# Database Storage

# Resources

- Architecture of a Database System (Chapter 5)
  https://dsf.berkeley.edu/papers/fntdb07-architecture.pdf

- Postgres documentation

- Oracle documentation
  https://docs.oracle.com/cd/E11882_01/server.112/e40540/physical.htm#CNCPT1389

# Destiny of Data : Queries

- What happens when we run a query ?
- Are all queries “equal” ?
- Are all systems good at answering queries ?

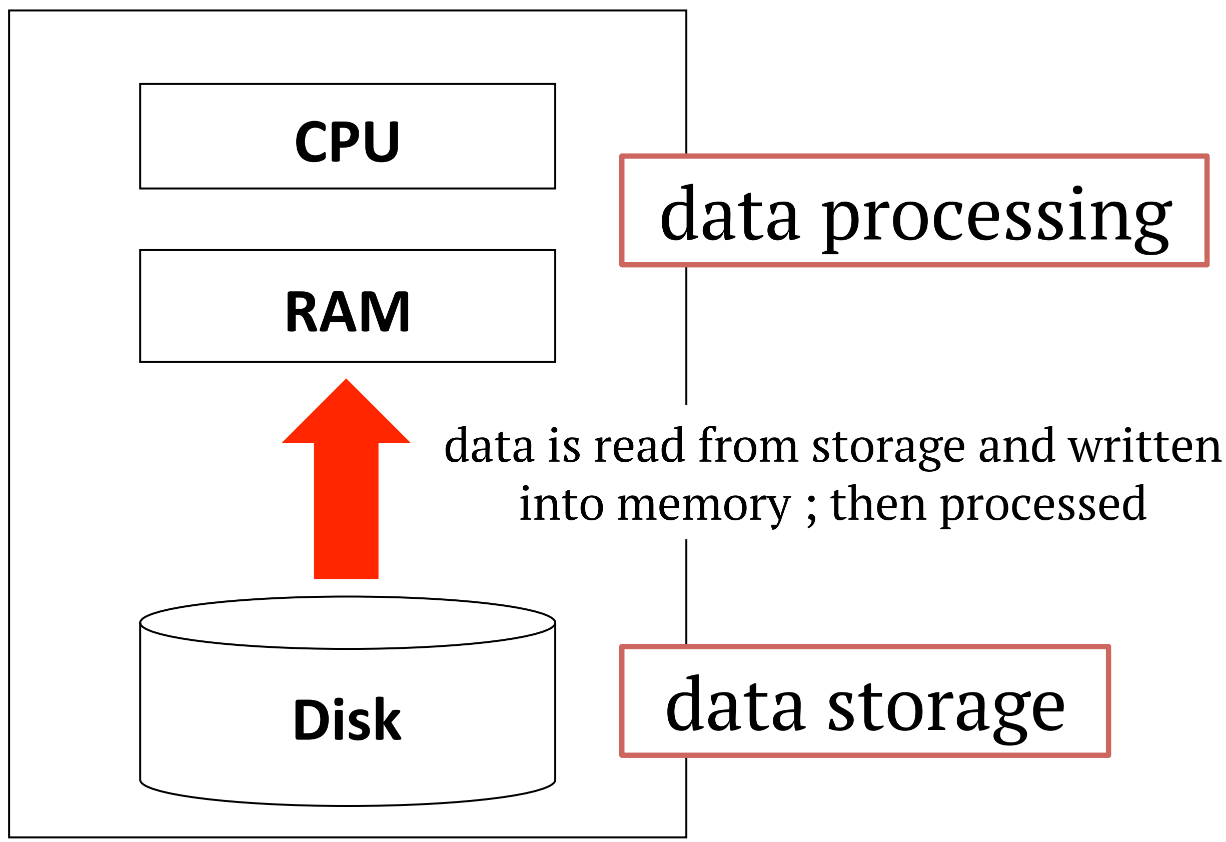
# What happens when we run a query ?

- Well, the data is read and the query evaluated

- Where is data read from ?
  - Disk
    - Data is persistent
    - It may not fit in memory (but there are exceptions)

- Where is the query computed ?
  - CPU
    - At query time, data moves “up” from disk to CPU registers

# What happens when we run a query?

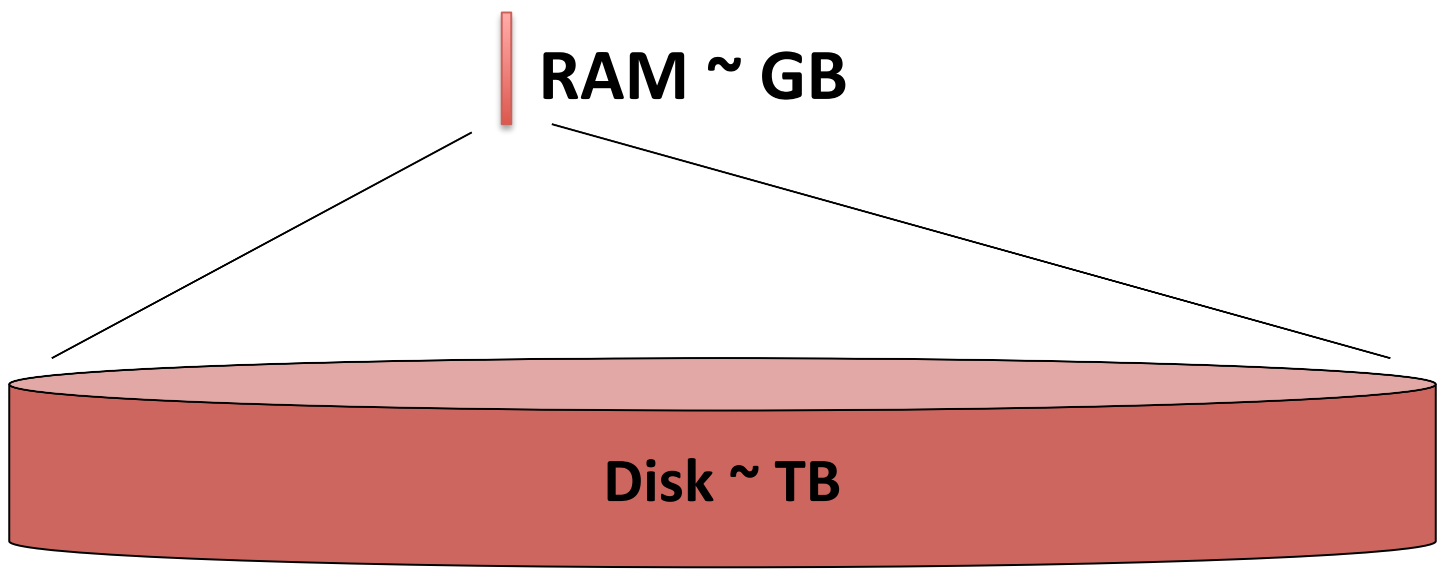


# Memory : the state of affairs

sources : https://jcmit.net/memoryprice.htm https://jcmit.net/diskprice.htm

| | Speed (Read/Write) | Cost/MB | |
|---|---|---|---|
| Cache | L1 read **3**TB/s | ~1000$/GB | fastest and most costly storage; volatile; managed by computer hardware |
| RAM | DDR4 read ~**25**GB/s | ~10$/GB | ~100x slower & cheaper than cache |
| Disk | SSD read ~**0.5**GB/s | ~0.2$/GB | Primary medium for the long-term storage of data |

# Memory : the state of affairs




- Data may not fit in memory, and R-DBMS architecture should account for this

# The Query-Evaluation “Game”

- Compute answers to queries on :
  - (Possibly large) volumes of data stored on disk
  - Limited (but fast) memory

  **Within a useful time** (useful for the user/application)

- To “win the game”, one needs to devise a <u>strategy</u> for :
  - **Organizing** data
  - **Moving** data from disk to memory
  - **Optimizing** query computation

