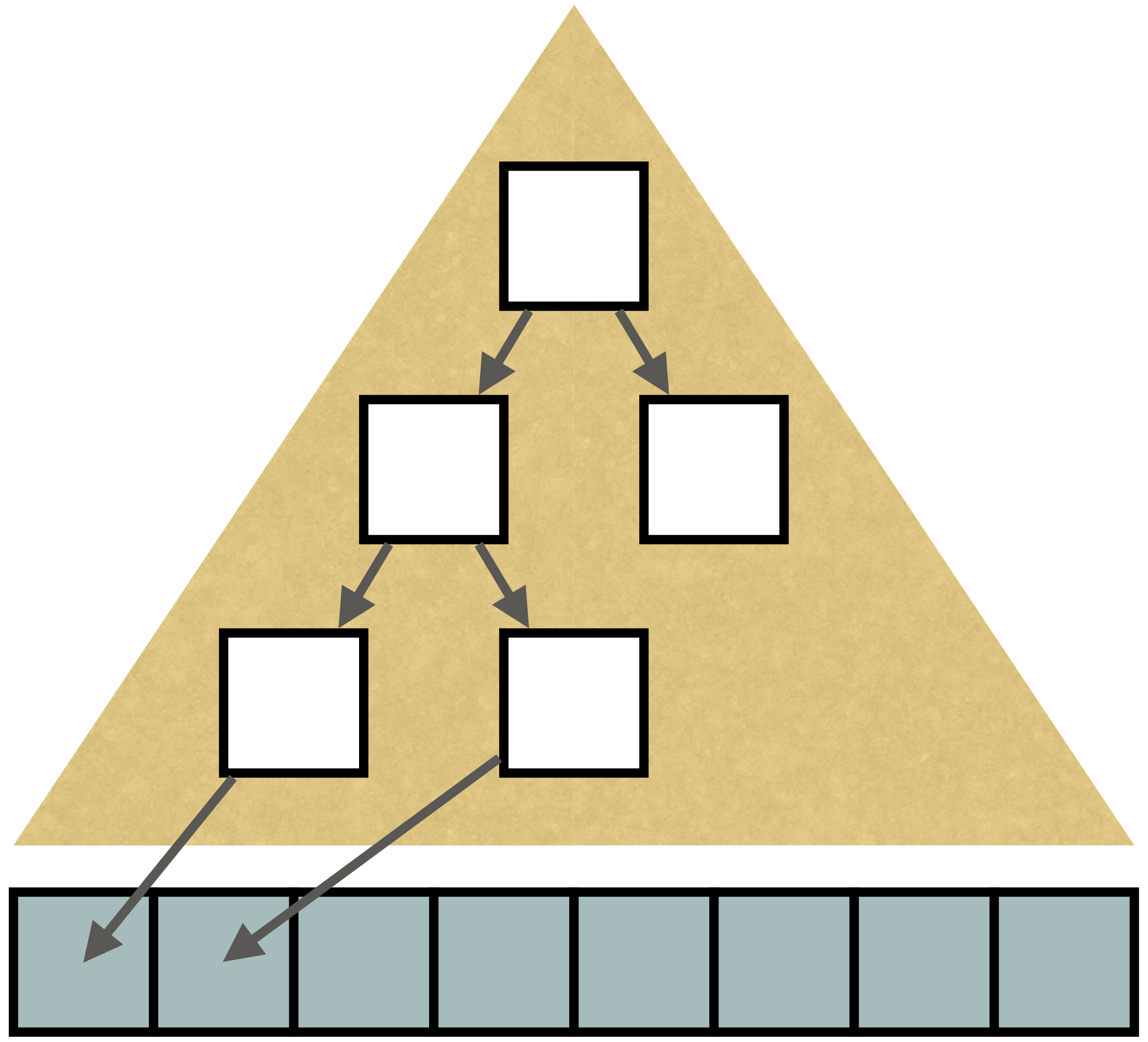
NDBI007: Practical class 2

Important Terms

- B - *page size* in bytes
- R - *object* (e.g., a record) *size* in bytes
- n - number of objects
- b - *blocking factor*, i.e., the number of objects that fit into a single page
  - Can be computed as $b = \left\lfloor \frac{B}{R} \right\rfloor$
- h - *height of a tree*, that is stored using the blocking factor b
  - Can be computed as $h = \lceil \log_b n \rceil$

