THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

9. DBMS Internals

CSCI 2541 Database Systems & Team Projects

Gabe

Working from material by Wood & Chaufournier

Today & Upcoming...

Today

- DB Internals
- Midterm – column names
- @ your mentor in your discord channel if you want feedback

Wednesday:

- Using AWS RDS with your project
- Mentor meeting!!!
    - Everyone should understand the whole project (code reviews) – we will ask everyone questions, and ask everyone to do tasks
    - Come prepared to demo efficiently (max 5 min) what you have; grading on three dimensions:
        i. Progress
        ii. Plan
        iii. Teamwork
    - Be prepared for questions. Answer directly, and don't feel bad if we cut you off.

Library Usage

For your project you **may** use…

- Anything in the standard python library
- Form helper libraries like Flask-WTF
- Login libraries like Flask-login
- CSS/HTML libraries like Bootstrap
- Javascript libraries like jquery

You may not use…

- Libraries which fully abstract away database operations (e.g., object relational mapping / ORM libraries)
- A framework other than Flask

If you aren't sure, ask me!

Meet your mentor

Advising

- Kevin (1-4) and Lucas (5-7)

Applications

- Jeet (8-10) and Cat (11-14)

Registration

- Billy (15-18) and Ethan (19-22)

Today's meeting:

- How will you organize your repository and use Git?
- How will you plan your project using Agile?
- *When not meeting*: start planning tasks!

DBMS Internals

DBMS

A database management system provides efficient, convenient, and safe multi-user storage and access to massive amounts of persistent data.

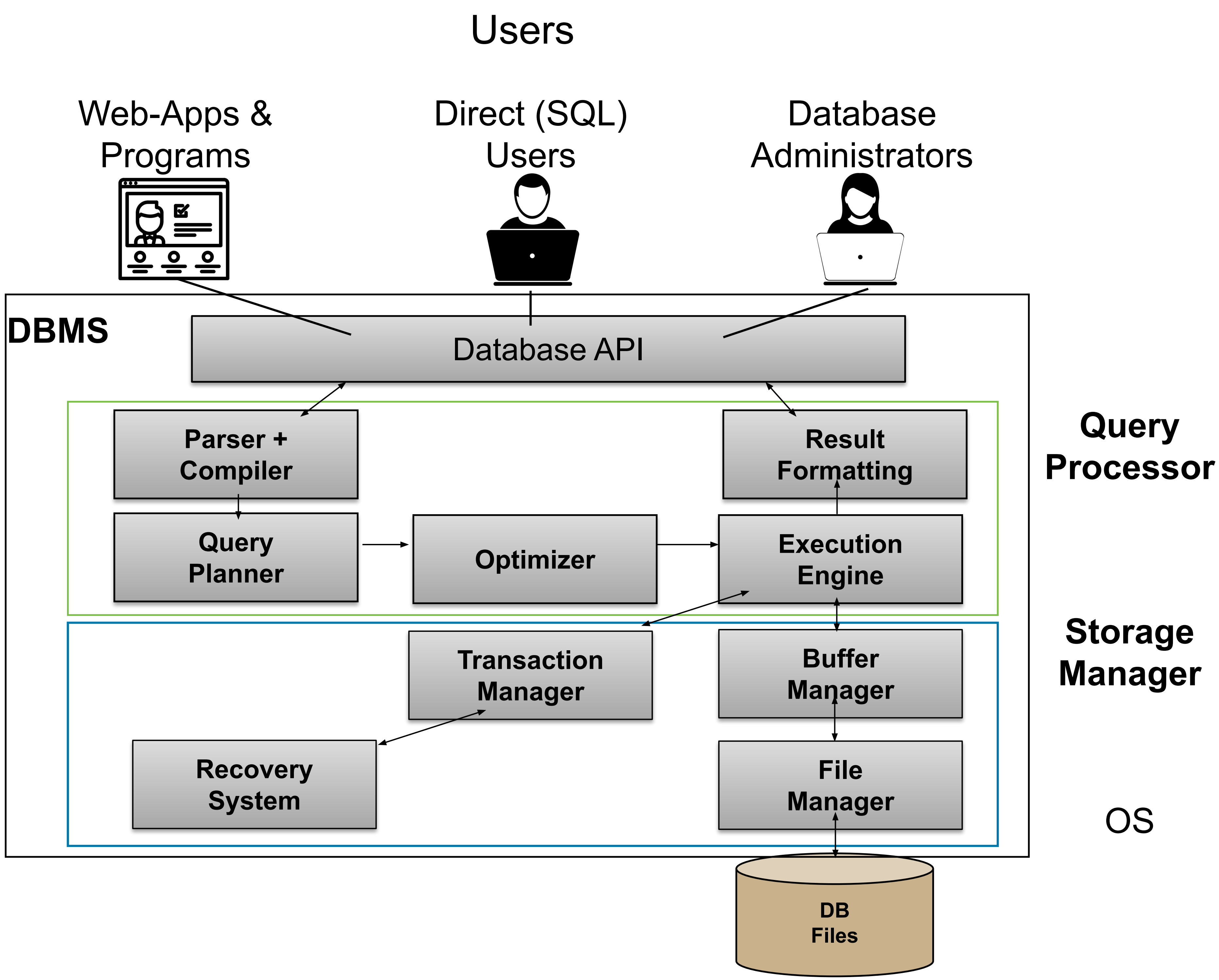
- **Efficient** - Able to handle large data sets and complex queries without searching all files and data items.
- **Convenient** - Easy to write queries to retrieve data.
- **Safe** - Protects data from system failures and hackers.
- **Massive** - Database sizes in gigabytes/terabytes/petabytes.
- **Persistent** - Data exists after program execution completes.
- **Multi-user** - More than one user can access and update data at the same time while preserving consistency…. concept of **transactions**

Components of a DBMS

A DBMS is a complicated software system containing many components:

- **Query processor** - translates user/application queries into low-level data manipulations
  - Sub-components: query parser, query optimizer
- **Storage manager** - maintains storage information including memory allocation, buffer management, and file storage
  - Sub-components: buffer manager, file manager
- **Transaction manager** - performs scheduling of operations and implements concurrency control algorithms
  - You will learn more about storage management and concurrency in the Operating Systems course… enjoy!

DBMS Architecture: Complete Picture



Structure that is **independent** of the underlying file formats

Queries to flexibly read, update, and delete information

Transactions that provide **multi-user consistency**

Storage and Organization: Overview

A database system relies on the operating system to store data on storage devices.

Database performance depends on:

- Properties of storage devices
- How devices are used and accessed via the operating system

Quick look into techniques for storing and representing data

- These apply for SQL as well as NoSQL systems
- Key in efficient storage and retrieval systems
  - Including search engines and big data analytics

Review (?) from architecture: Memory Definitions

What is **(Temporary) Memory**?

What is **Permanent/Persistent/Non-volatile Memory**?

What is **Cache Memory**?

Review (?) from architecture: Memory Definitions

Temporary memory retains data only while the power is on.

- Also referred to as **volatile** storage.
- e.g. dynamic random-access memory (DRAM) (main memory)

Permanent memory stores data even after the power is off.

- Also referred to as non-volatile storage or secondary storage
- e.g. flash memory, SSD, hard drive, DVD, tape drives

Cache is faster memory used to store a subset of a larger, slower memory for performance.

- processor cache (Level 1 & 2), disk cache, network cache

Physical Storage: Memory Hierarchy

Primary Storage: cache & main memory

- Can be directly accessed by CPU
- Currently used data

Secondary Storage: flash, SSD, magnetic disks, optical disks, tapes

- Larger capacity, low cost, slow access
- Cannot be directly processed by CPU

DB stores large amount, persist over time

- Data is stored in secondary storage
- Contrast with run-time data structures

Time taken to fetch data depends on how data is organized on disk/file

DBMS storage

Why not store everything in Main Memory (DRAM)?

DBMS storage

Why not store everything in Main Memory (DRAM)?

Costs too much.

Main memory is volatile.

- We want data to be saved between runs. (Obviously!)
- Situations that cause permanent loss of data occur less frequently in disks than primary memory
- Disk/Flash storage is non-volatile

Magnetic Hard Disks

Secondary storage device of choice for BIG data.

Main advantage over tapes: *random access* vs. *sequential*.

Data is stored and retrieved in units called *disk blocks* or *pages*.

Unlike RAM, time to retrieve a disk page varies depending upon location on disk.