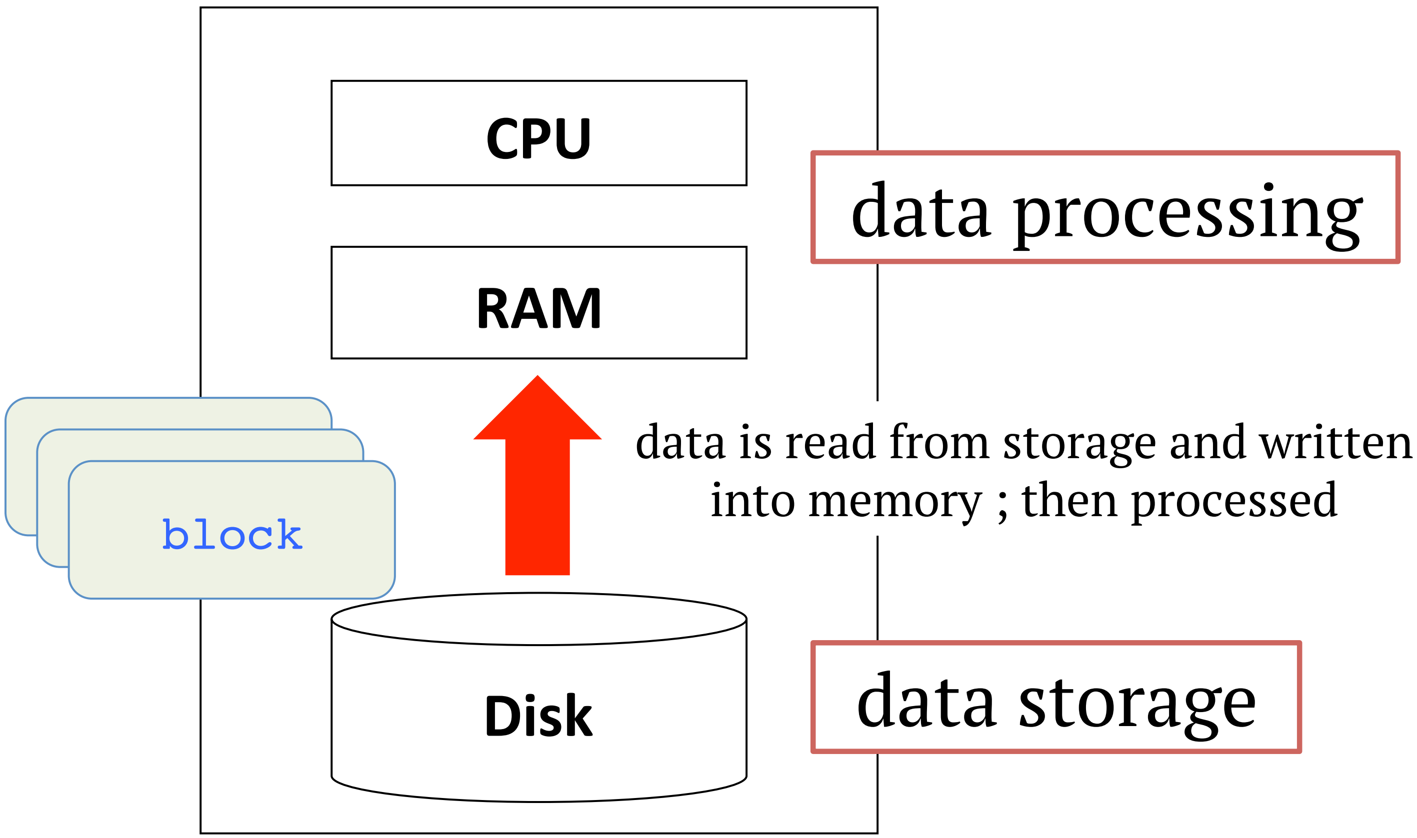
# Data Layout : Postgres Demo

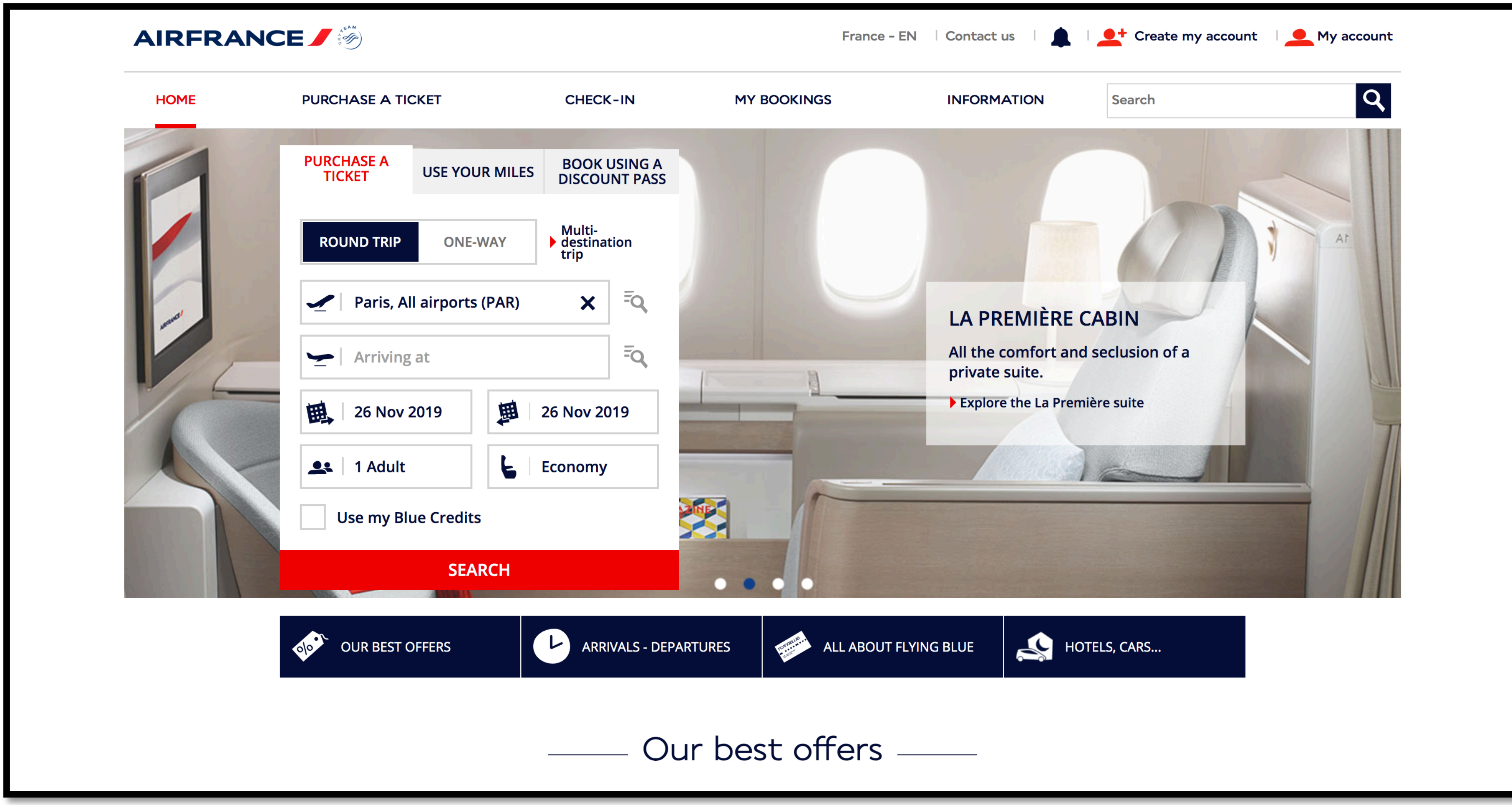
# So what is Postgres doing ?

https://link.springer.com/content/pdf/bbm%3A978-1-4302-0018-5%2F1.pdf

- Postgres stores table data in multiple ***files***.
  - each file can grow up to 1GB (this is a choice of the Postgres system)
- A file stores a set of database **records**.
- Records are partitioned into <u>*fixed-length storage units*</u> called **blocks**.
  - default size (tunable) : 8KB (maximum Postgres 32K)
  - each block-id have a 32-bit integer ID (allows ~2 billion blocks)
  - max table size : #blocks x block_size (16TB to 64TB)
- **Blocks are units of both storage allocation and data transfer.**
  - Neither single records (as one may think at first), nor files are transferred from disk to memory : blocks !

# What happens when we run a query?






(2018) 4$^{th}$ european company

100+ million passengers

300+ destinations

# LOG IN

Flying Blue number or e-mail address

Forgot your Flying Blue number?

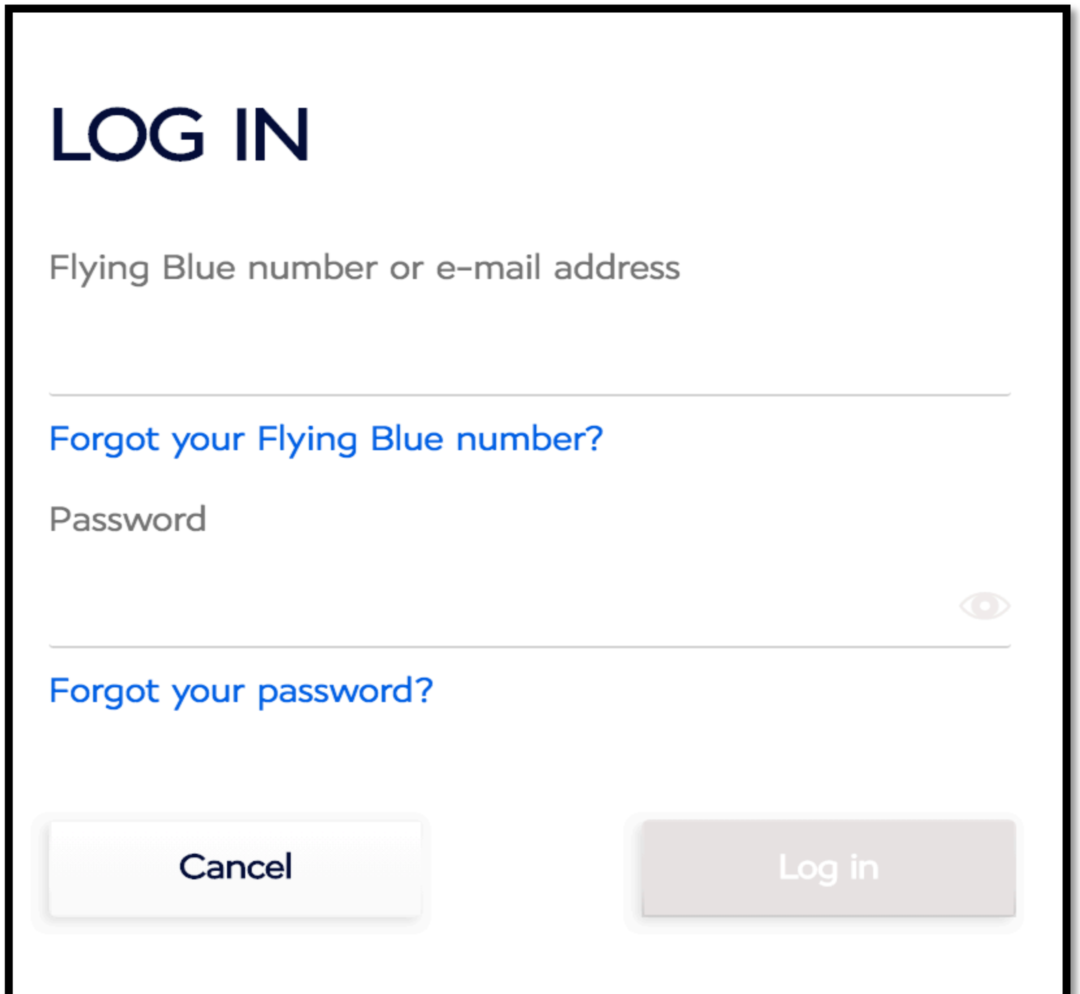
Password

Forgot your password?

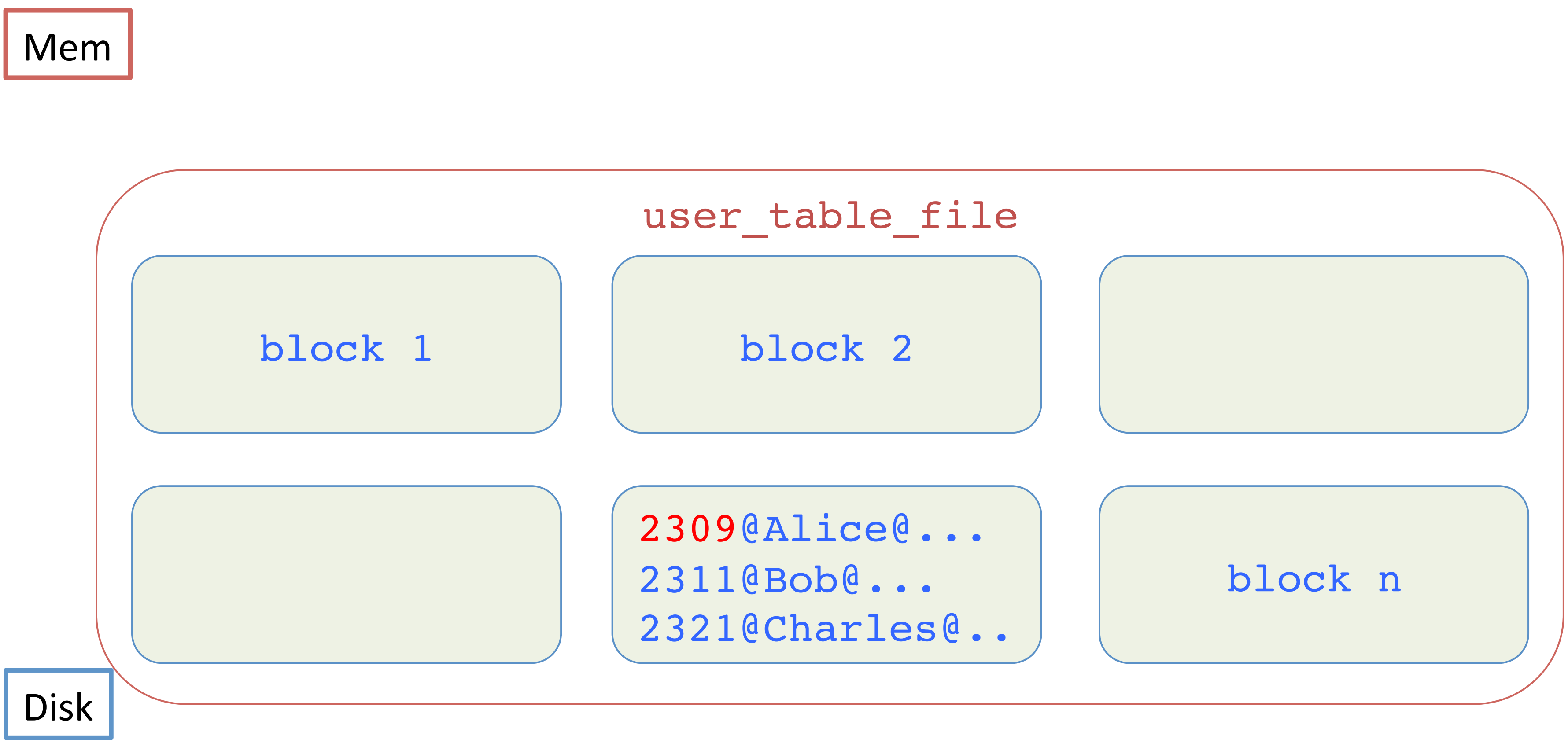
Cancel

Log in

```sql
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



```
SELECT  *               #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



```
SELECT  *               #user profile data
FROM    users_table
WHERE   user_ID = 2309
```