Index Sequential File

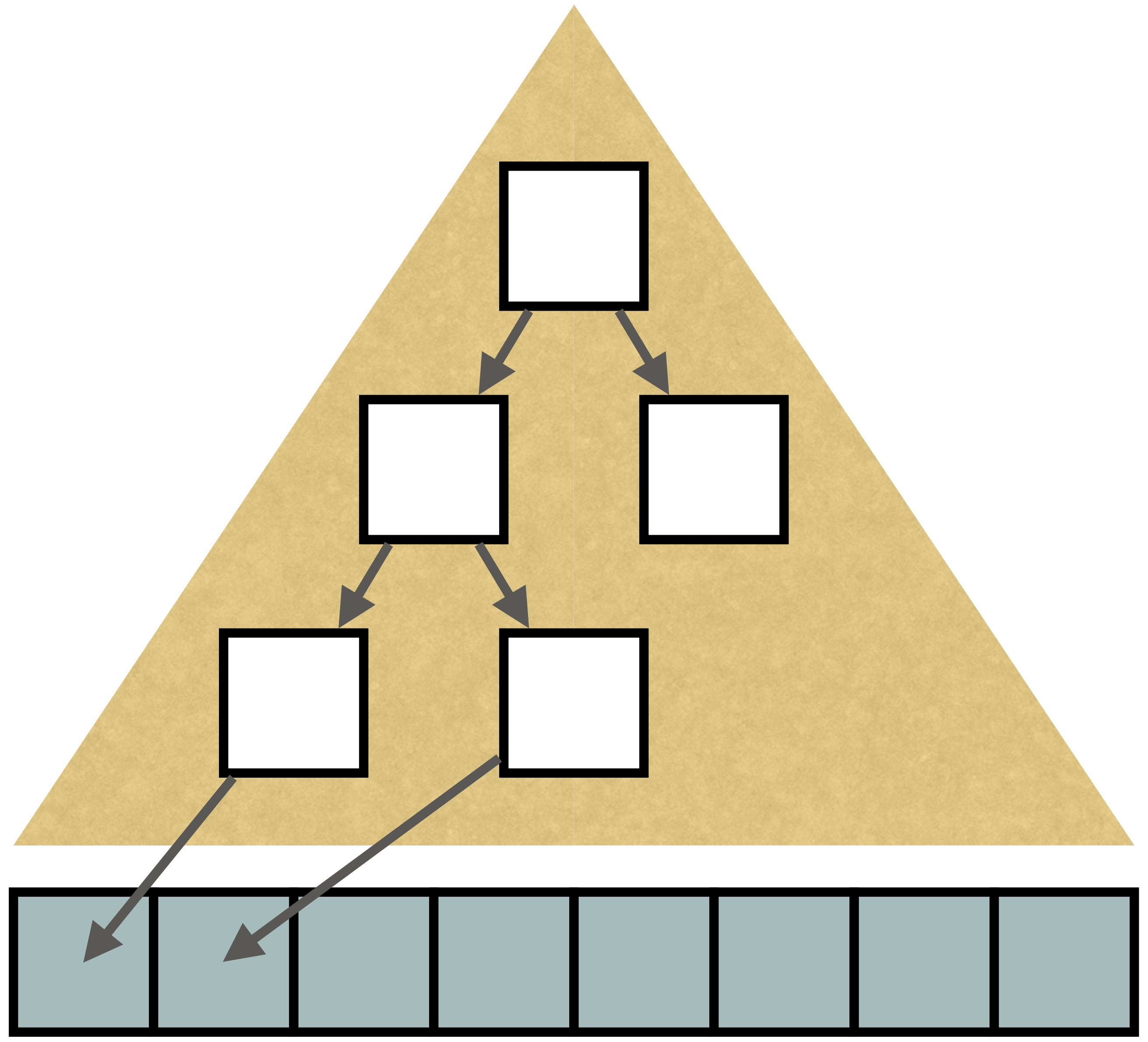
- Consists of at least two files
  - *Primary file* contains all the data, that are sorted according to a primary key
  - *Index file* contains the index of the primary file, built over the primary key
- Static index is a hierarchical structure of index pages that contains record of type [value of the primary key; pointer to a page]