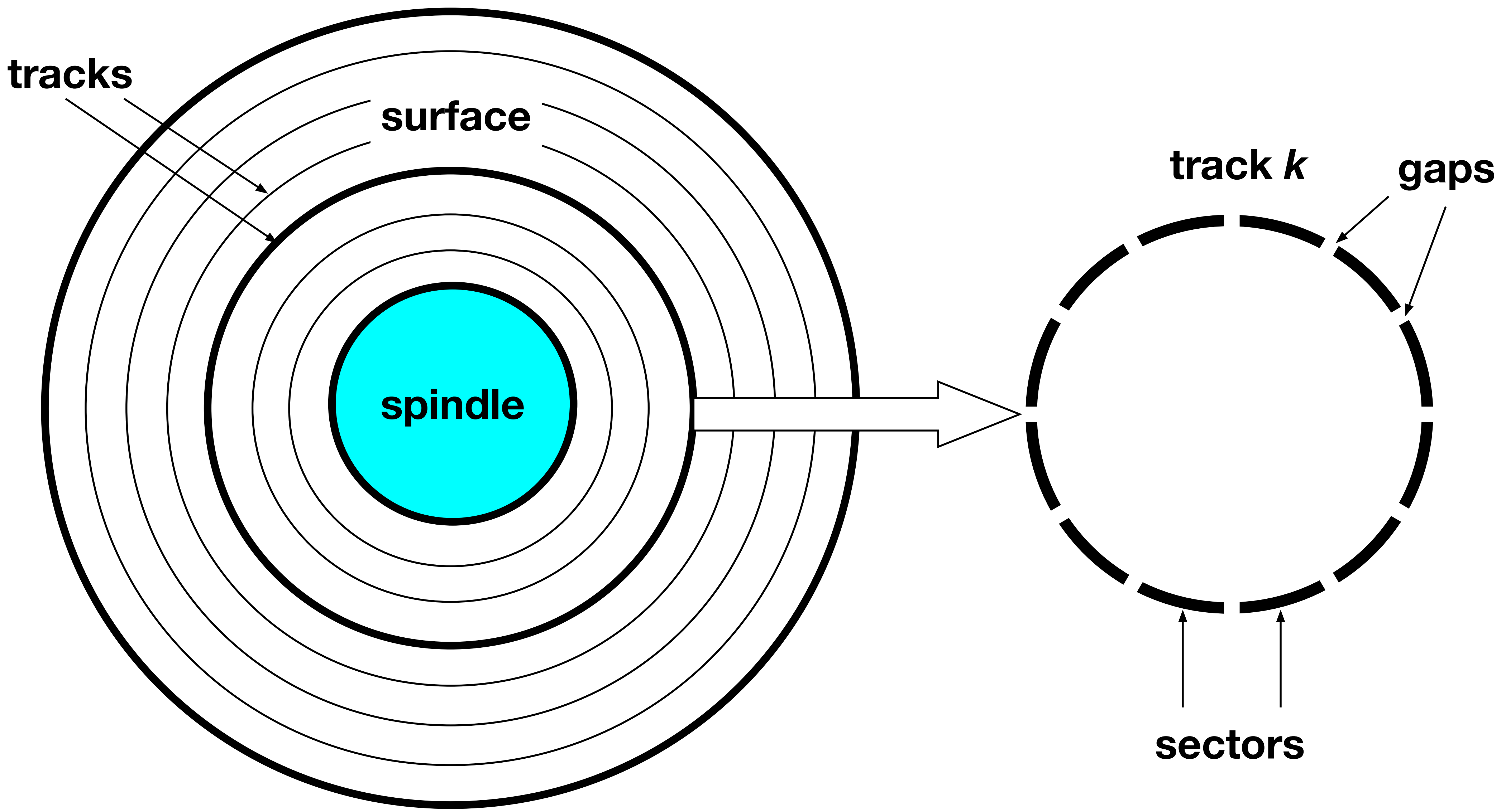
- Therefore, relative placement of pages on disk has major impact on DBMS performance!

Disk Geometry

Disks consist of **platters**, each with two **surfaces**.

Each **surface** consists of concentric rings called **tracks**.

Each **track** consists of **sectors/blocks** separated by **gaps**.



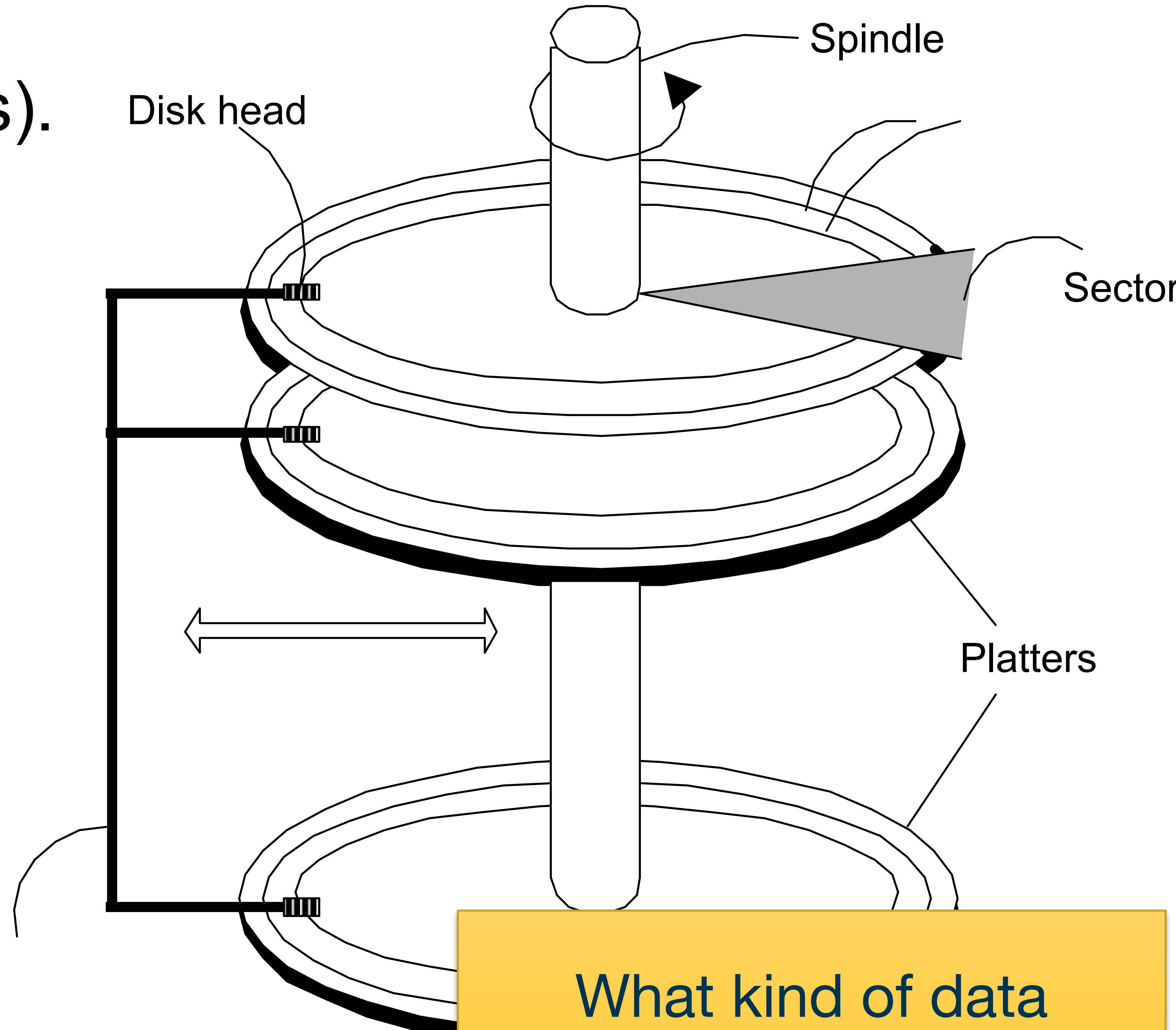
Components of a Disk

The platters spin (say, 90rps).

The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder*

Only **one** head reads/writes at any one time.

Block size is a multiple of *sector size* (which is fixed).



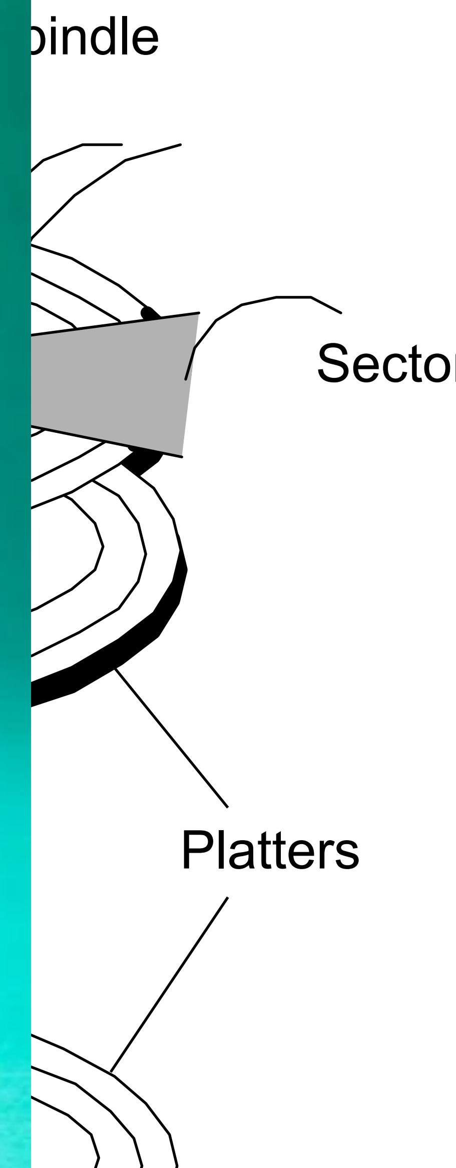
What kind of data accesses will be fastest?