Mem

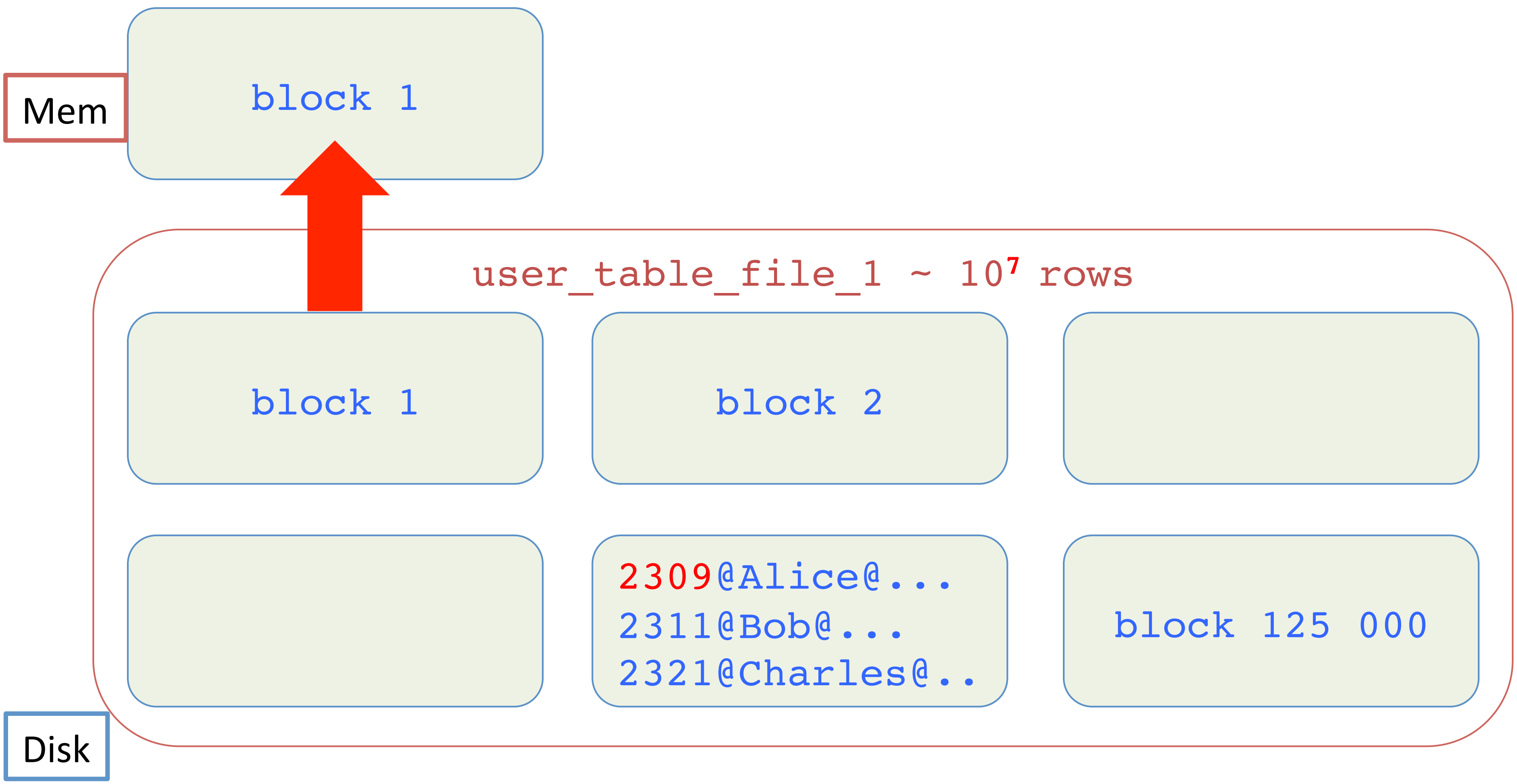
block 1

Assume client record 100 Bytes
Assume block size 8K => 80 clients per block
Assume 30M registered accounts / 80 => 375K blocks
1 file maximum 125K blocks => 3 files

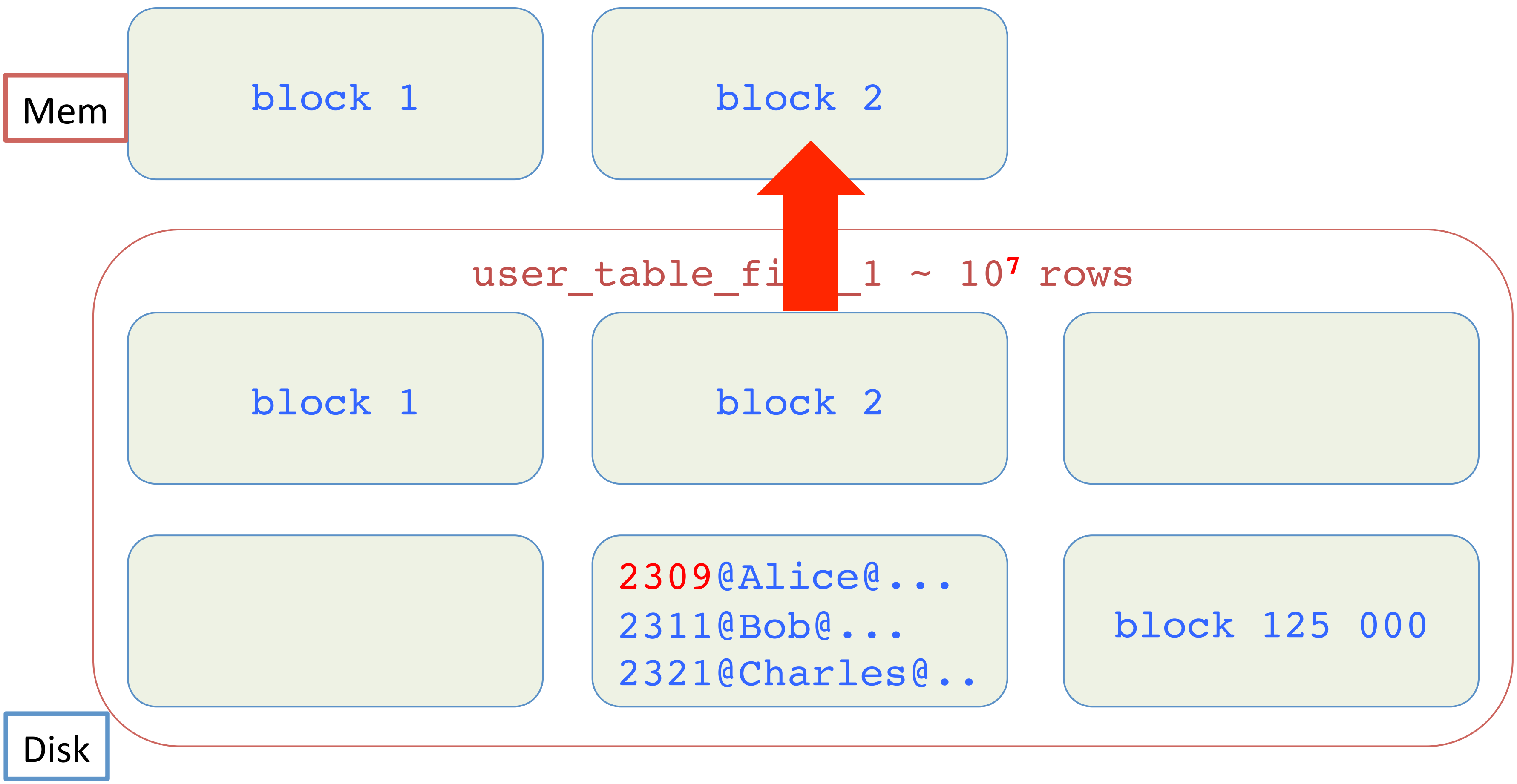
block 125 000

Disk

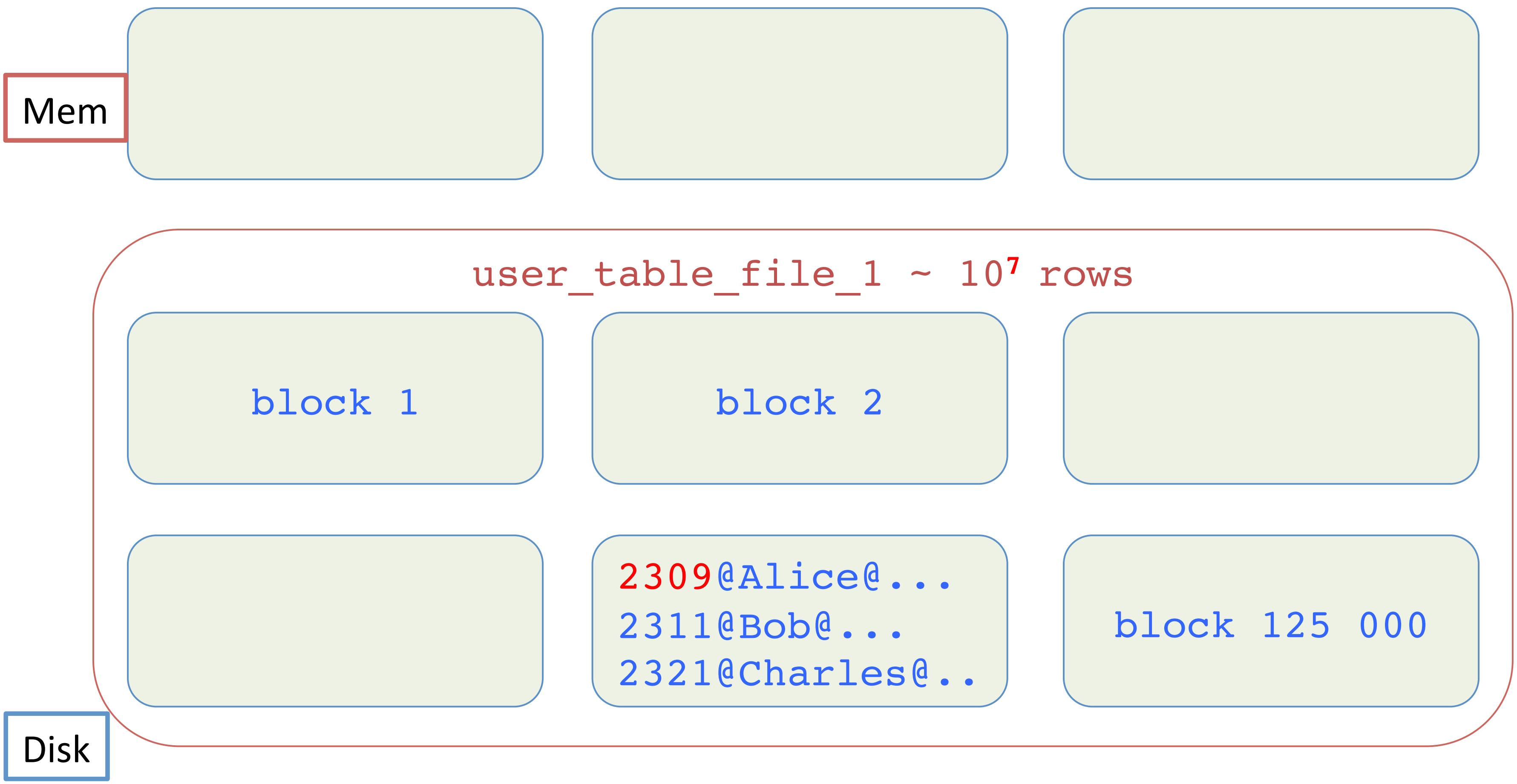
```
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



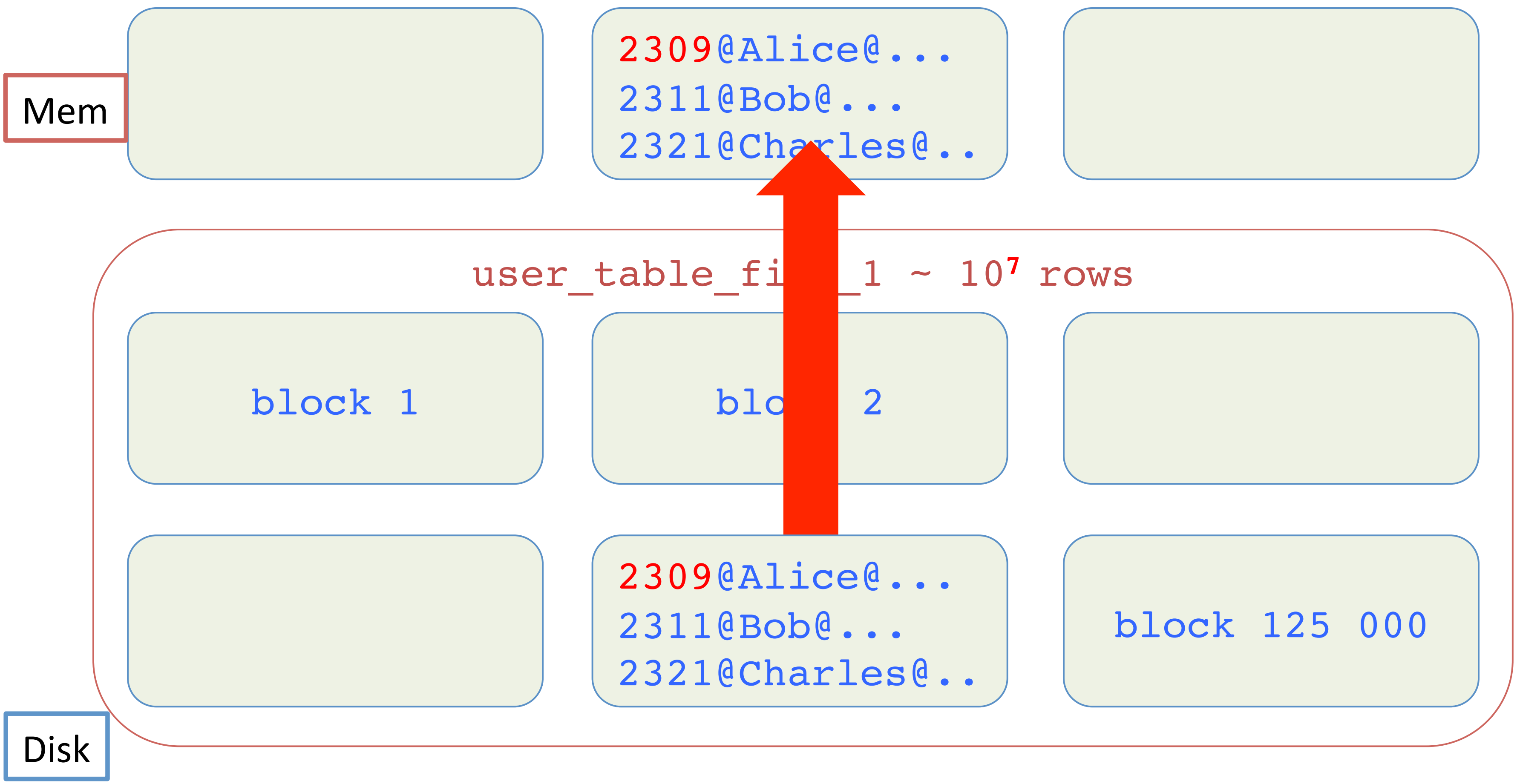
```
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