- There exists the following types of static indexes
  - Primary key, non-primary (secondary) key
  - Direct index
  - Indirect index

Primary Key Index

- In sequential file, the *primary key* is *sorted* based on the primary key*
  - It is exploited in the primary index structure as it enables *omit one level of the index*



- The primary key index *record consists of two values*
  - Value of the primary key (e.g., 5 B)
  - Pointer to a page 4 B
- Total size of a record is 9 B
  - The *size is fixed* for all records
  - Only one the last level of the index the pointers point not to another index page, but to a page of the primary file

* In the case of non-sequential file, it is the same as direct index

Exercise 2.1: Primary Key Index

- Build primary key index for a sequential file that contains 5,000,000 student records (of size 256 B)
    - Determine *index height* and compute the *size of every index level*
- You will have to compute *blocking factor* b for the primary file in order to determine *number of blocks* N
    - Remember that the index (bottom) level points directly into the primary file
- You will have to compute blocking factor for the primary index
    - Suppose page size equal to 4 kB and record size 9 B
- The number of pages on the next level can be computed as

$$n_{PAGES,L=i} = \left\lceil \frac{n_{PAGES,L=i-1}}{b} \right\rceil$$

Primary Key Index: Access to Hard Drive

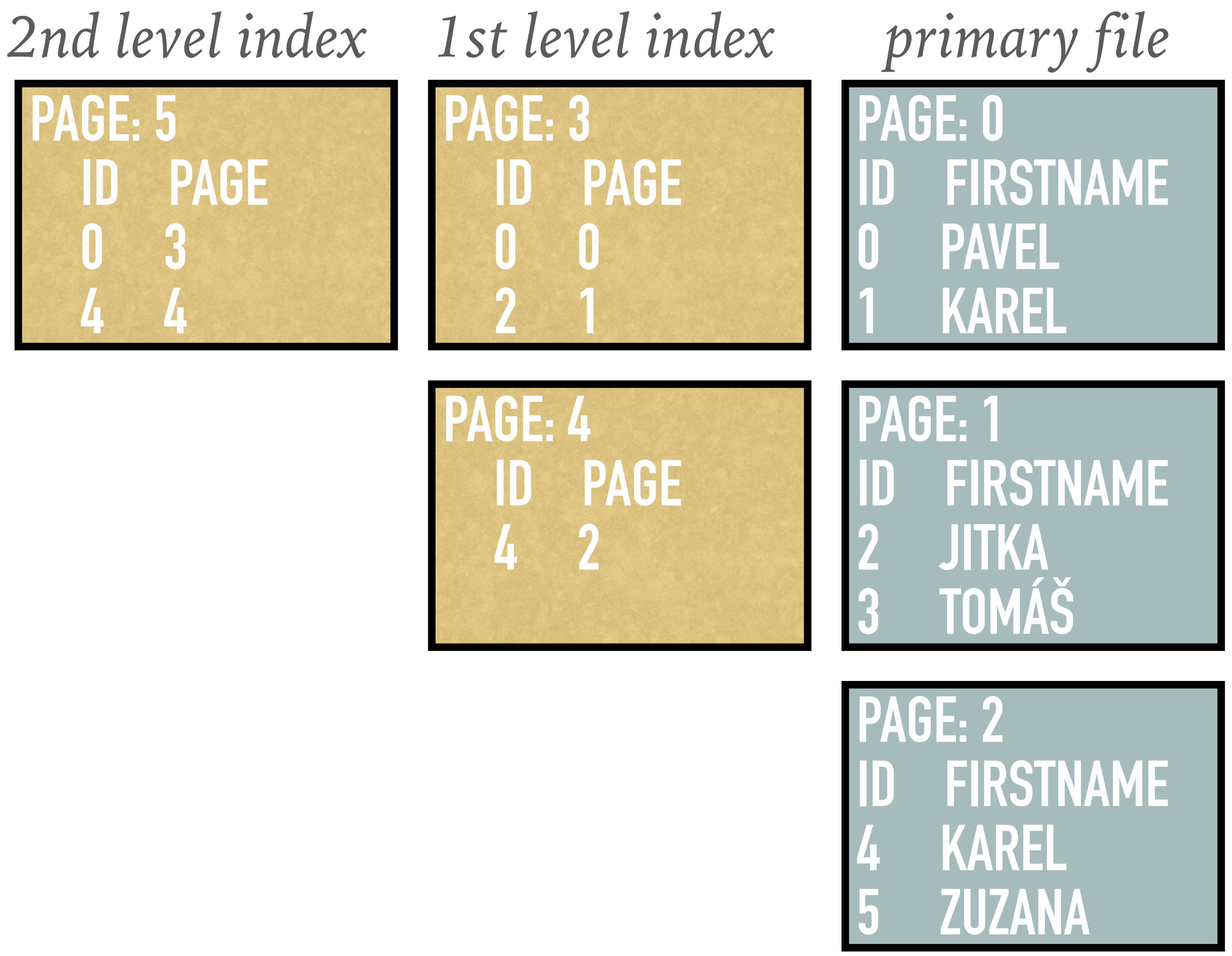
- If the index is stored in external memory, it requires $h + 1$ hard drive accesses to get a record based on a primary key
- The first two index levels are small so we keep them in the *main memory* to save external memory accesses
  - Therefore, we need only $h - 1$ hard drive accesses to retrieve a record