Components of a Disk

T

T
ir
o
u
c

C
re
o



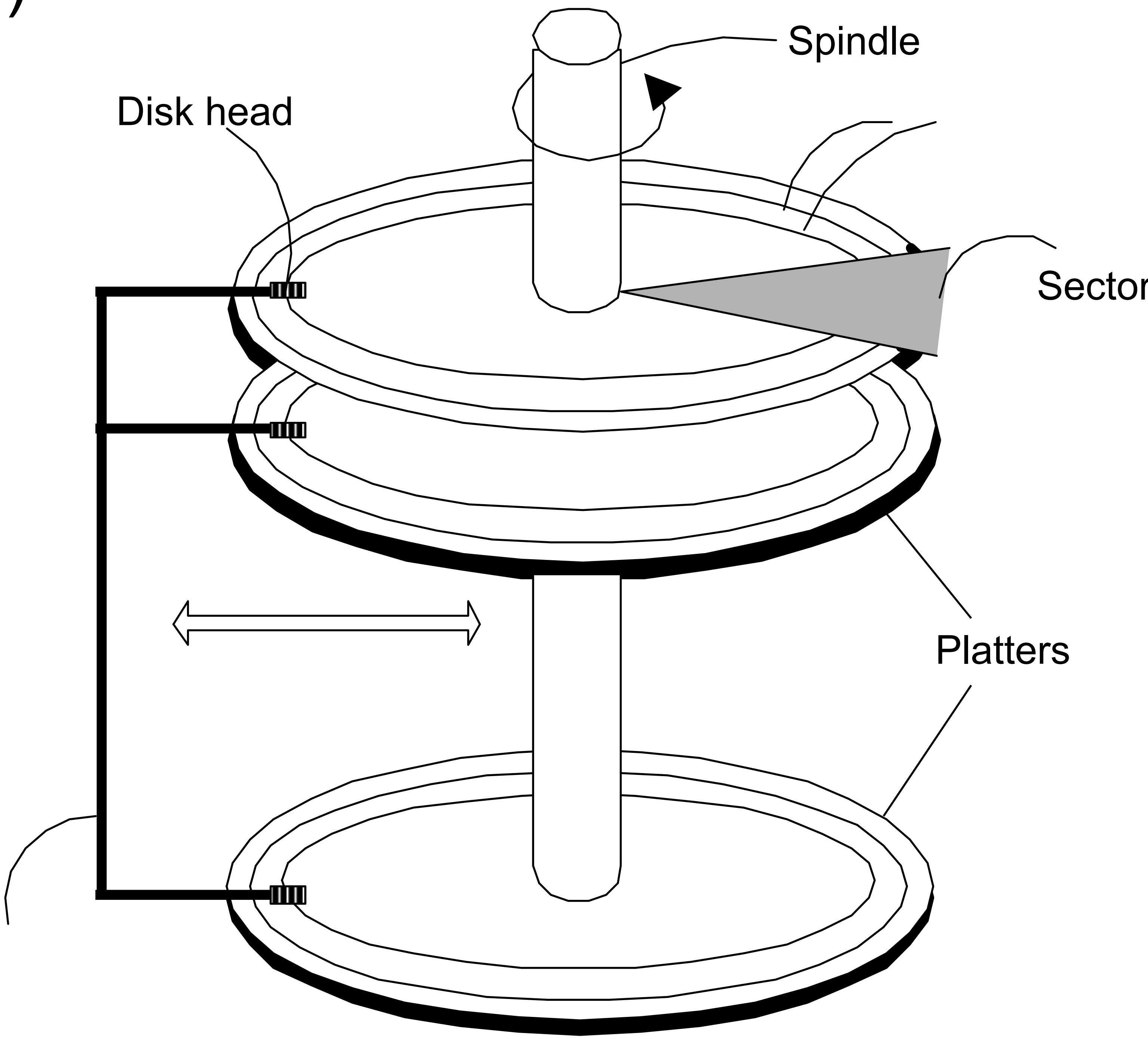
B
of *sector size* (which is fixed).

of data
be fastest?

Accessing a Disk Page

Time to access (read/write) a disk block:

What physically must happen to read?



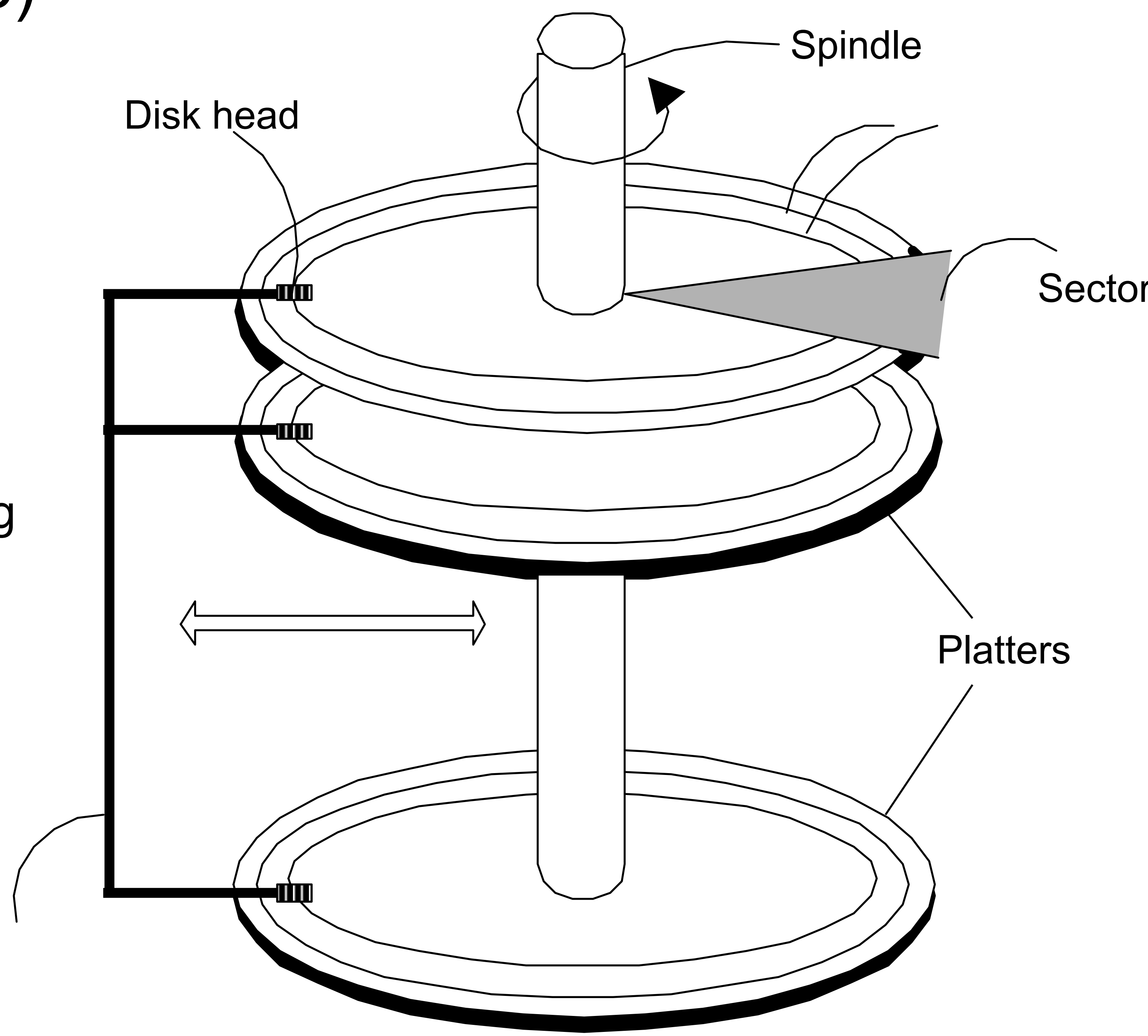
Accessing a Disk Page

Time to access (read/write) a disk block:

- *seek time* (moving arms to position disk head on track)
- *rotational delay* (waiting for block to rotate under head)
- *transfer time* (actually moving data to/from disk surface)

Seek time and rotational delay dominate.

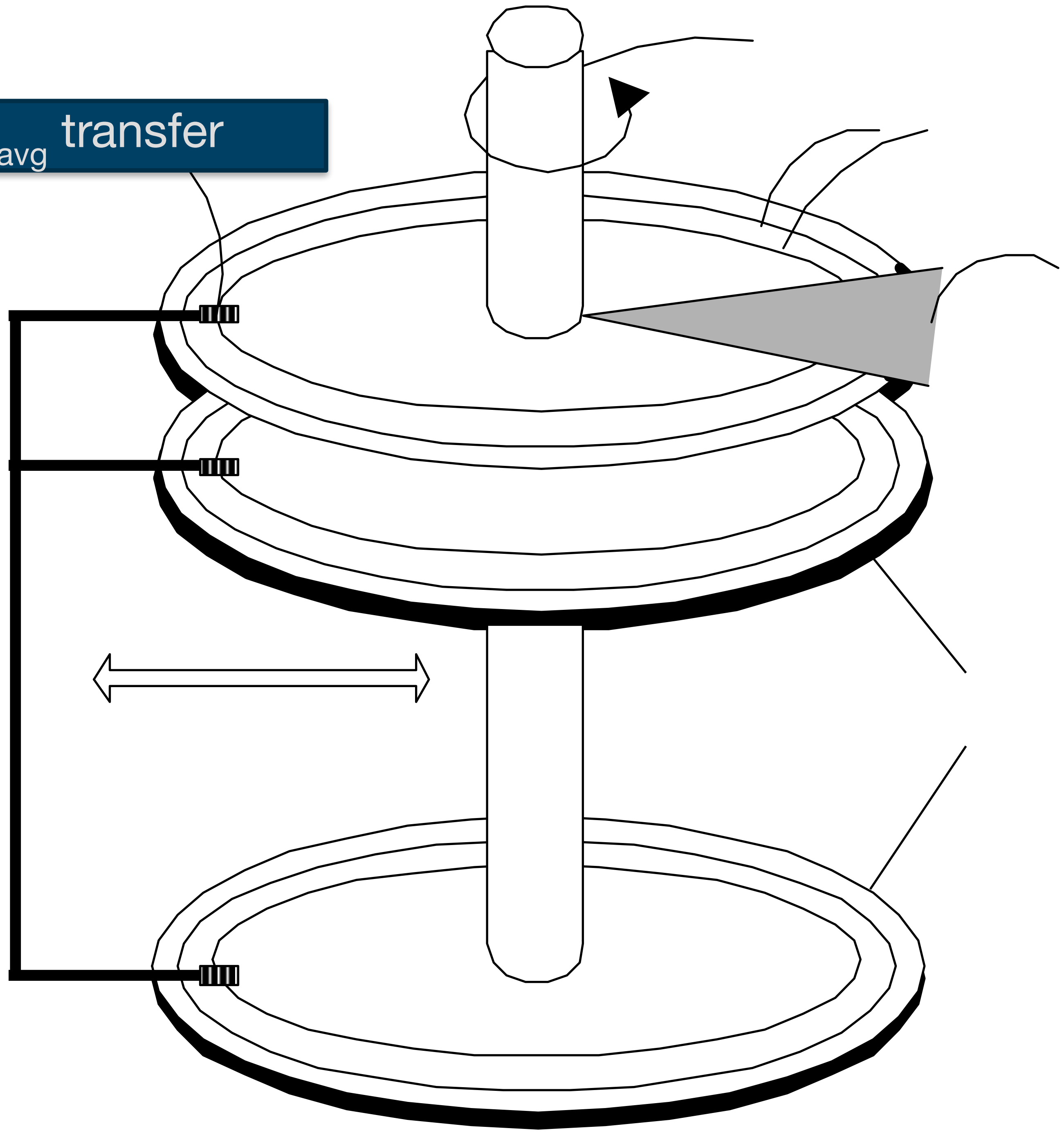
Key to lower I/O cost: **reduce seek/rotation delays!**



Disk Access Times

Average time to access a target sector approximated by :

$T_{access} = T_{avg}\ seek + T_{avg}\ rotation + T_{avg}\ transfer$



Seek time (Tavg seek)

- Time to position heads over cylinder containing target sector.
- Typical Tavg seek = 9 ms

Rotational latency (Tavg rotation)

- Time waiting for first bit of target sector to pass under r/w head.
- Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min = 6 ms

Transfer time (Tavg transfer)

= ~200 MB/sec

- Time to read the bits in the target sector.