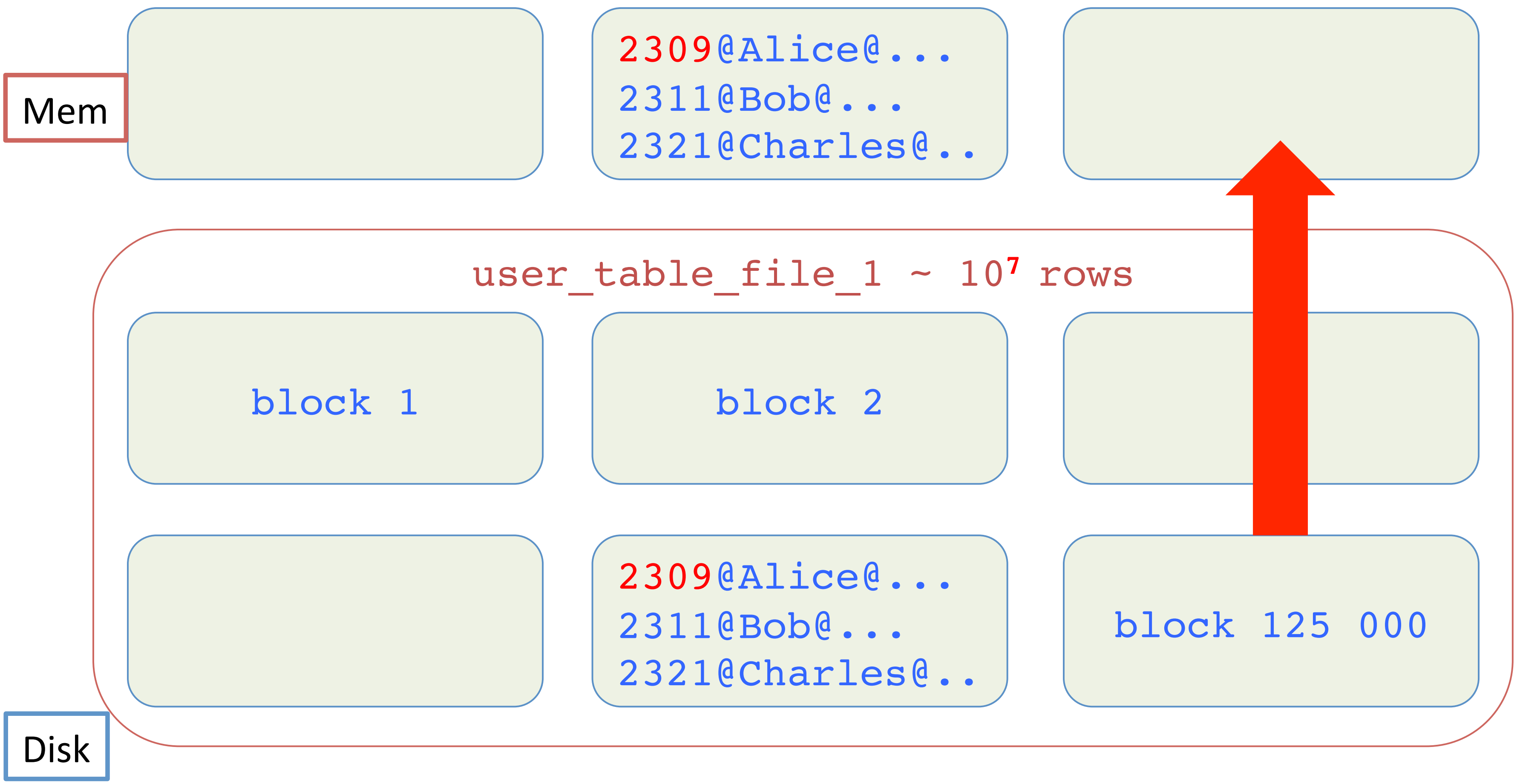
```
SELECT  *               #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



```
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



```
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309
```



```sql
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309
```

```
SELECT  *            #user profile data
FROM    users_table
WHERE   user_ID = 2309     #PK
```

In reality DB use indexes !!

index
ROW for **user_ID** = 2309
block 7459 offset 45

Mem

user_table_file_1 ~ $10^7$ rows

block 1

block 2

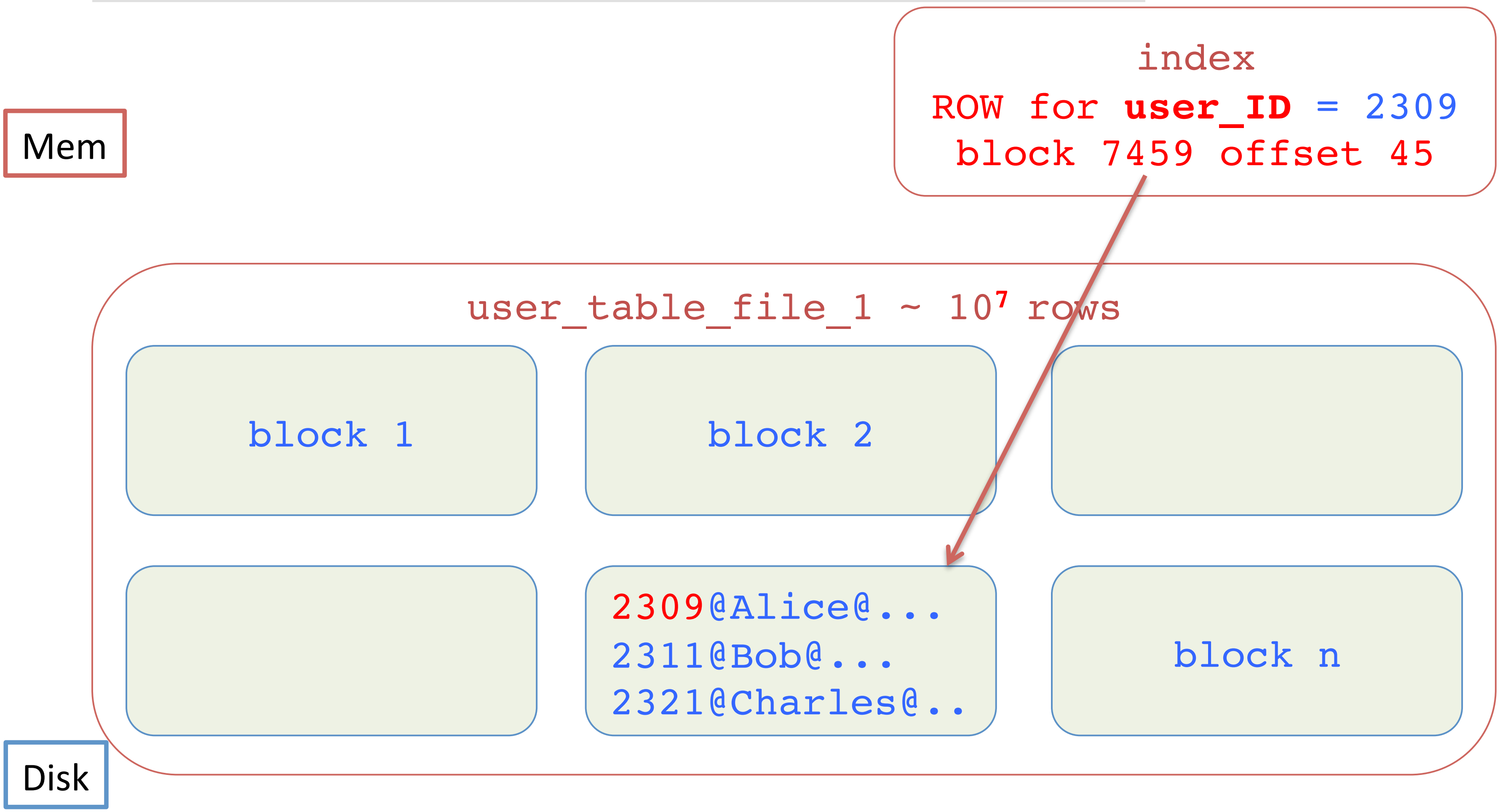
2309@Alice@...
2311@Bob@...
2321@Charles@..