- In real applications, the whole primary index is commonly kept in the *primary memory* (RAM)
  - The *primary key* is typically *small* (4-8 B)
  - The retrieval of a record based on the primary key requires only 1 access to the external memory
  - The presence of primary index in main memory is also utilized by the indirect indexes

Primary Key Direct Index

ID	firstName	secondName
0	Pavel	Straka
1	Karel	Zeman
2	Jitka	Nováková
3	Tomáš	Zelený
4	Karel	Svoboda
5	Zuzana	Novotná

- Primary file cannot be sorted by keys of multiple indexes
- The sample depicts the primary key index for the database for ID*
- To see how this structure works we can query for Tomas
  - The query is ID = 3
  - We start at page 5 (index root)
  - Then we go to page 3 (we follow the highest lowest ID value)
  - From page 3 to the page 1 (the same principle as before)
  - We find Tomas on the page 1

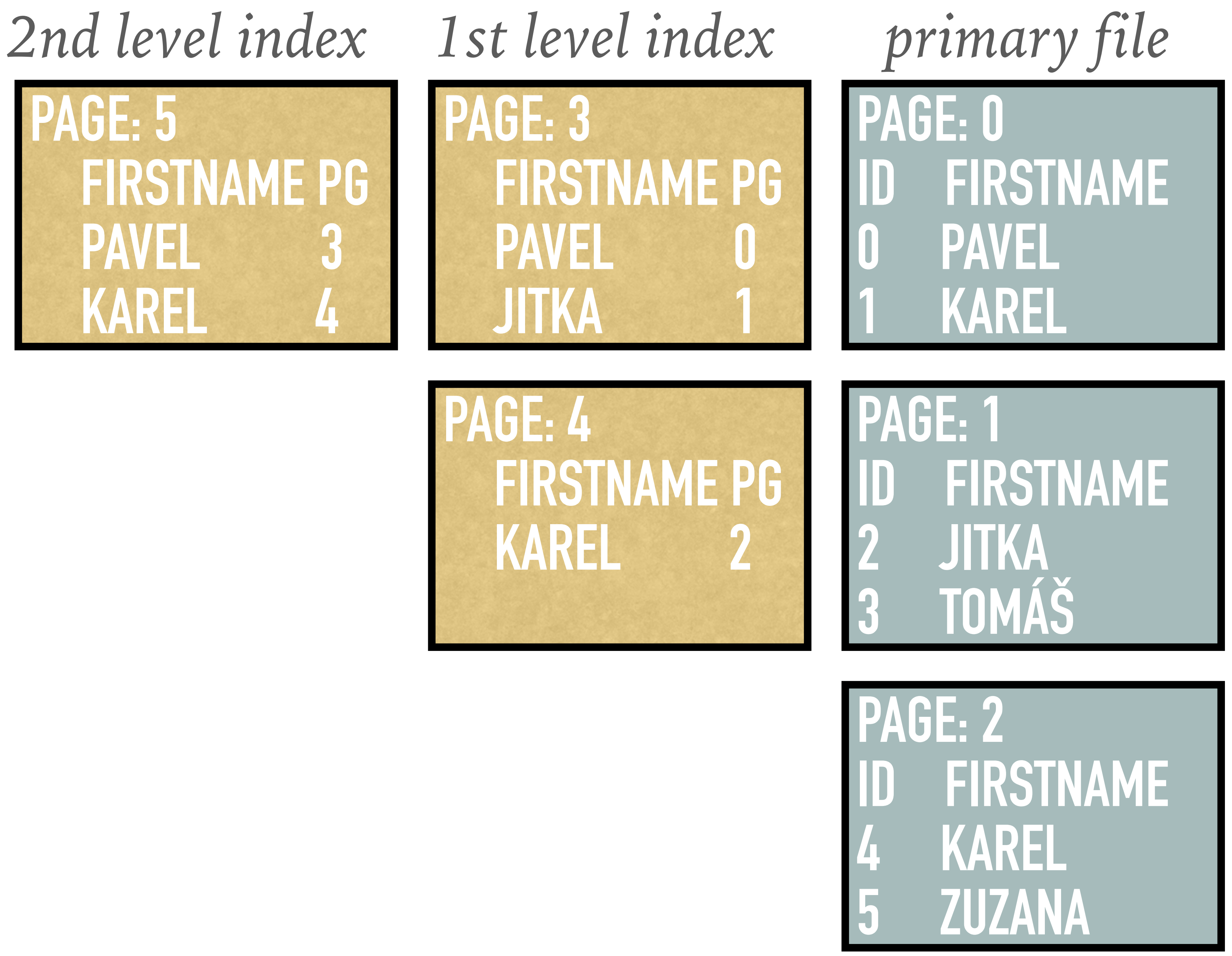


* Note, that we use different page size in pictures just to save space and make picture simpler

Non-Primary Key Direct Index

- We try to apply the same process to build a direct index for a non-primary key attribute, i.e., firstName
- However, this approaches does not work, i.e., **the index is broken**
- It can be easily demonstrated by a simple query for Karel
  - We start at page number 5 (root of the index)
  - Here, we take the largest smaller key, i.e., Pavel, and we go to page 3
  - In page 3, we repeat the same process, this time Jitka is the largest smaller key. Jitka stands for page number 1
  - But in this way we fail to retrieve Karel on page 0