Accessing Data

SELECT * FROM EMP;

Need to scan entire file

- Read all records

Access all blocks/pages of the file on the disk

- Assume N pages

$$T_{access} = T_{avg}\text{ seek} + T_{avg}\text{ rotation} + T_{avg}\text{ transfer}$$

How long does this take ?

Accessing Data

SELECT * FROM EMP;

Need to scan entire file

- Read all records

Access all blocks/pages of the file on the disk

- Assume N pages

How long does this take ?

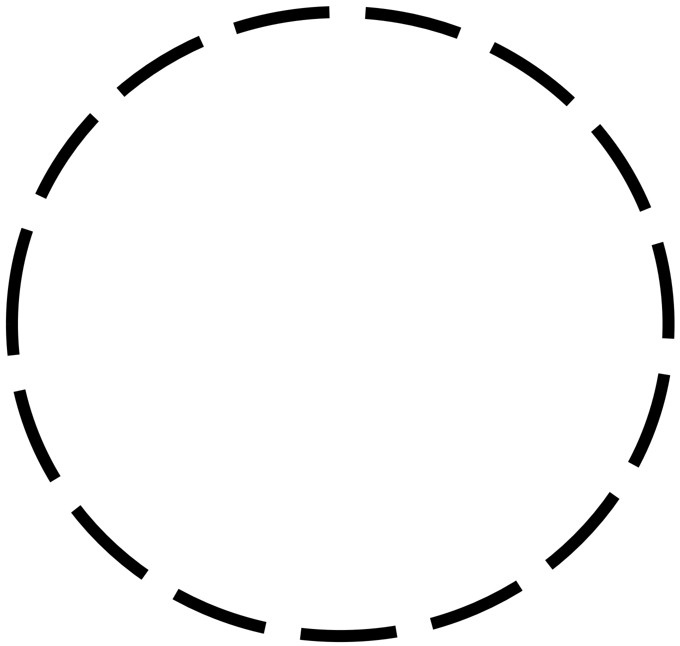
How could we make this more efficient?

Simple approach: N* Taccess

- Taccess = Tavg seek + Tavg rotation + Tavg transfer
- **May need to seek and rotate for every block!**

Impact of Disk Layout

If we can keep the data from a DB in a contiguous region on disk we can eliminate seeks and rotation!



First Block: = Taccess = T_{avg} seek + T_{avg} rotation + T_{avg} transfer

Second Block = T_{avg} transfer

Third block = T_{avg} transfer

...

But…

Unfortunately we don't usually have very much control over exactly where data is located on disk

- When you call write you don't need to specify what platter and track! That would be a pain

Often DBMS just reserve large files to store tables in

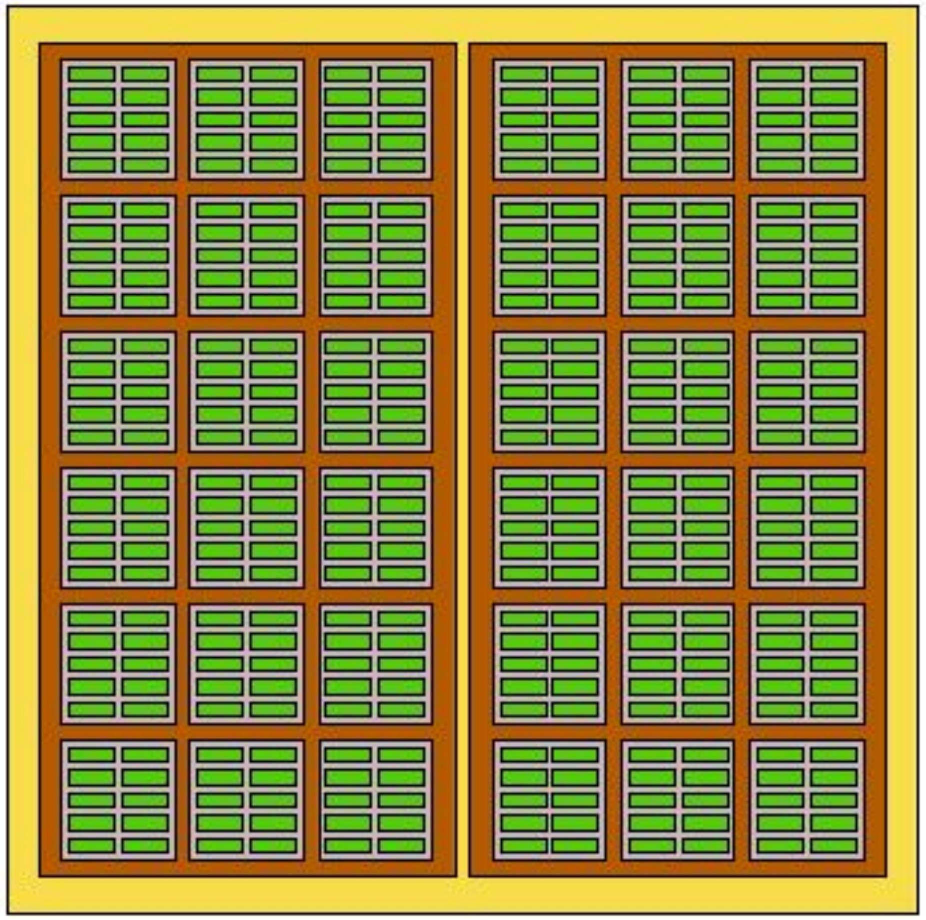
- Assume that the OS File System will lay out those files in contiguous regions
- For really high performance environments, can co-design file system and DBMS!

Solid State Drives: SSDs

Solid State Drives (SSDs) use different technology to store data - flash memory instead of spinning disks

- Data stored in grid of blocks
- Can access blocks directly (no moving parts)
- Similar interface to HDDs: block-level access
- Higher cost and lower capacity
  - HDD: 8TB for $150
  - SDD: 1TB for $250

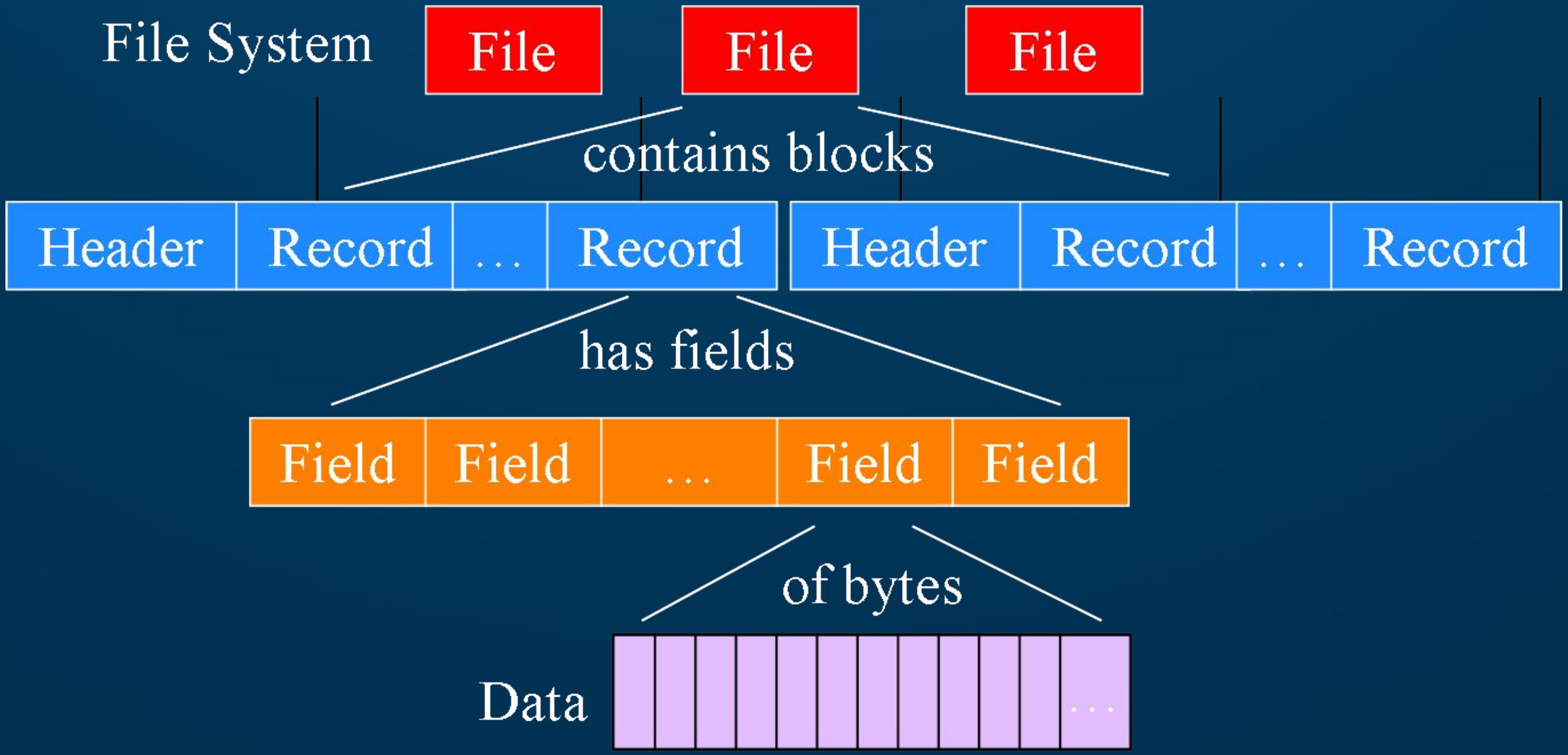
How will this affect DBMS performance?