block n

Disk

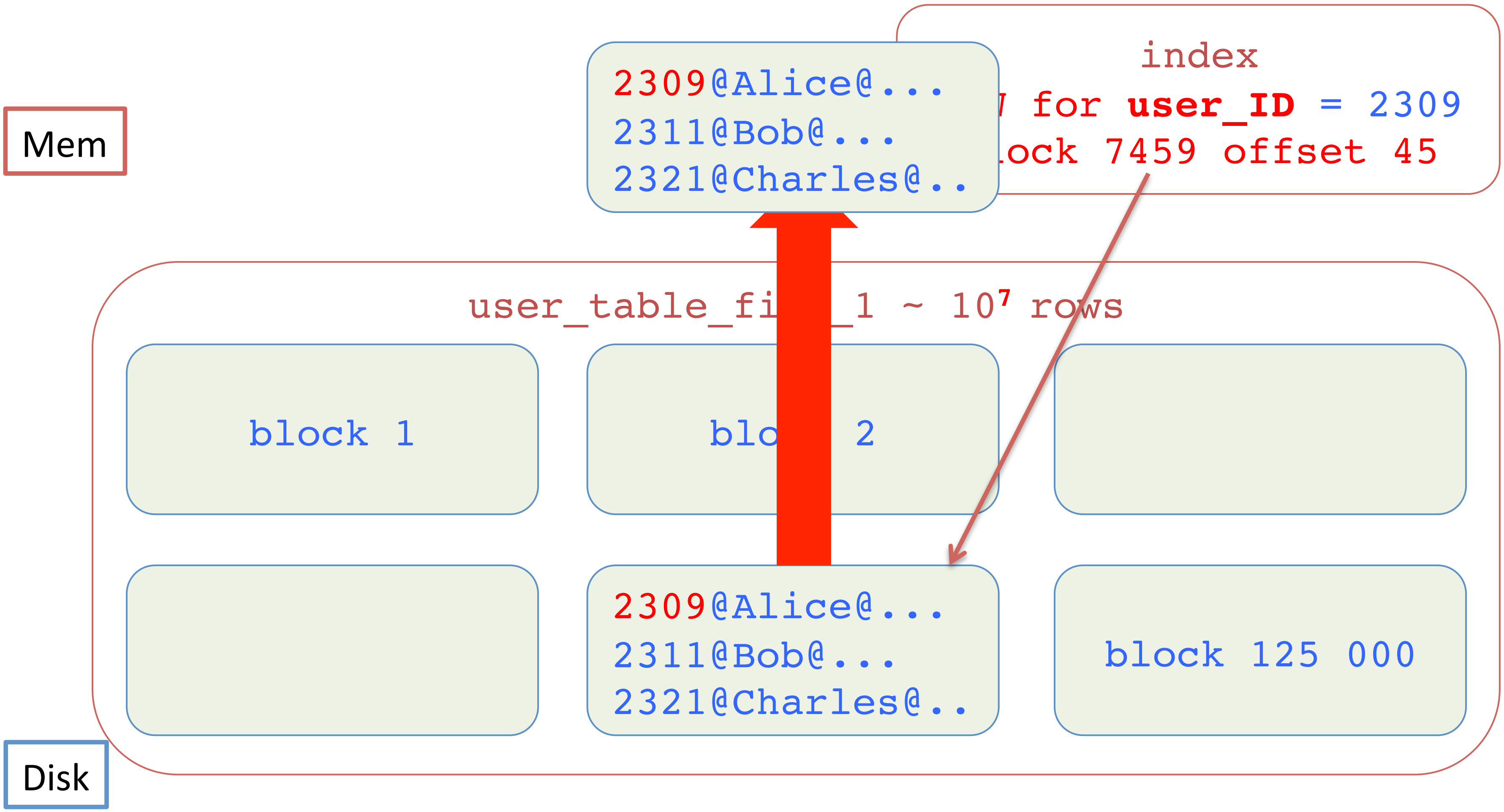
```
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309      #PK
```

In reality DB use indexes !!



```
SELECT  *              #user profile data
FROM    users_table
WHERE   user_ID = 2309     #PK
```

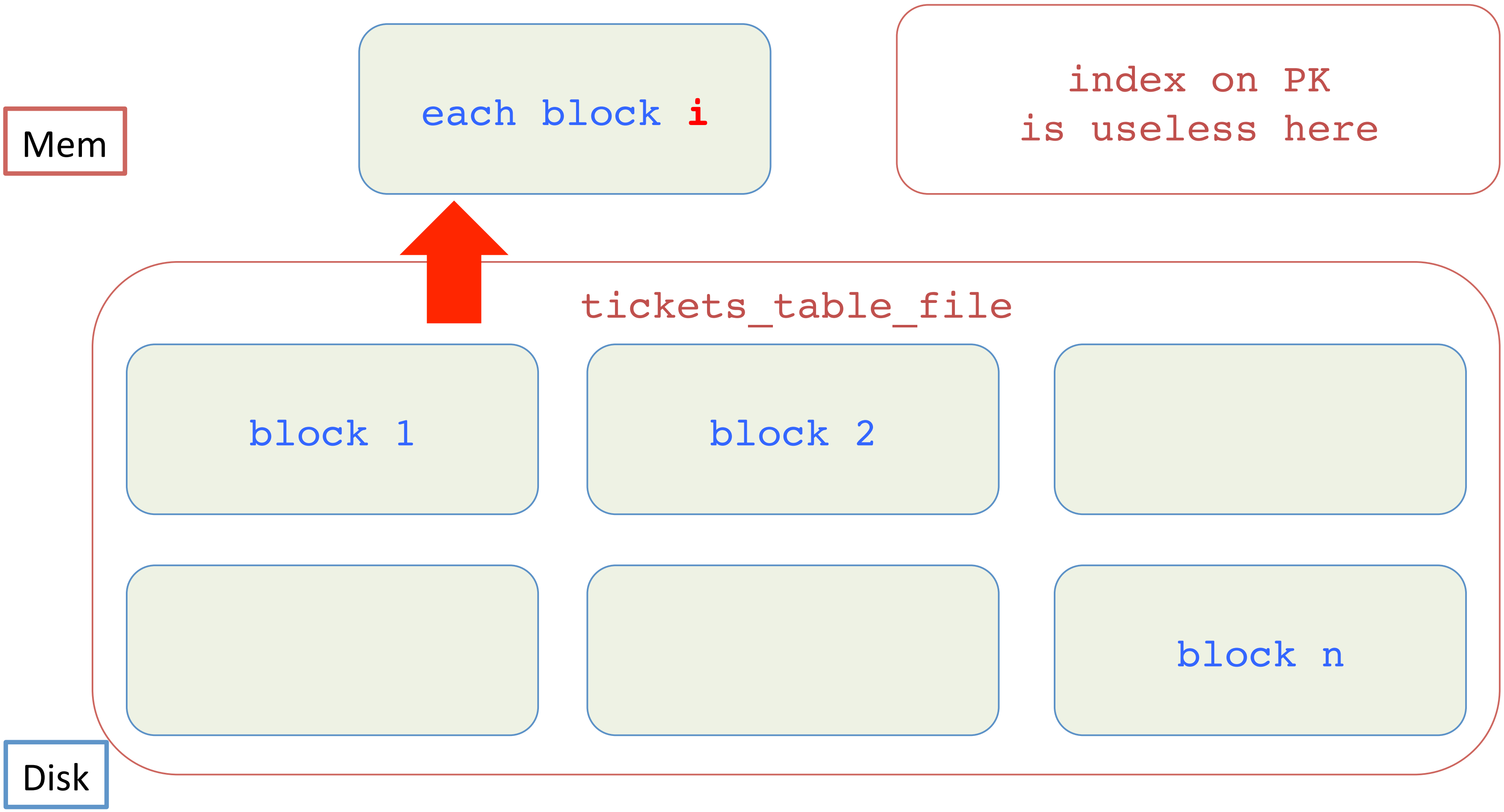
In reality DB use indexes !!



```
SELECT    SUM(price)
FROM      tickets_table
WHERE     year = NOW.year
```

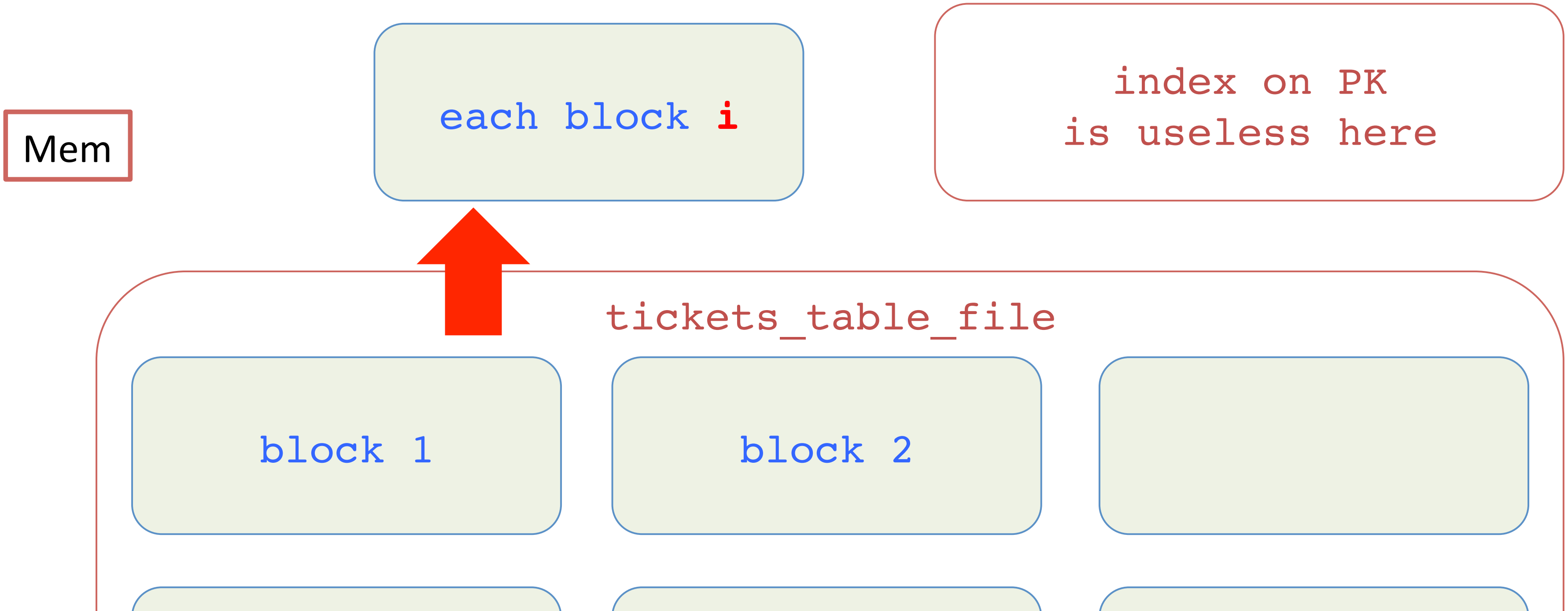
```
SELECT  SUM(price)
FROM    tickets_table
WHERE   year = NOW.year
```

Analytics case



```
SELECT  SUM(price)
FROM    tickets_table
WHERE   year = NOW.year
```

Analytics case




*Quiz : how many files and blocks/year with a 40 Byte ticket record in Postgres ? Assume 100M tickets/year.*

# The Query-Evaluation “Game”

- **Blocks are units of both storage allocation and data transfer.**
  - Neither single records (as one may think at first), nor files are transferred from disk to memory : **blocks** !

# The Query-Evaluation "Game"

- To "win the game" the DB seeks to minimize the number of block transferred from disk to memory
  - avoid loading a block twice
  - avoid loading useless blocks
  - keep as many blocks as possible in main memory
    - Locality principle
  - reduce the number of disk accesses

# Is Postgres showing us the universal strategy ?

https://dsf.berkeley.edu/papers/fntdb07-architecture.pdf

Spatial control of data : where data is placed on disk

1. Use the typical OS file system facilities
   (like Postgres)

2. Interact directly with the device drivers for the disks
   (raw disk acces)

Crux : sequential access to disk blocks is between 10 and 100 times faster than random access.