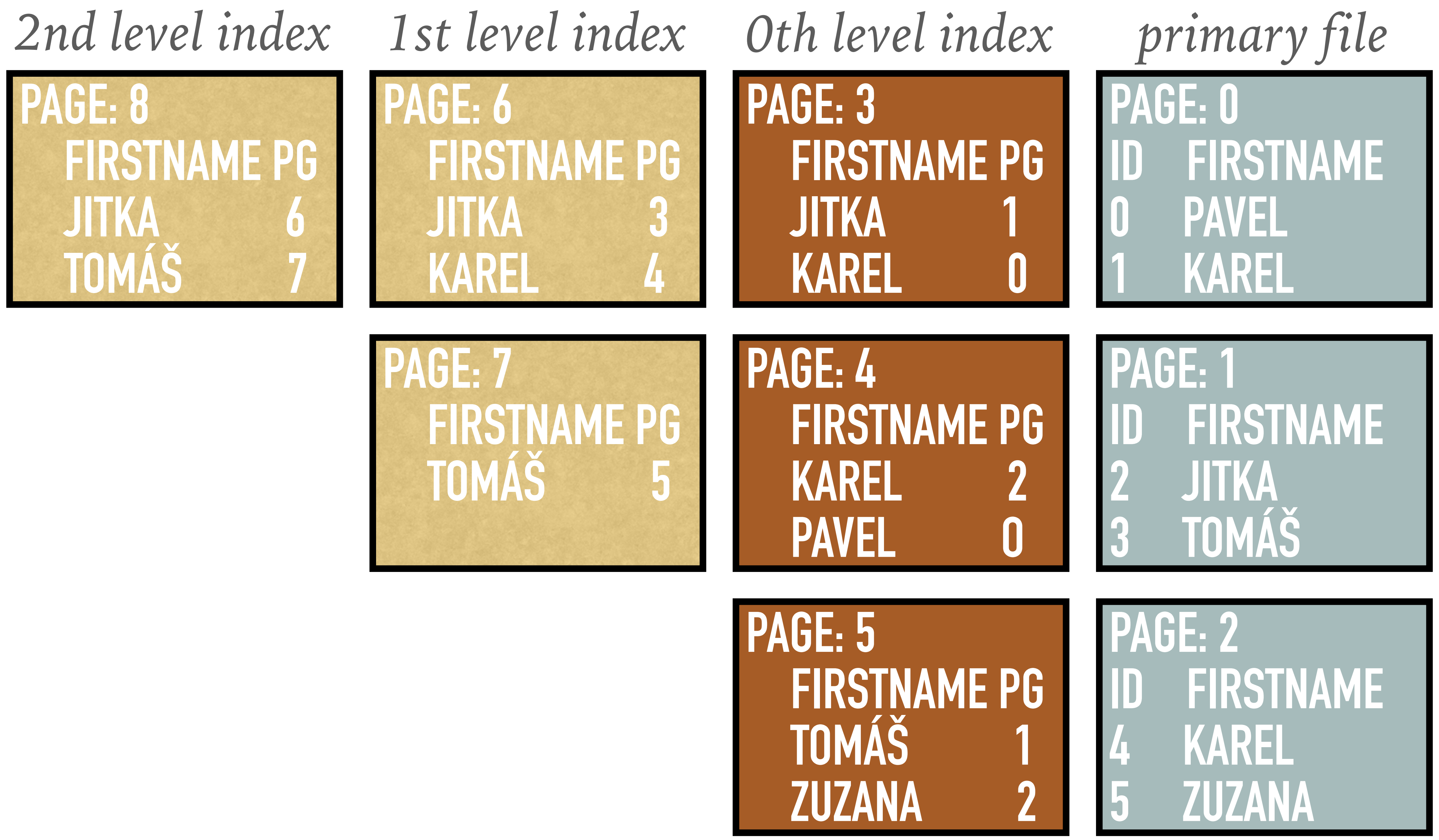
ID	firstName	secondName
0	Pavel	Straka
1	Karel	Zeman
2	Jitka	Nováková
3	Tomáš	Zelený
4	Karel	Svoboda
5	Zuzana	Novotná



Non-Primary Key Direct Index (Correct)