Representing Data in Databases

A **database** is made up of one or more files.

- Each **file** contains one or more blocks.
- Each **block** has a header and contains one or more records.
- Each **record** contains one or more fields.
- Each **field** is a representation of a data item in a record.



File = Relation; Record = row/tuple; Field = column/attribute

Organization of Records

Record is collection of related information

- Each tuple/row is a record
- each value is one or more bytes, corresponds to a particular field of record
- each field specifies some attribute
- collection of field definitions and their types constitutes record type or format
  - data type associated with each field
- blocks are fixed size, but record sizes vary

Two main types of records:

- Variable length: size of record varies – e.g. w/ VARCHAR
- Fixed length: all records have fixed length – CHAR

Fixed Length Records

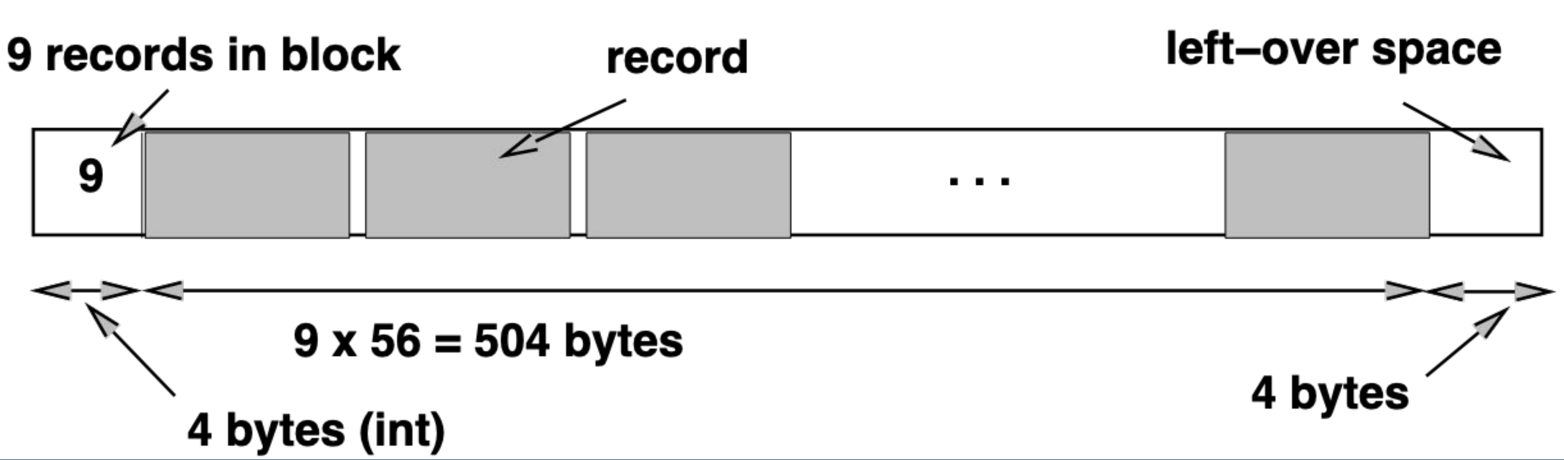
<u>Customer ID</u>	**First Name**	**Surname**	**Birthday**	**Age**	**Fav Color**
123	Pooja	Singh	1/4/1984	37	Blue
456	San	Zhang	3/15/2001	19	Blue
789	John	Zhang	11/12/2006	14	Buff

How should we store a fixed length record?

Fixed Length Records

<u>Customer ID</u>	First Name	Surname	Birthday	Age	Fav Color
123	Pooja	Singh	1/4/1984	37	Blue
456	San	Zhang	3/15/2001	19	Blue
789	John	Zhang	11/12/2006	14	Buff

- Need a fixed size for each field/attribute
- Store the offset from start of record to each field
  - Will be the same for all records in a table



Variable Length Records

<u>Customer ID</u>	First Name	Surname	Birthday	Age	Fav Quote
123	Pooja	Singh	1/4/1984	37	Carpe Diem
456	San	Zhang	3/15/2001	19	To be or not to be
789	John	Zhang	11/12/2006	14	We hold…

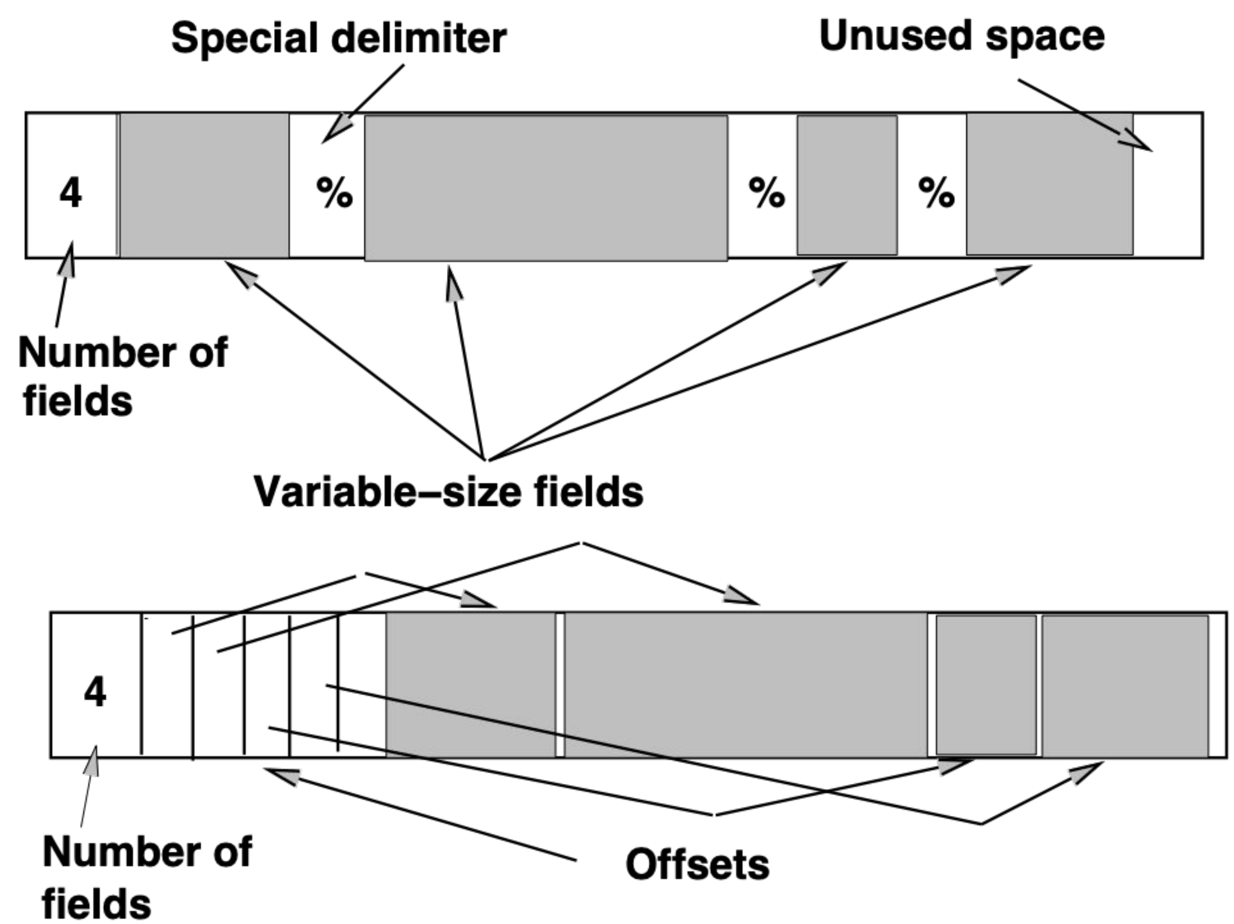
How should we store a variable length record?

Variable Length Records

<u>Customer ID</u>	First Name	Surname	Birthday	Age	Fav Quote
123	Pooja	Singh	1/4/1984	37	Carpe Diem
456	San	Zhang	3/15/2001	19	To be or not to be
789	John	Zhang	11/12/2006	14	We hold…

1) Use a delimiter between each field

2) Store an offset to each field within a record



Record Types

Fixed length vs **Variable length** records

- fixed is easier to implement
- fixed wastes space when block size not multiple of record size

Spanned vs **Unspanned**

- when parts of a record can be placed onto a block, need pointers to next block where remainder of record is placed

Record Layout

How should we store records in a file?

<u>**Customer ID**</u>	**First Name**	**Surname**	**Birthday**	**Age**	**Fav Quote**
123	Pooja	Singh	1/4/1984	37	Carpe Diem
456	San	Zhang	3/15/2001	19	To be or not to be
789	John	Zhang	11/12/2006	14	We hold…
…	…	…	…	…	…

Record Layout

How should we store records in a file?

<u>Customer ID</u>	First Name	Surname	Birthday	Age	Fav Quote
123	Pooja	Singh	1/4/1984	37	Carpe Diem
456	San	Zhang	3/15/2001	19	To be or not to be
789	John	Zhang	11/12/2006	14	We hold…
…	…	…	…	…	…

Heap File: dump all records together in a heap, keep adding new records to the end of the file

- Fast insertion!
- Slow lookups!

Sorted File: carefully store all records in sorted order

- Slow insertion!
- Fast lookups!

