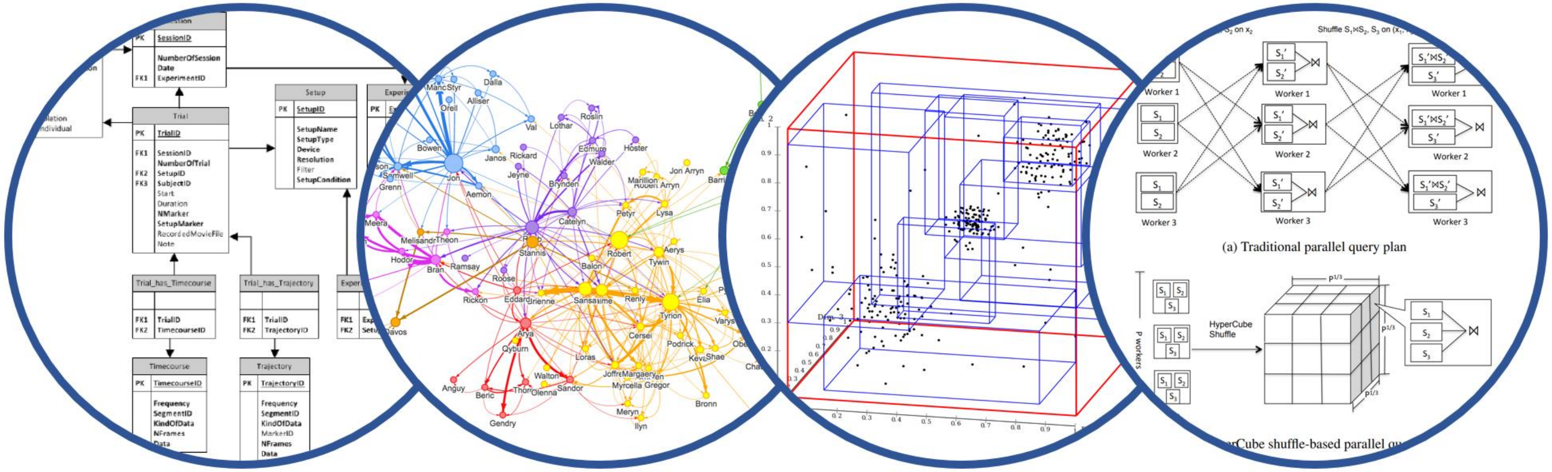



# Introduction to Data Management

# Query Execution

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

# Announcements

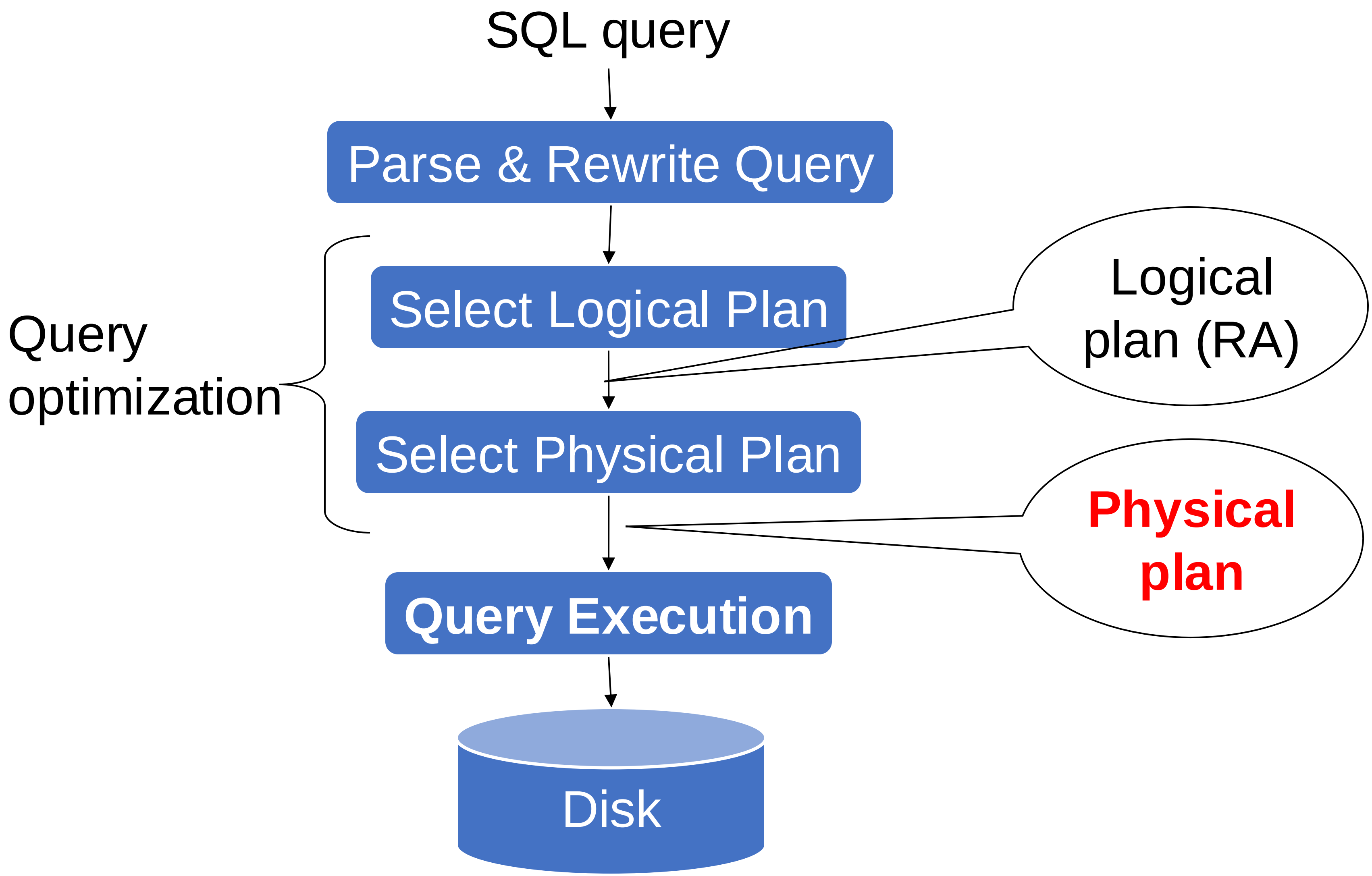
HW6:

- Part 1 due 11/15  No late days (for quick feedback)

- Part 2 due 11/27.  Much more work than part 1

Sections tomorrow: OH for Part 1, Demo for Part 2

# Query Evaluation Steps Review



# Relational Algebra Operators

- Union $\cup$, ~~intersection $\cap$~~, difference $-$
- Selection $\sigma$
- Projection $\pi$
- Cartesian product $\times$, join $\bowtie$
- (Rename $\rho$)

# Relational Algebra Operators

- Union $\cup$, ~~intersection $\cap$~~, difference $-$
- Selection $\sigma$
- Projection $\pi$
- Cartesian product $\times$, join $\bowtie$
- (Rename $\rho$)

RA

# Relational Algebra Operators

- Union ∪, ~~intersection ∩~~, difference -
- Selection σ
- Projection π
- Cartesian product ×, join ⋈
- (Rename ρ)
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

RA

# Relational Algebra Operators

- Union ∪, ~~intersection ∩~~, difference -
- Selection σ
- Projection π
- Cartesian product ×, join ⋈
- (Rename ρ)
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

RA

Extended RA

# Agenda for this and next week

The query engine:

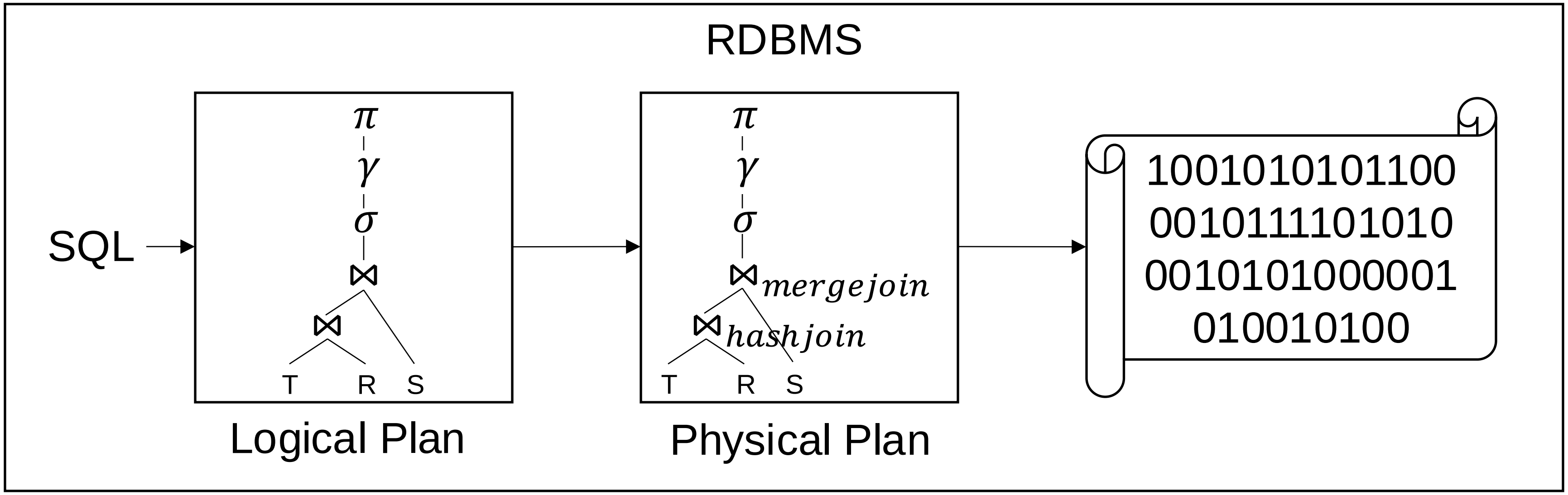
Today

- Query execution in main memory
- Indexes and external memory
- Query optimization and cardinality estimation

# Overview of the Query Engine

# Logical vs Physical Plans

- SQL is translated into RA
- RA (logical plan) does not fully describe execution
- RA *with algorithms* (physical plan) is needed



# Plan Enumeration

- SQL → RA
- RA → RA → RA → …          plan enumeration
- RA → Physical Plan …
- → Least cost physical plan
- Execute least cost plan