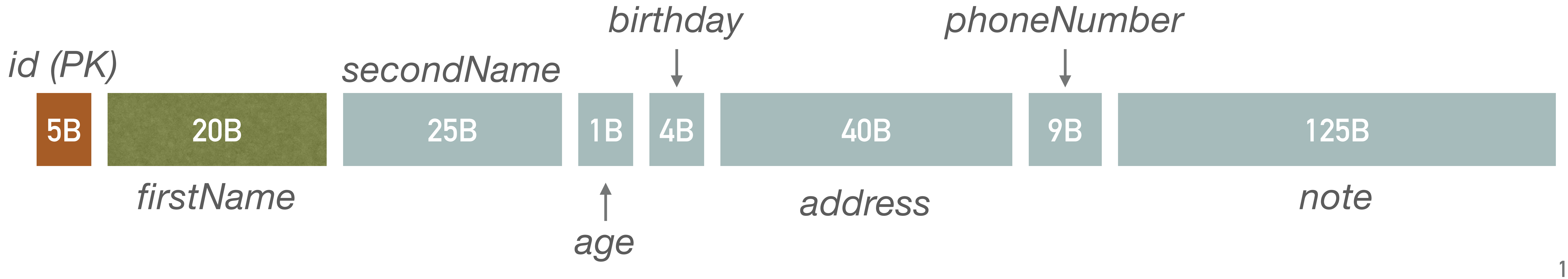
- Solution: Additional level between the index and the primary file
  - I.e., zero level index
- Query for Karel once again:
  - We start on page 8 (index root)
  - We continue on page 6 (Jitka) and then on page 3
  - Here, we see the first record for Karel, then we scan the following index page until we reach a higher key
    - Hence, we get Karel on page 0 as well as Karel on page 2
- The 0th level is a copy of given key with pointer to the respective page
  - This level is sorted by the key and it is basically a very this replication of a primary file
- Note: The "zero level index" is used in the case of non-sequential file with index
  - The primary file is not sorted by any property

ID	firstName	secondName
0	Pavel	Straka
1	Karel	Zeman
2	Jitka	Nováková
3	Tomáš	Zelený
4	Karel	Svoboda
5	Zuzana	Novotná



Exercise 2.2: Direct Index

- Build direct index on firstName for a sequential file that contains 5,000,000 student records
  - Suppose that index record is 20 B + 4 B (size of key + size of the pointer) and page size is 4 kB
  - Determine *index height* and compute the *size of every index level*
  - Compare the structure with primary key index structure
    - I.e., number of levels, sizes of levels, total size of index (in MB)

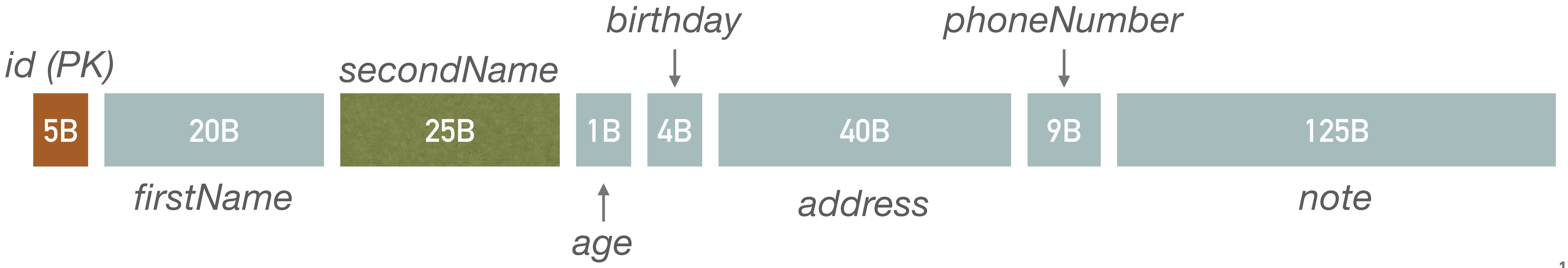


Indirect Index

- Direct indexing and primary index share one disadvantage
  - In the case of any modification (records shuffling) in the primary file, the first (zero) level must be updated
- The solution is indirect indexing that does not point to the primary file pages
  - It points to the primary keys, i.e., indirect index can be described as a map from some property to a primary key
  - In addition, indirect index does not point to the file directly, therefore it is not affected by modifications of the primary file
  - As the primary index is commonly stored in primary memory (RAM), we just need to read pages from the indirect index and retrieve pages from the primary file
- Although the first level is slightly larger than that of a direct index, the main advantage is that an indirect index does not need to be updated in case od primary file movements