DBMS Operations

Queries will require operations on disk

- **Insert** a record
- **Delete** a record
- **Modify** a record
- **Scan** all records
- **Search** for records that satisfy a condition
  - Range Search
  - Equality Search
- **Reorganize** to clean up deleted records
  - Garbage collection

Heap Files

Record are unordered

Insertion?

Deletion?

Search?

Sorted Files

Sort records based on a particular field (primary key?)

Insertion?

Deletion?

Search?

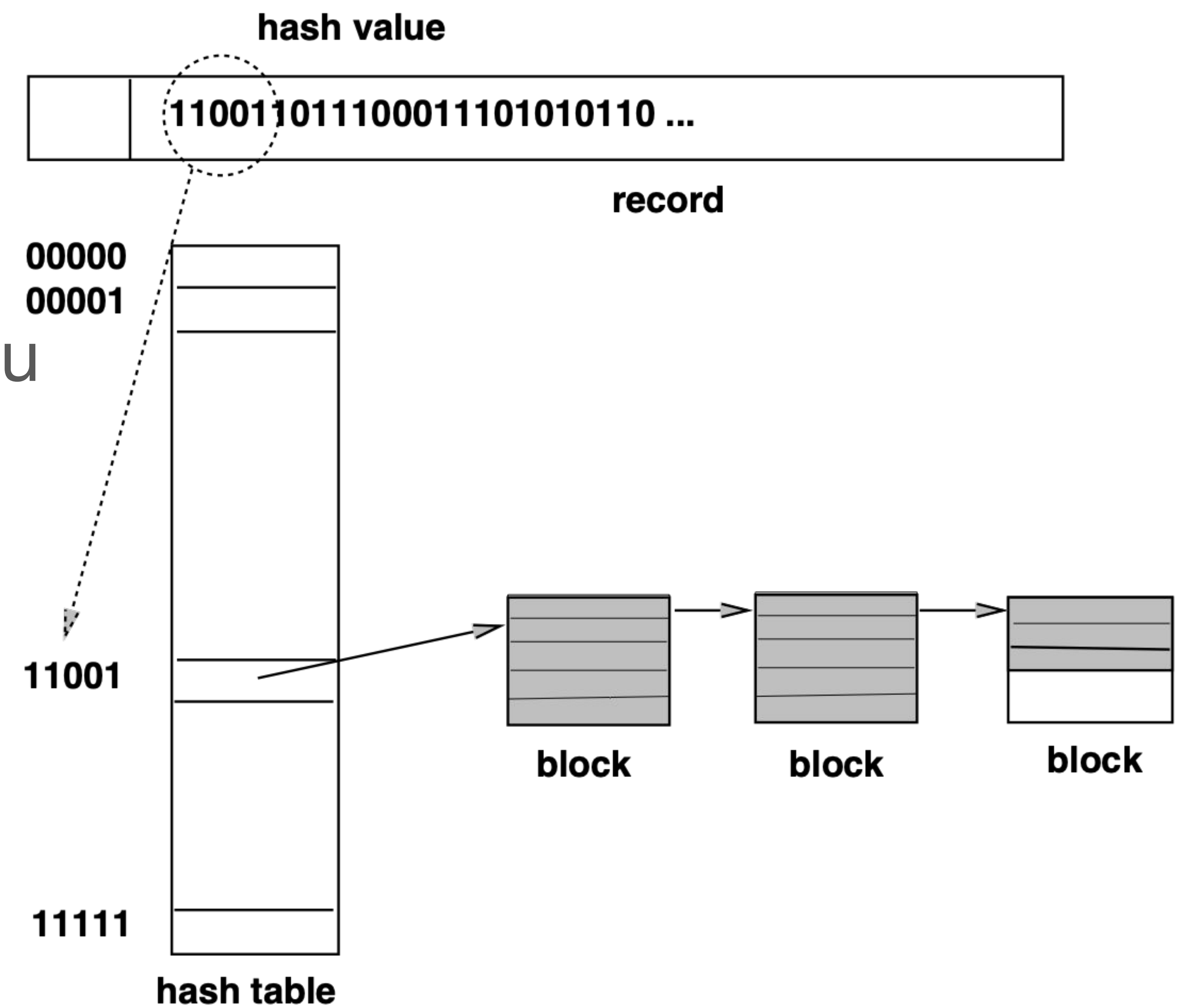
Hashed Files

Distribute records among buckets based on a hash key

- Use hash key to find a bucket of similar records
- Keep adding blocks as you get more records in that bucket

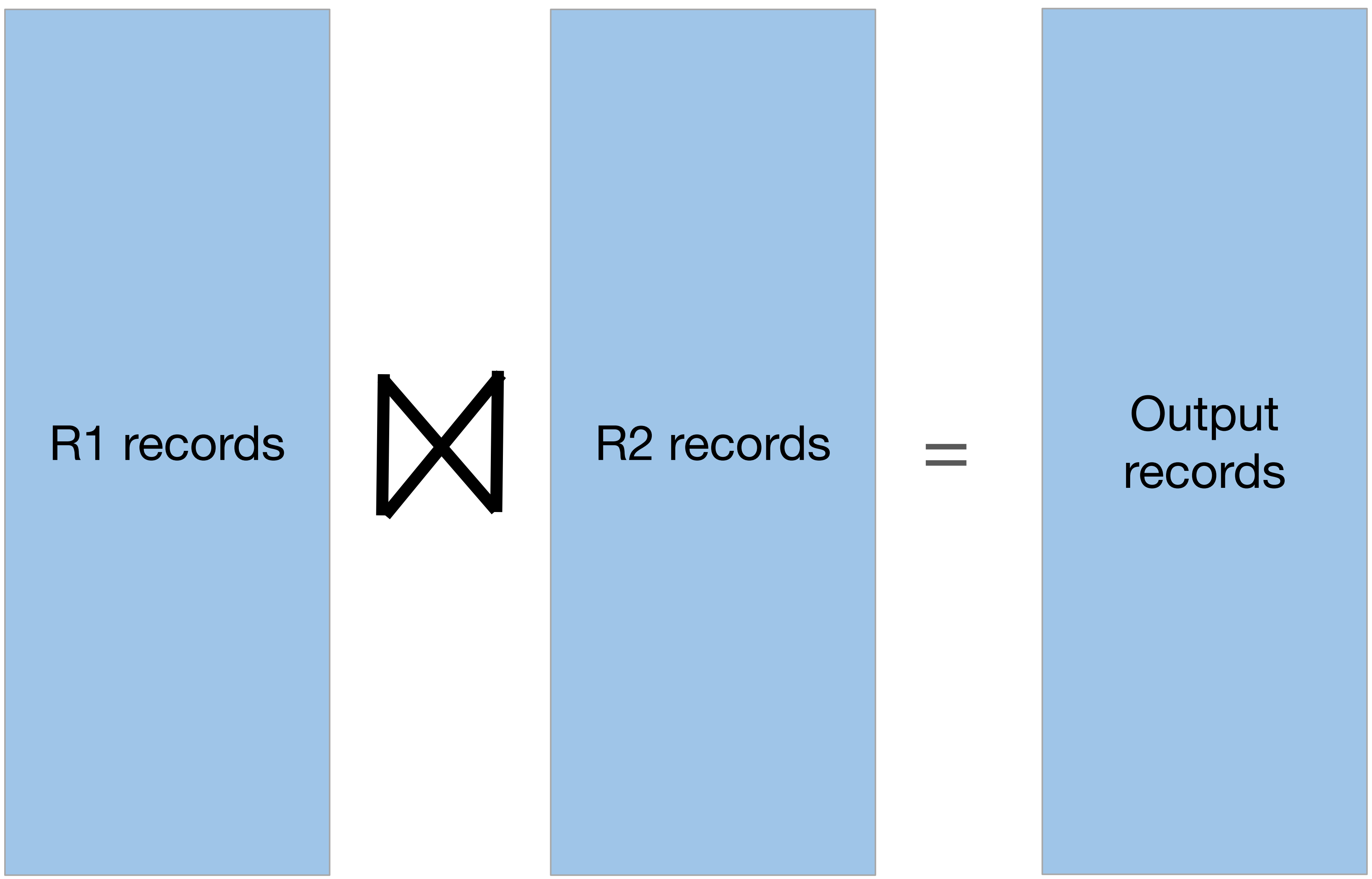
What kind of search can this help with?

- Range search?
- Equality search?