Current solution: allocate **1 large file** controlled via OS

# Is Postgres showing us the universal strategy ?

https://dsf.berkeley.edu/papers/fntdb07-architecture.pdf

Temporal control of data : when data gets physically written to disk

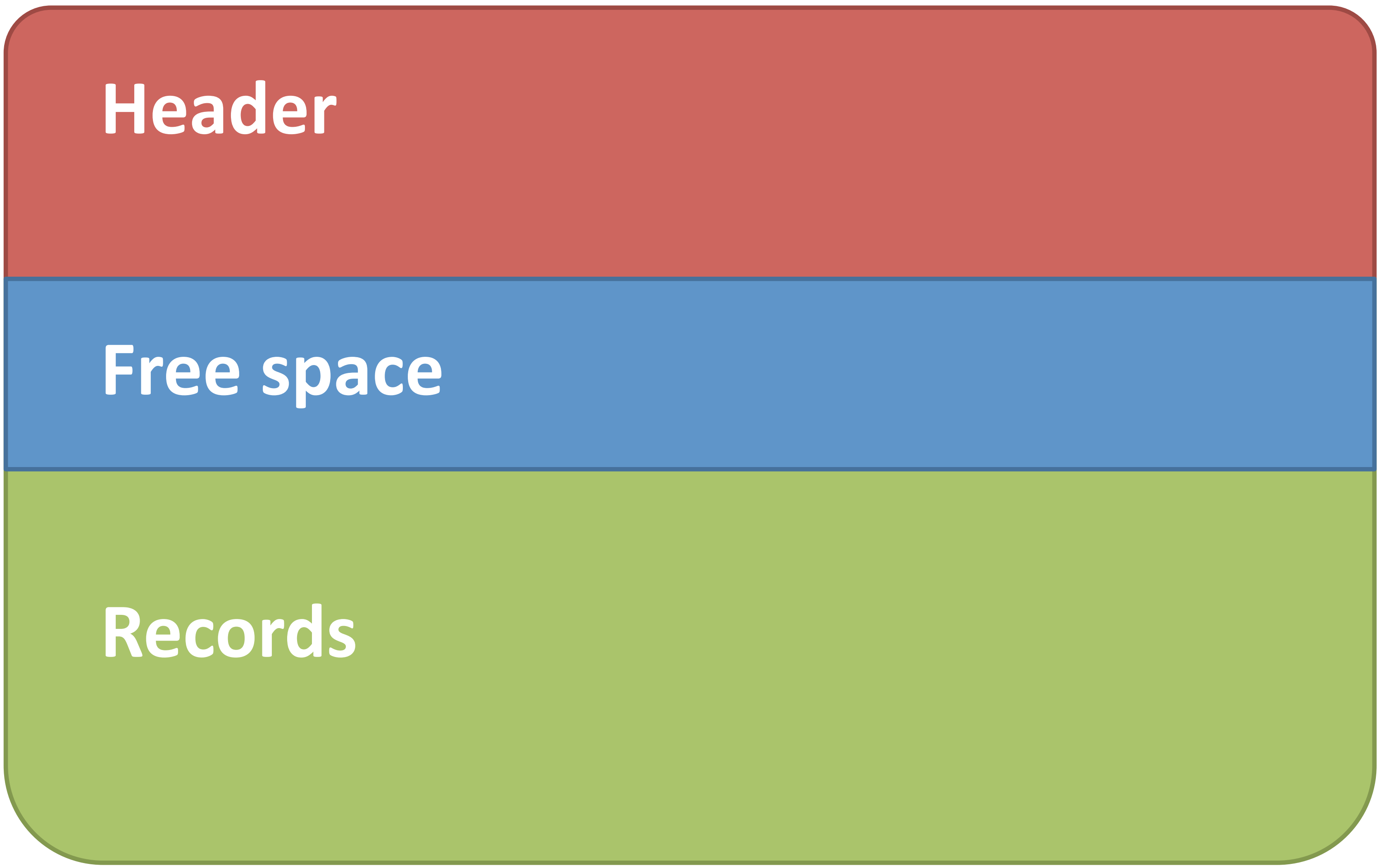
1. Use the typical OS file system facilities
   (like Postgres)

2. Interact directly with the device drivers for the disks
   (raw disk acces)

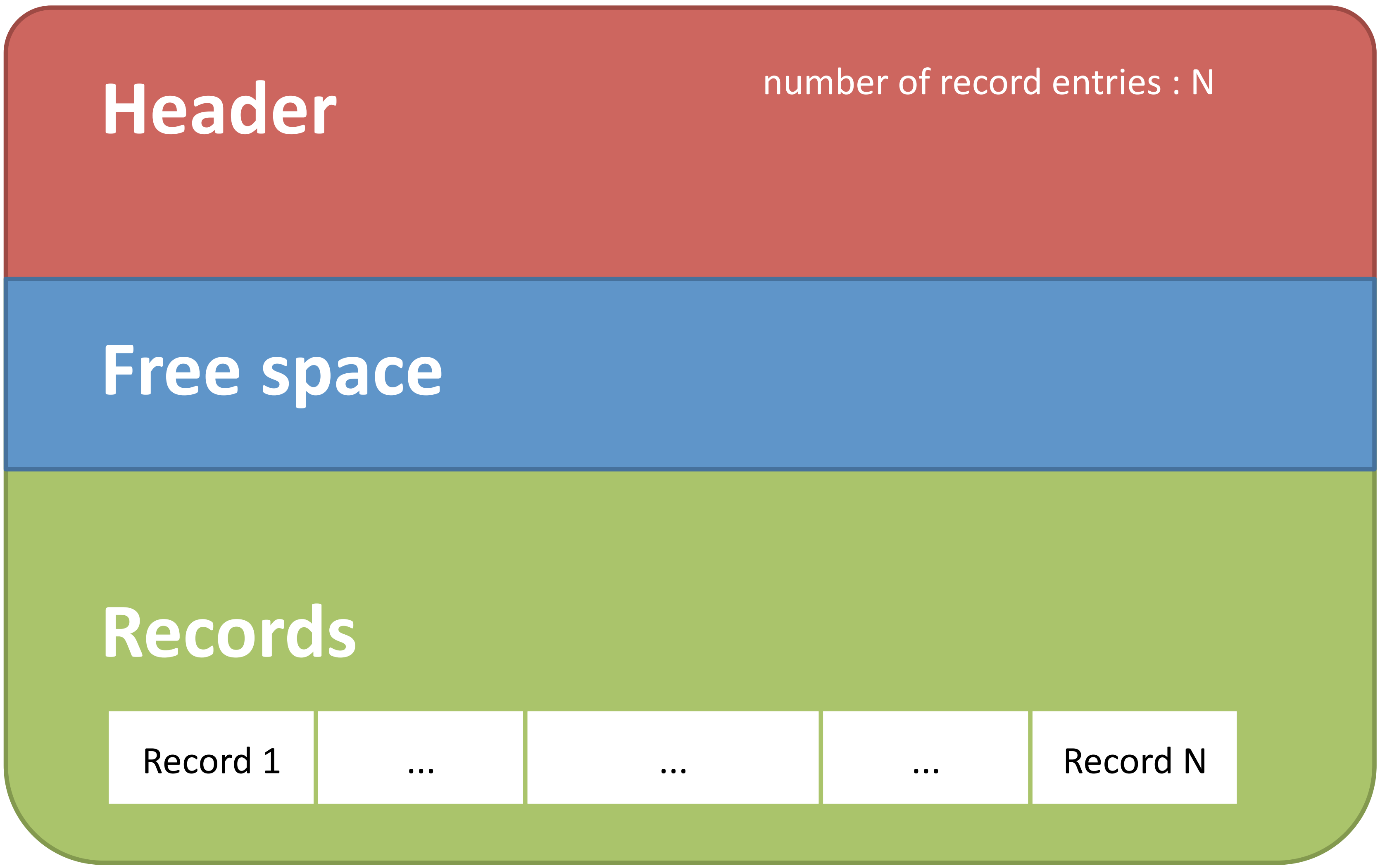
Crux : OS buffering can confound the intention of the DBMS by silently **postponing or reordering writes**