Exercise 2.3: Indirect Index

- Build indirect index on secondName for a sequential file that contains 5,000,000 student records
  - Note that first level records and other level records differ in its size
    - First level: 25 B + 5 B (secondName key size + primary key size)
    - Other level: 25 B + 4 B (secondName key size + pointer to another page)
  - Determine *index height* and compute the *size of every index level*



Searching in Index From Multiple Attributes

- Two properties can be concatenated (e.g., firstName and secondName)
  - Enables us to search for both of the attributes at once
  - Attribute ordering in the index is fixed
    - E.g., firstName followed by secondName does not allow us to search for secondName followed by firstName

Bitmaps

- Note: Having 50 percent men and 50 percent women in our database, usage of previous indices is not effective at all
  - We prefer bitmaps with database sequential scan over hierarchical index

- Bitmap consists of multiple columns
- Each column is stored in *separate page*
  - Pages are *stored sequentially*, allowing effective reading
- A value of a given column is represented by a single bit
  - E.g., having page size 4 kB, we can store $4{,}096 \bullet 8 = 32{,}768$ values in every page
  - Useful for attributes having *small domain*, e.g., traditional concept of gender (male, female)
- Bitmaps allow *effective evaluation of logical operations* over columns (T = 1, F = 0)
- Based on the value distribution, we may also consider some *compression* (e.g., RLE compression*)

ID	isMale	isFemale
0	1	0
1	1	0
2	0	1
3	1	0
4	1	0
5	0	1

* https://en.wikipedia.org/wiki/Run-length_encoding

Example 2.4: Bitmaps for Birthdays

- Birthday (day, month) can be represented in different ways using bitmaps

One column for each day in year

- Positives:
  - *One column is read* to get all people having birthday in a certain day
  - We can *easily add information* about other important day for a price of just another single column
- Negatives:
  - Bitmap takes *a lot of space*, i.e., $366 \bullet 153 \bullet 4 \bullet 2^{10} \approx 218.7\ MB$
    - Compression may decrease the size but read time increases as we need to decompress bitmap

ID	01/01	...	07/03	...	31/12
0	1	...	0	...	0
1	1	...	0	...	0
2	0	...	1	...	0
3	0	...	1	...	0
4	1	...	0	...	0
5	0	...	0	...	1

* We consider database having 5,000,000 records, hence $5{,}000{,}000 \div 32{,}768 = 153$ pages are required to store single column

Example 2.4: Bitmaps for Birthdays (Continued)

Two sets of bitmaps

- *One for day* (31) and *other for months* (12)
- We need a single AND operation to read this
- Positives:
  - *Smaller size*, i.e., $43 \bullet 153 \bullet 4 \bullet 2^{10} \approx 25.7\ MB$
- Negatives:
  - We have to *read two columns* to get information about birthdays in a given day

day

ID	1	...	7	...	31
0	1	...	0	...	0
1	1	...	0	...	0
2	0	...	1	...	0
3	0	...	1	...	0
4	1	...	0	...	0
5	0	...	0	...	1

month

ID	1	...	3	...	12
0	1	...	0	...	0
1	1	...	0	...	0
2	0	...	1	...	0
3	0	...	1	...	0
4	1	...	0	...	0
5	0	...	0	...	1

Example 2.4: Bitmaps for Birthdays (Continued)

Binary representation of a day in a year

- Number 366 can be saved into 9 bits
  - E.g., 01/01 = 000 000 001, 02/01 = 000 000 010, 01/02 = 000 100 000
- Positives:
  - *Much smaller size*, i.e., $9 \bullet 153 \bullet 4 \bullet 2^{10} \approx 5.4\ MB$
- Negatives:
  - We have to *read all columns* to find all birthdays in a certain day

ID	9	...	3	2	1
0	1	...	0	1	0
1	1	...	0	1	1
2	0	...	1	0	0
3	0	...	1	0	1
4	1	...	1	1	0
5	0	...	1	1	1