Join Implementation

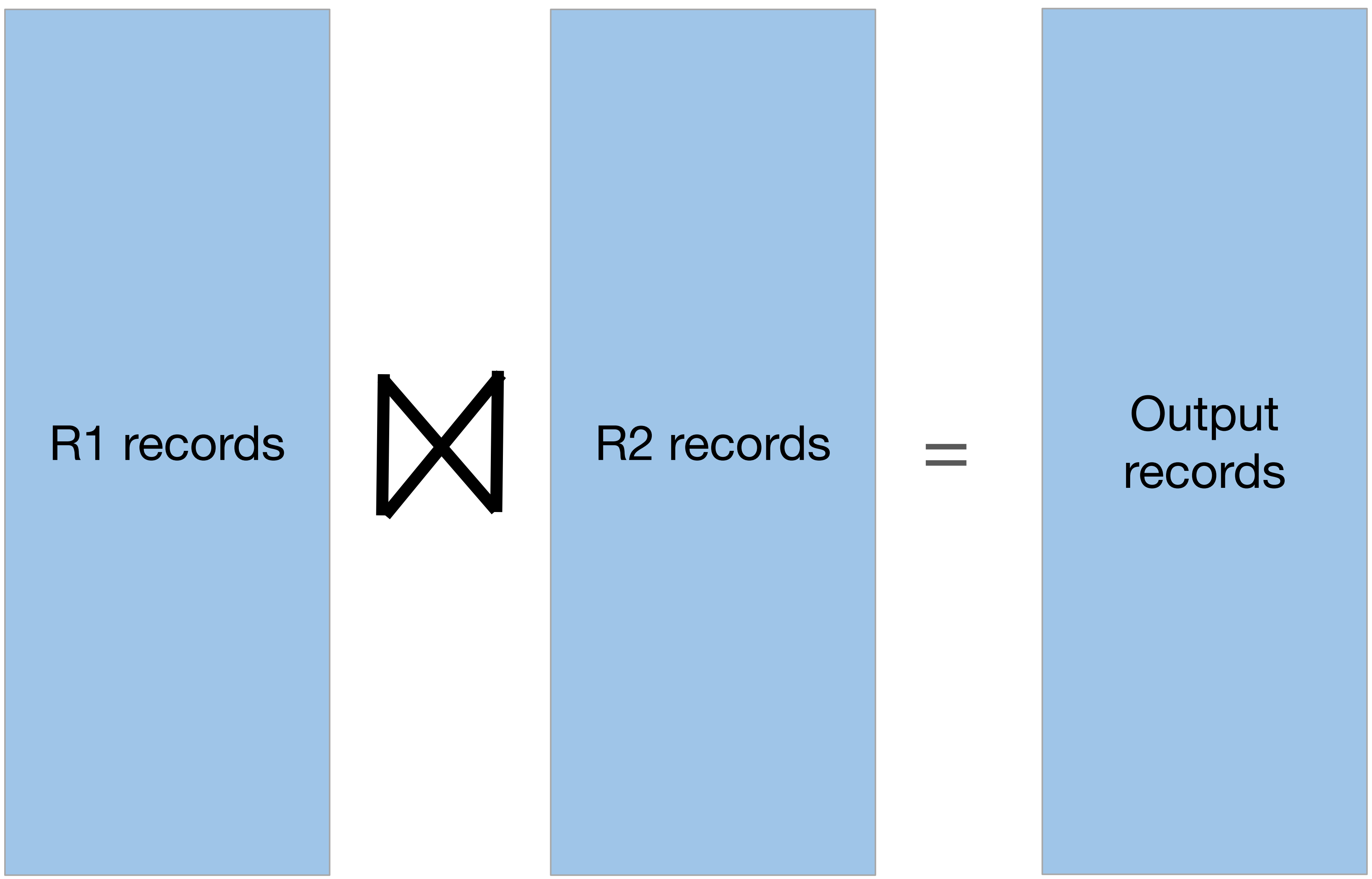
SELECT x from R1 JOIN R2 ON R1.a = R2.a



Implementation?

Join Implementation

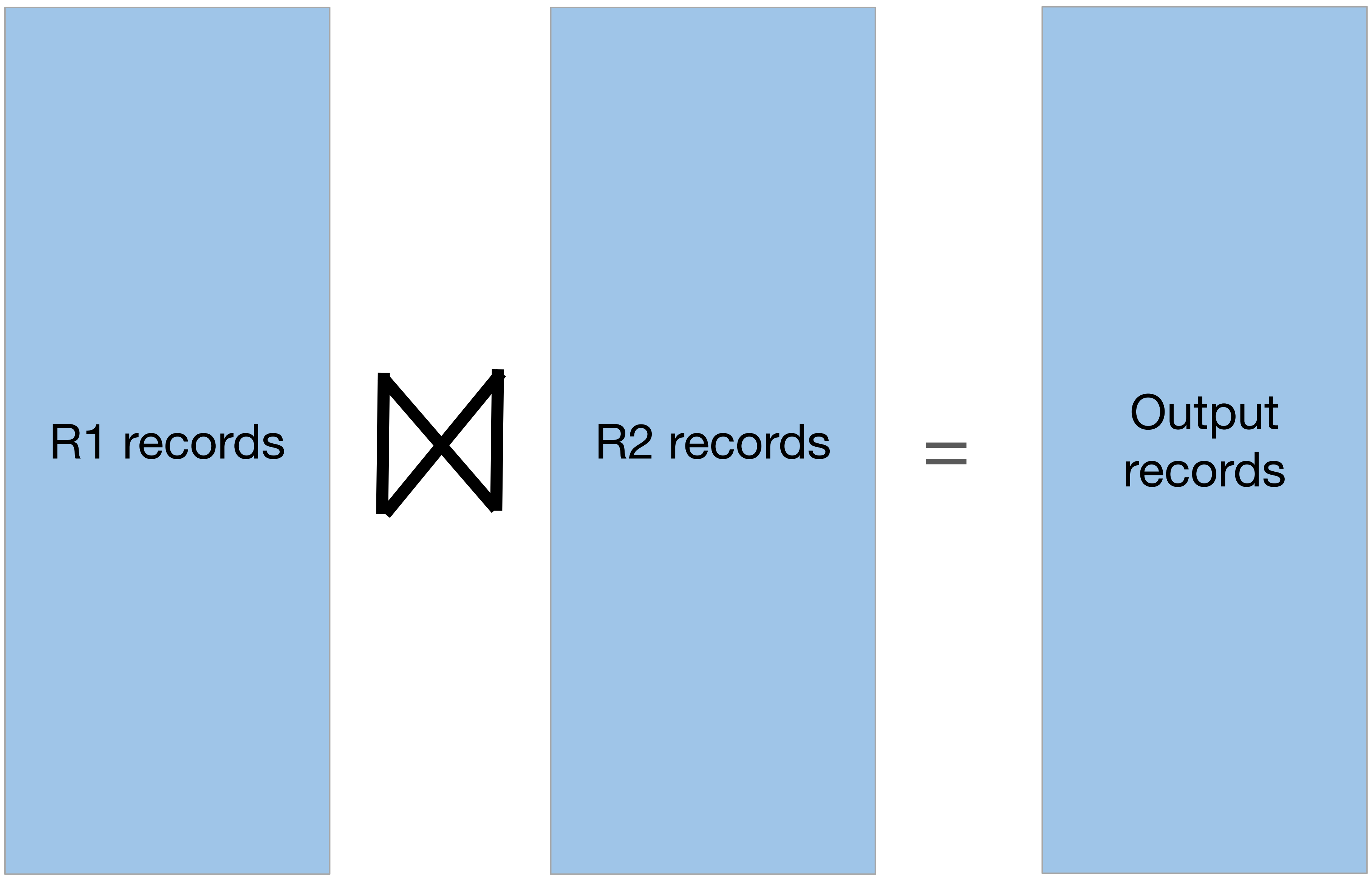
SELECT x from R1 JOIN R2 ON R1.a = R2.a




```
for r1 in records(R1):
    for r2 in records(R2):
        if r1.a = r2.a:
            add(OUT, r1.x)
```

Join Implementation

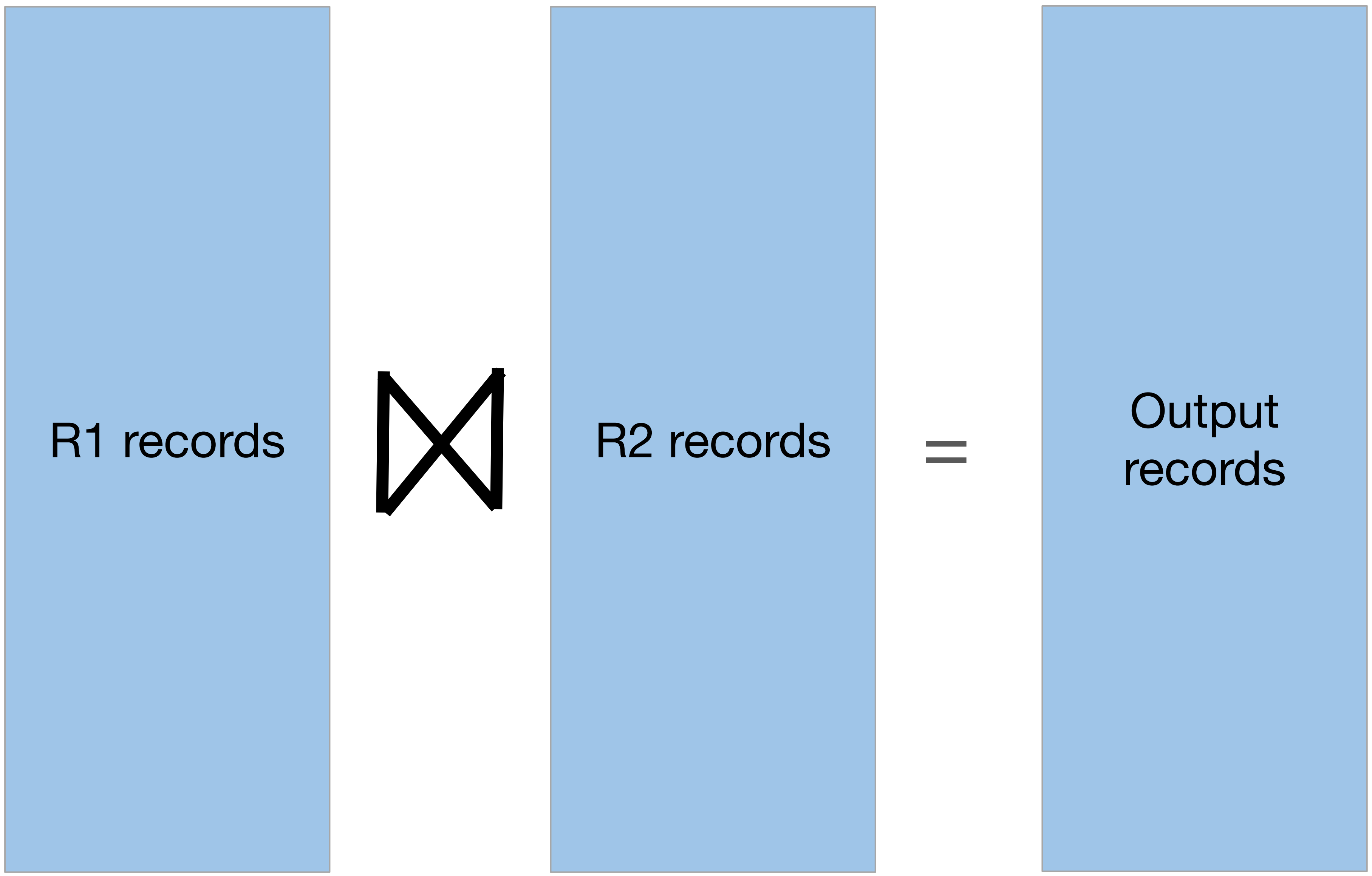
SELECT x from R1 JOIN R2 ON R1.a = R2.a




```
for r1 in records(R1):
    for r2 in records(R2):
        if r1.a = r2.a:
            add(OUT, r1.x)
```