Current solution : use specific APIs provided by OS

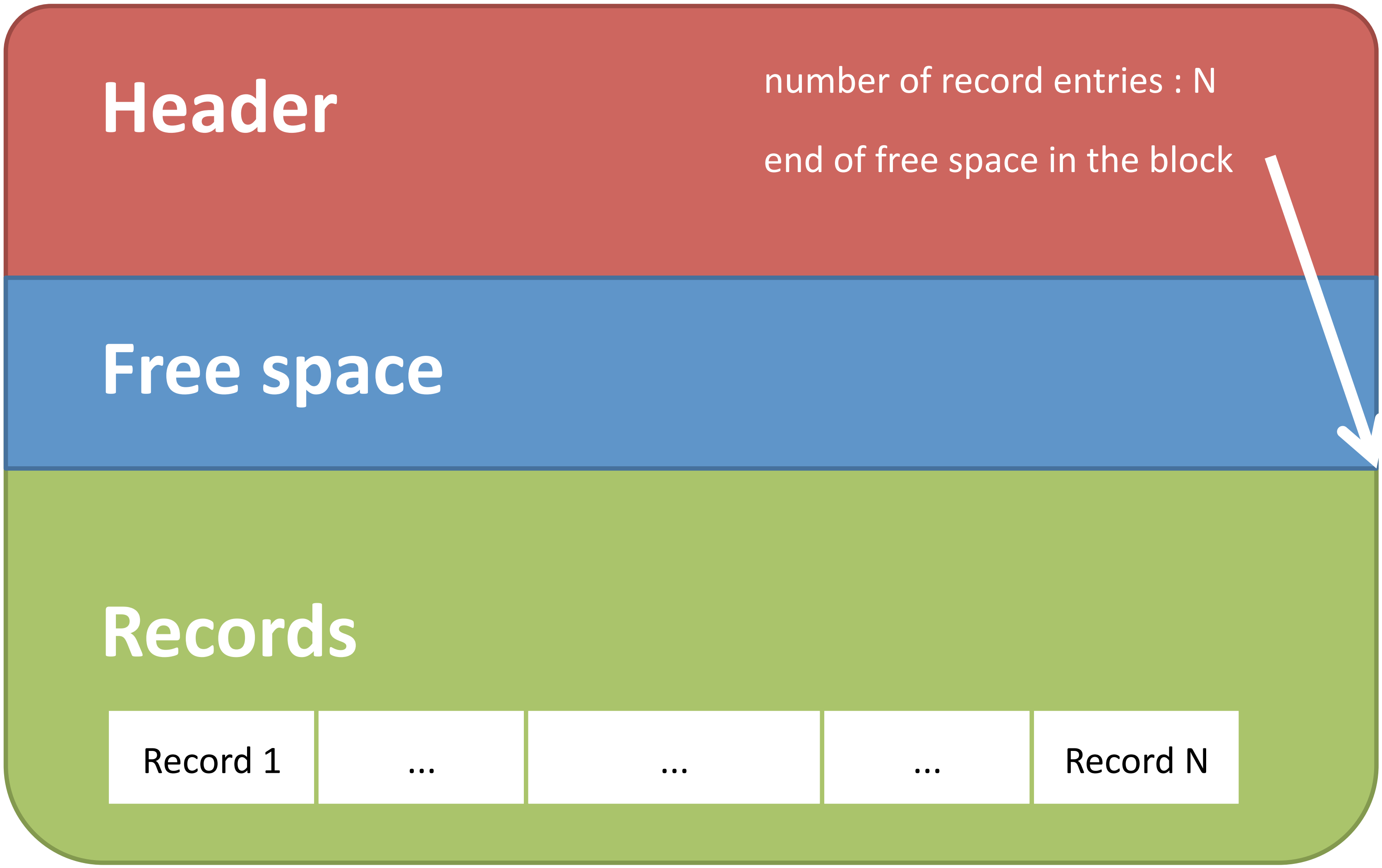
# Block organization



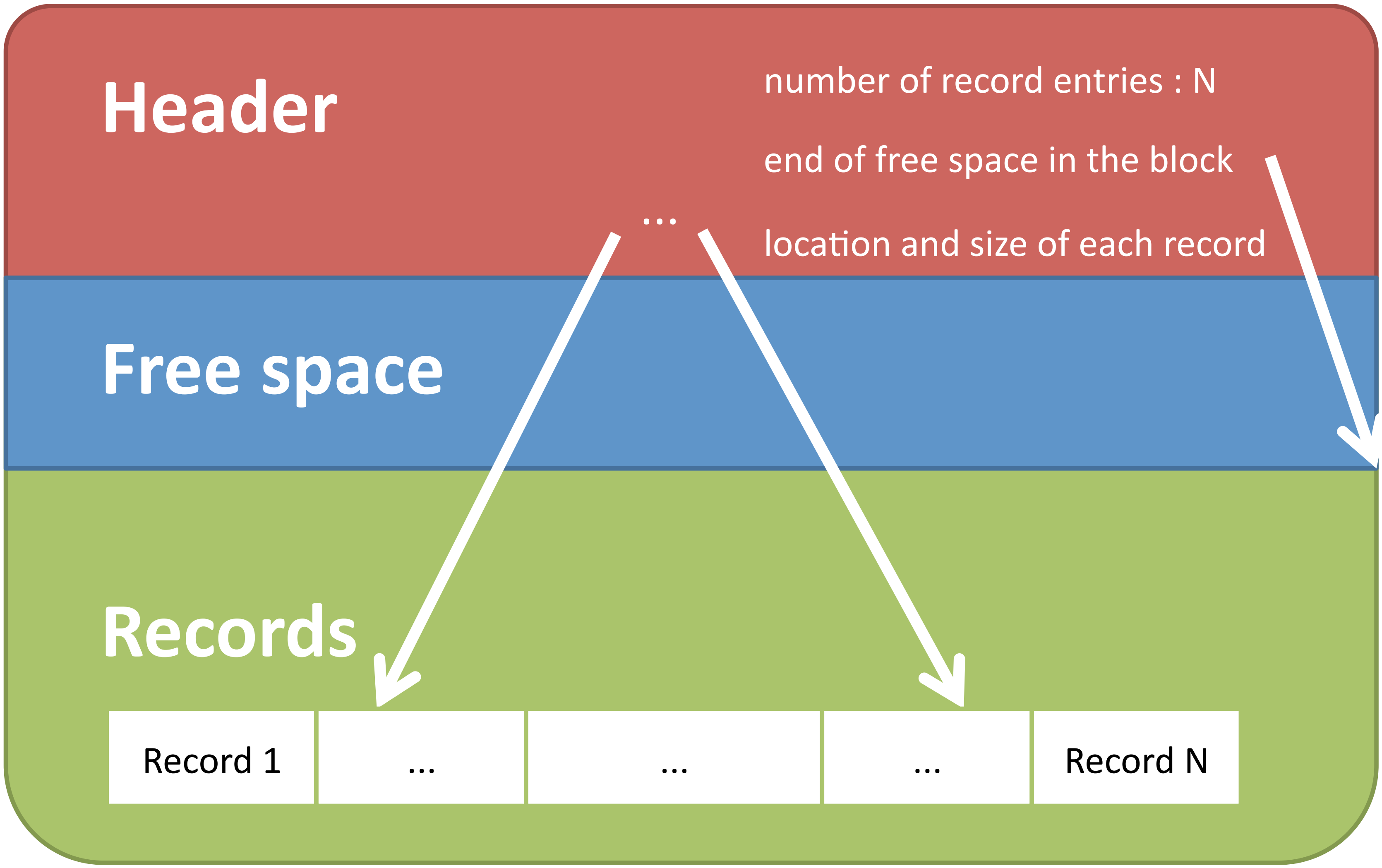
# Block organization



# Block organization



# Block organization



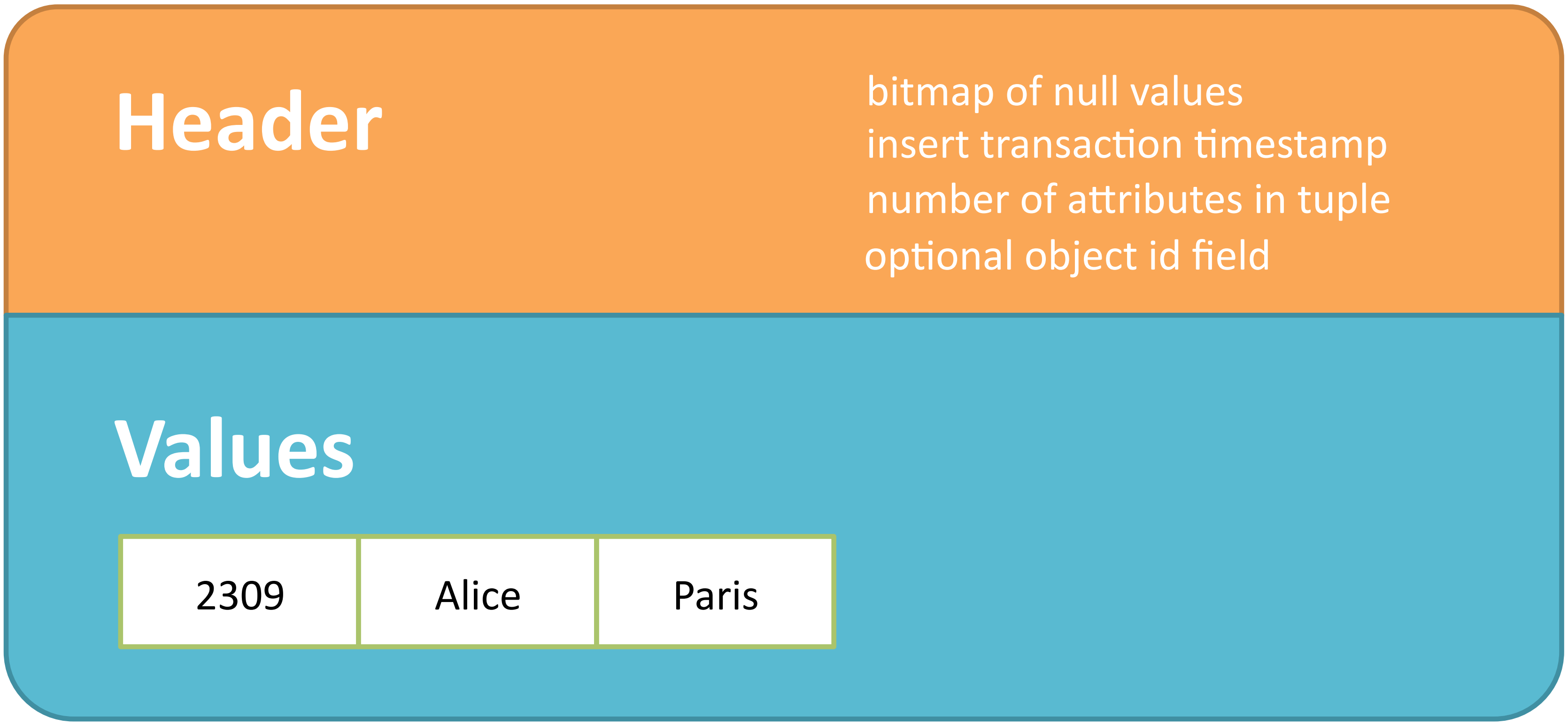
# Block organization

- Records are stored sequentially (row-oriented)
  - but in the last 10 years some type of OLAP systems turned to column-oriented

- Records can be moved around within a page to keep them contiguous with no empty space between them
  - if this happens, entry in the header must be updated.

# Record organization

https://www.postgresql.org/docs/9.0/storage-page-layout.html

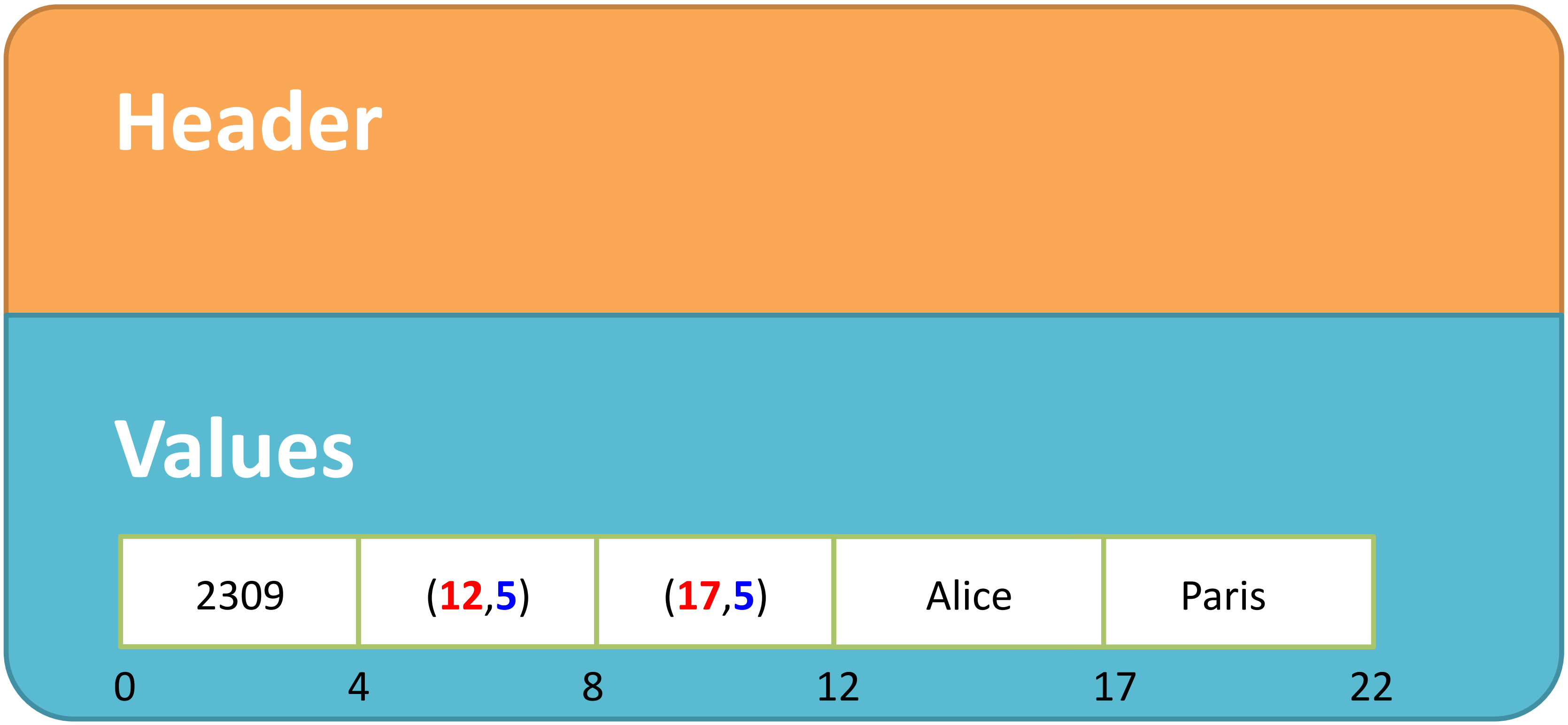



- Postgres : fixed-size header (~23 bytes), followed by optional null bitmap, optional object ID field, user data

# Record organization

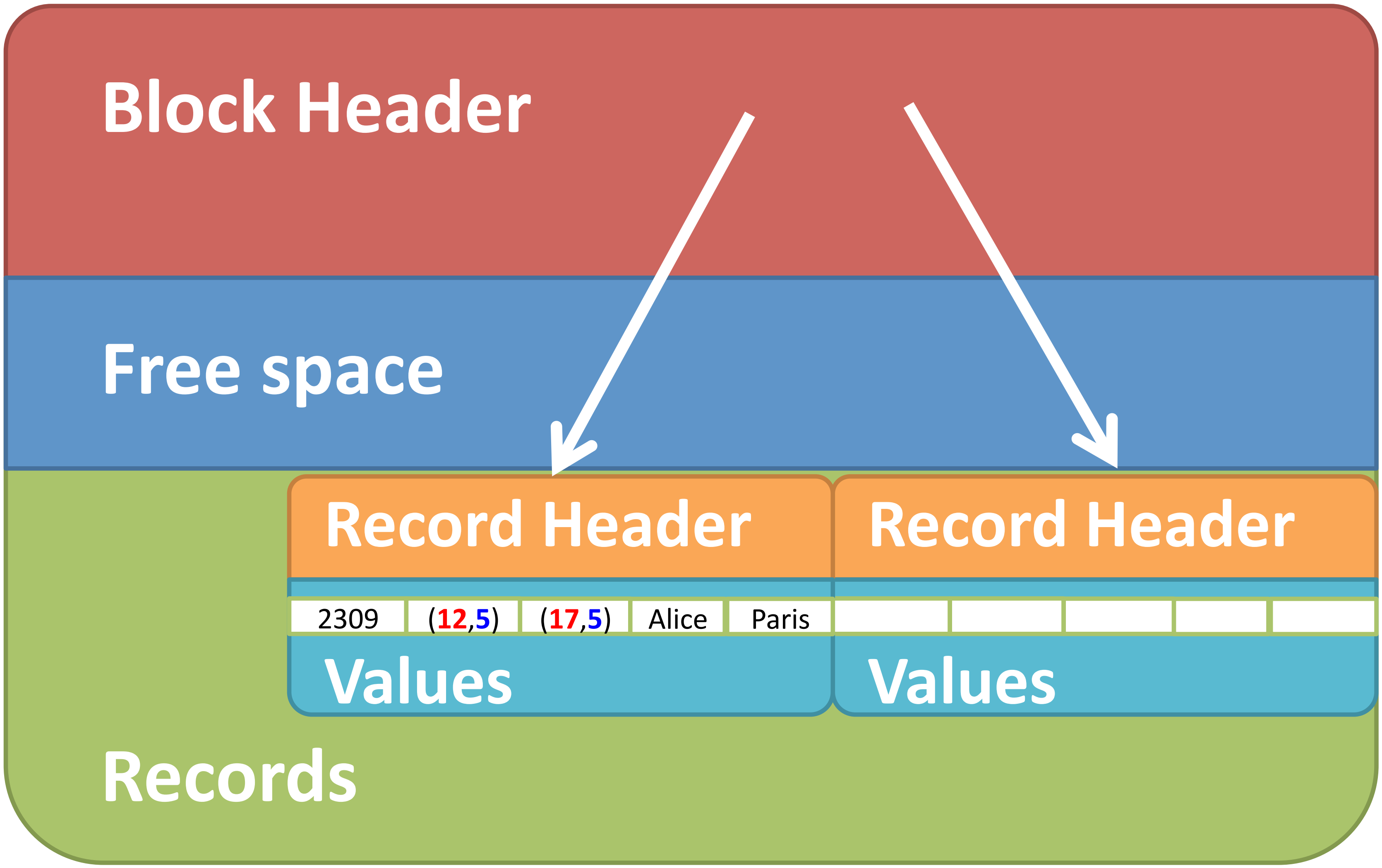
- Most of records are variable length
  - they occur as soon as one uses the ***varchar*** type
- Attributes are stored in order
  - Following the CREATE TABLE statement
- Variable length attributes can be represented by fixed size (offset, length), with actual data stored after all fixed length attributes
  - Efficient for searching a field in the middle of the row

# Record organization



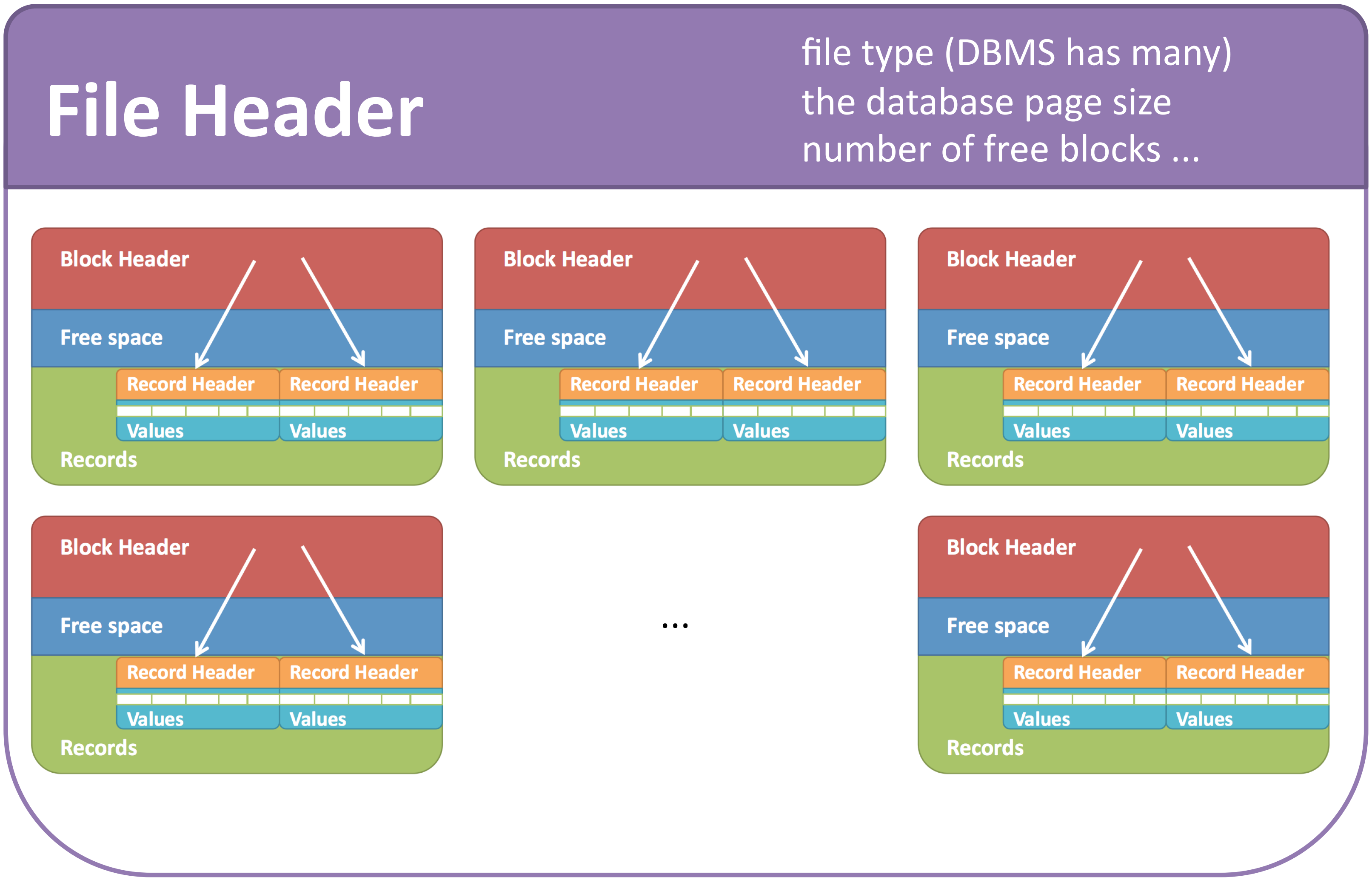

- Variable length attributes represented by fixed size **(offset, length)** with actual data stored after all fixed length attributes

# Summing up

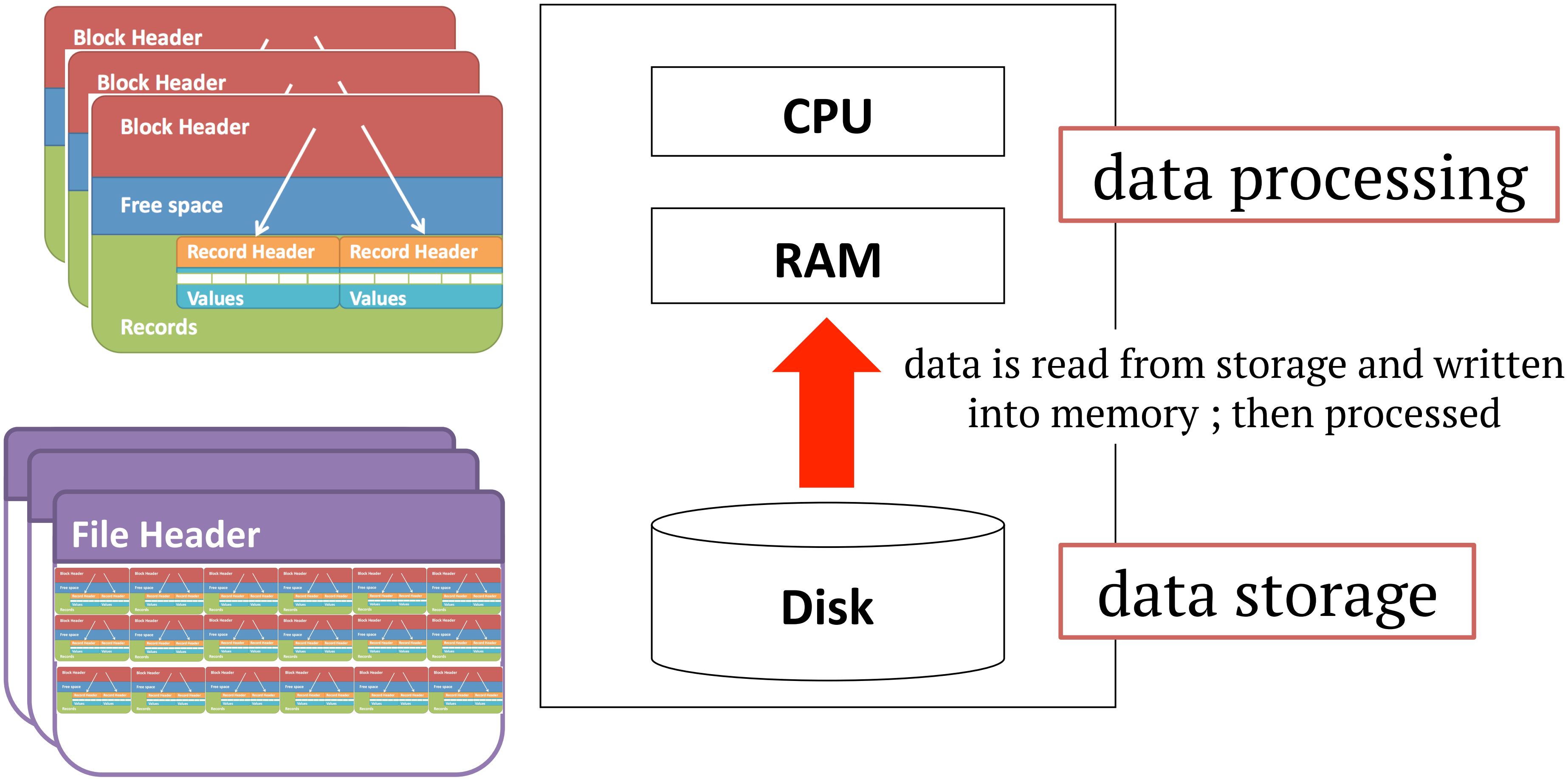

Block Header
Free space
Record Header
Record Header
2309
(12,5)
(17,5)
Alice
Paris
Values
Values
Records

# Summing up

https://www.sqlite.org/fileformat.html#:~:text=The%20first%20100%20bytes%20of%20the%20database%20file%20comprise%20the,first%20(big%2Dendian).



# What happens when we run a query?



# Oracle block size recommendations

http://www.dba-oracle.com/s_oracle_blockk_size.htm

| **2 KB or 4 KB : for online transaction processing (OLTP) or mixed workload<br>8 KB, 16 KB, or 32 KB : for decision support system / OLAP workload environments** | |
|---|---|
| Smaller block size | Larger block size |
| •Good for small rows with lots of random access.<br>•Reduces block contention | •Has lower overhead, so there is more room to store data.<br>•Permits reading several rows into the buffer cache with a single I/O (depending on row size and block size).<br>•Good for sequential access or very large rows (such as LOB data). |
| •Has relatively large space overhead due to metadata (that is, block header).<br>•Not recommended for large rows. There might only be a few rows stored for each block, or worse, row chaining if a single row does not fit into a block | •Wastes space in the buffer cache, if you are doing random access to small rows and have a large block size. For example, with an 8 KB block size and 50 byte row size, you waste 7,950 bytes in the buffer cache when doing random access.<br>•Not good for index blocks used in an OLTP environment, because they increase block contention on the index leaf blocks. |

# Buffer Management

Things we did not cover

- Buffering
  - part of memory holding blocks
- Buffer management
  - block-replacement policies (LRU/MRU, etc..)

# Summing up

- Databases physical organization store records in **blocks** that are moved from disk to memory
- Performances depend on block movement
- Factors that impact block movement are :
  – Of course, DBMS architecure (system)
  – The type of query (user)
  – The relational schema design (user)
    - We will see the importance of "star-schemas"
  – Tuning (eg., indexes) (DB admin)
  – Optimizations (system)

# Types of Queries

- *Not all queries are equal.* They may differ by:
  - Result cardinality (number of answers)
  - Selectivity (fraction of data really needed for evaluation)
  - Complexity (number of joins / conditions / nesting...)

- *Different applications, different types of queries, different DBMS (relational, datawarehouse, NOSQL, Hadoop, etc)*