Join Implementation

SELECT x from R1 JOIN R2 ON R1.a = R2.a

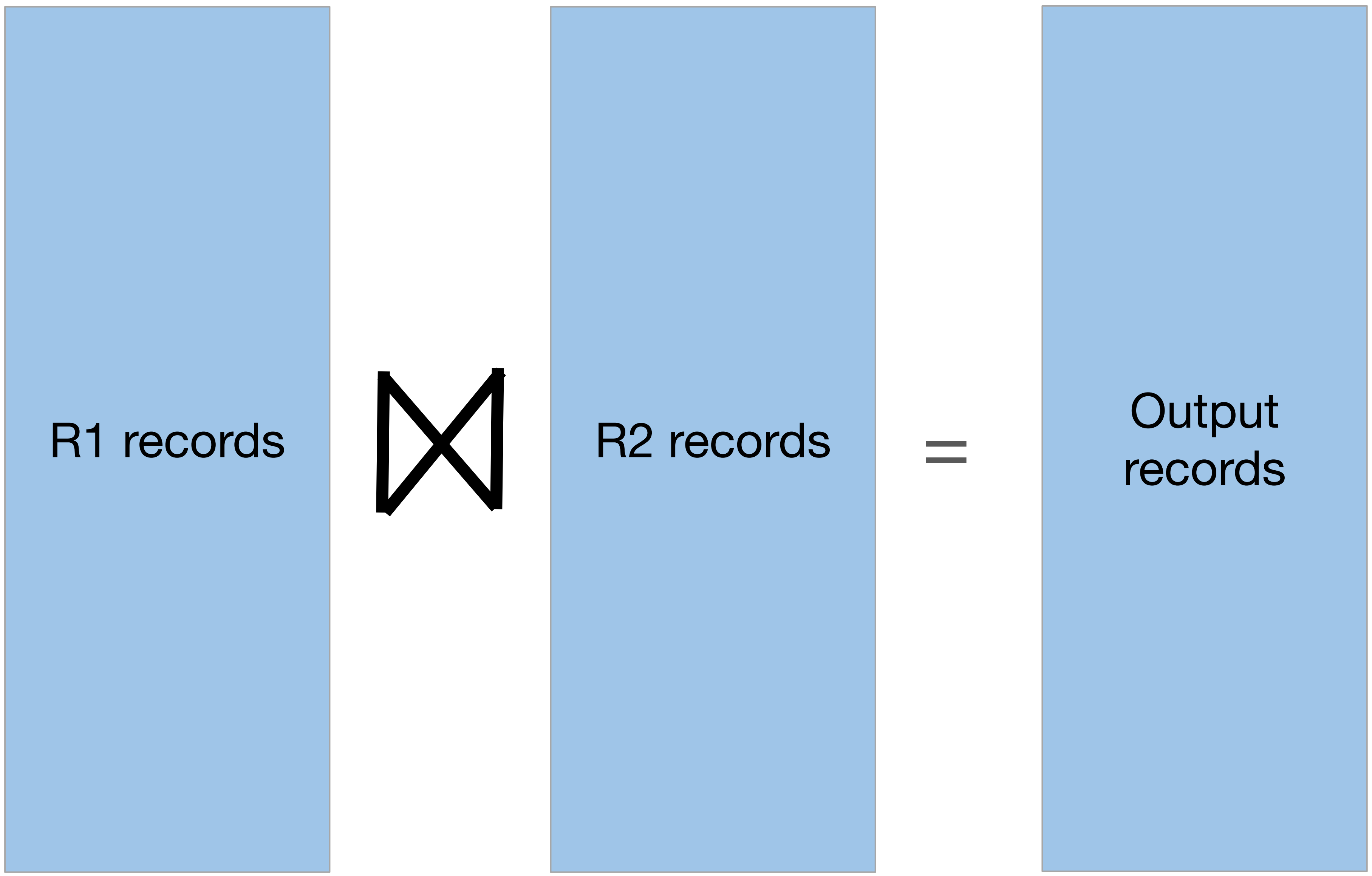


sort(R1) # sort on a
sort(R2)
# walk from top of each
# sorted record, downwards
# comparing like records

similar to the "merge
phase" in merge-sort

Join Implementation

SELECT x from R1 JOIN R2 ON R1.a = R2.a




```
sort(R1) # sort on a
sort(R2)
r1 = pop(R1)
r2 = pop(R2)
while r1 != nil and r2 != nil:
    if r1.a = r2.a:
        add(OUT, r1.x)
        r1 = pop(R1)
        r2 = pop(R2)
    elif r1.a < r2.a:
        r1 = pop(R1)
    else:
        r2 = pop(R2)
```

Asymptotic complexity?

Join Implementation

SELECT x from R1 JOIN R2 ON R1.a = R2.a

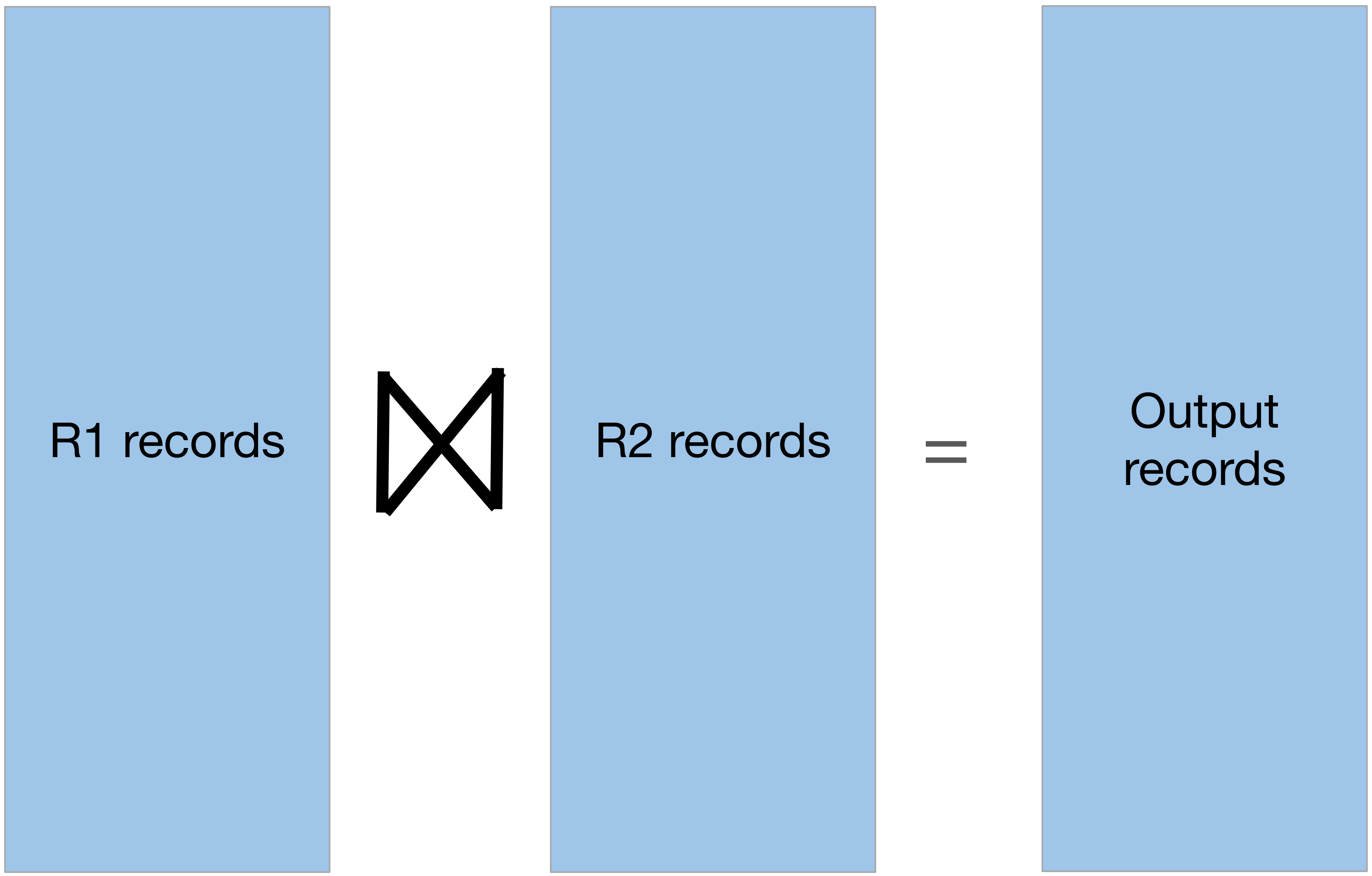
R1 records $\bowtie$ R2 records = Output records

```
sort(R1) # sort on a
sort(R2)
r1 = pop(R1)
r2 = pop(R2)
while r1 != nil and r2 != nil:
    if r1.a = r2.a:
        add(OUT, r1.x)
        r1 = pop(R1)
        r2 = pop(R2)
    elif r1.a < r2.a:
        r1 = pop(R1)
    else:
        r2 = pop(R2)
```

Asymptotic complexity?
O(NlogN)

Join Implementation

SELECT x from R1 JOIN R2 ON R1.a = R2.a



Other option: Hash Joins

- Hash all records in one relation
- Iterate through the other relation
- Look for matches in the other relation using the hashtable

O(N), but "constant costs" of hashing on reach record, and space requirements for the HT