# Plan Enumeration

RDBMS

SQL

```
SELECT *
  FROM T, R, S
 WHERE …
```

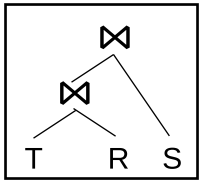
# Plan Enumeration

RDBMS

SQL

```
SELECT *
  FROM T, R, S
 WHERE …
```

Logical Plan



# Plan Enumeration

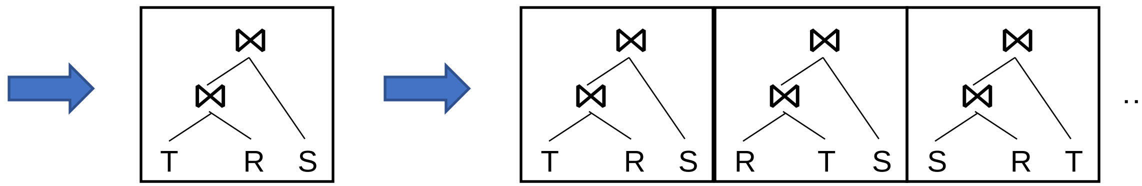
RDBMS

SQL

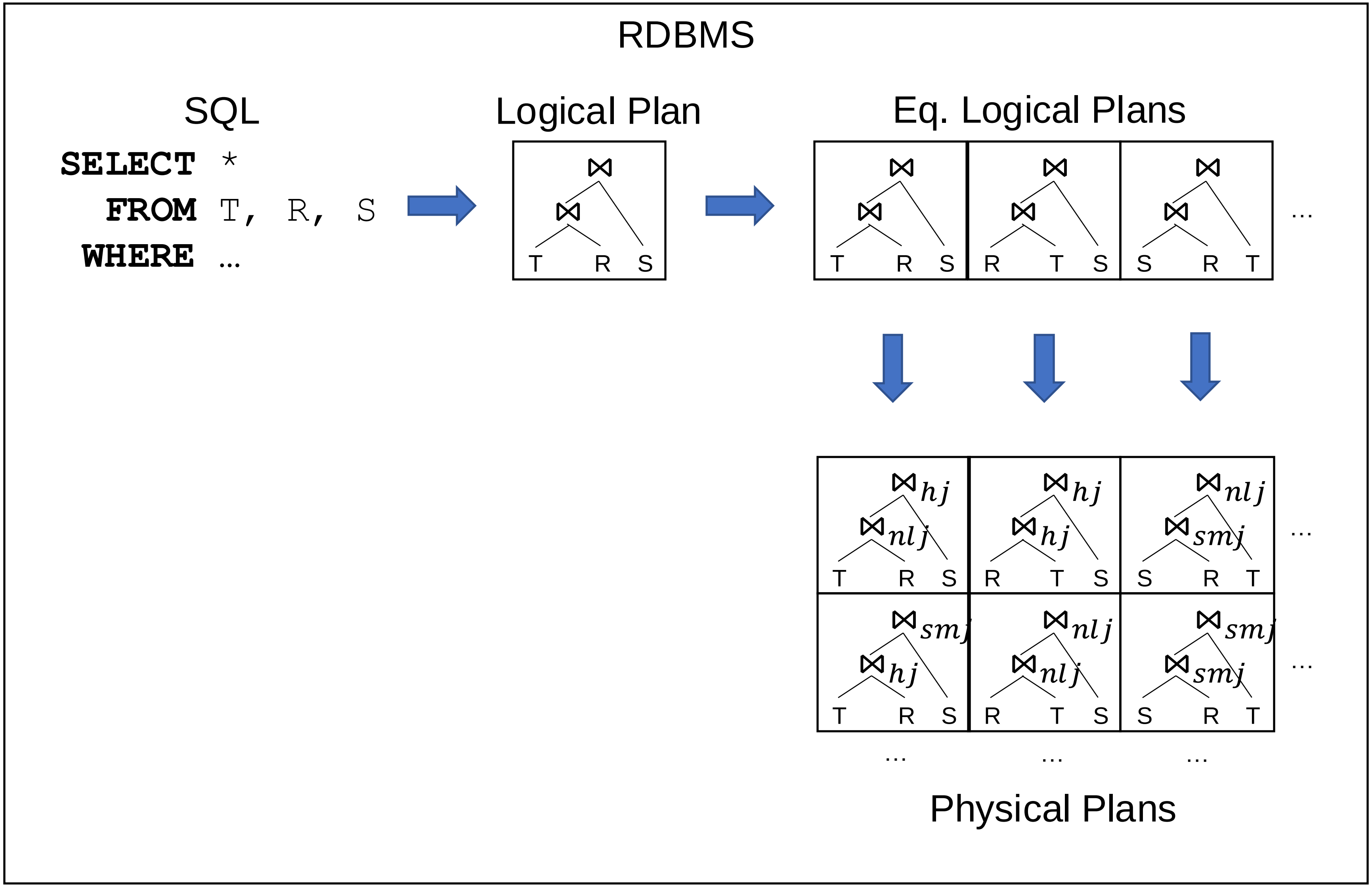
```
SELECT *
  FROM T, R, S
 WHERE …
```

Logical Plan

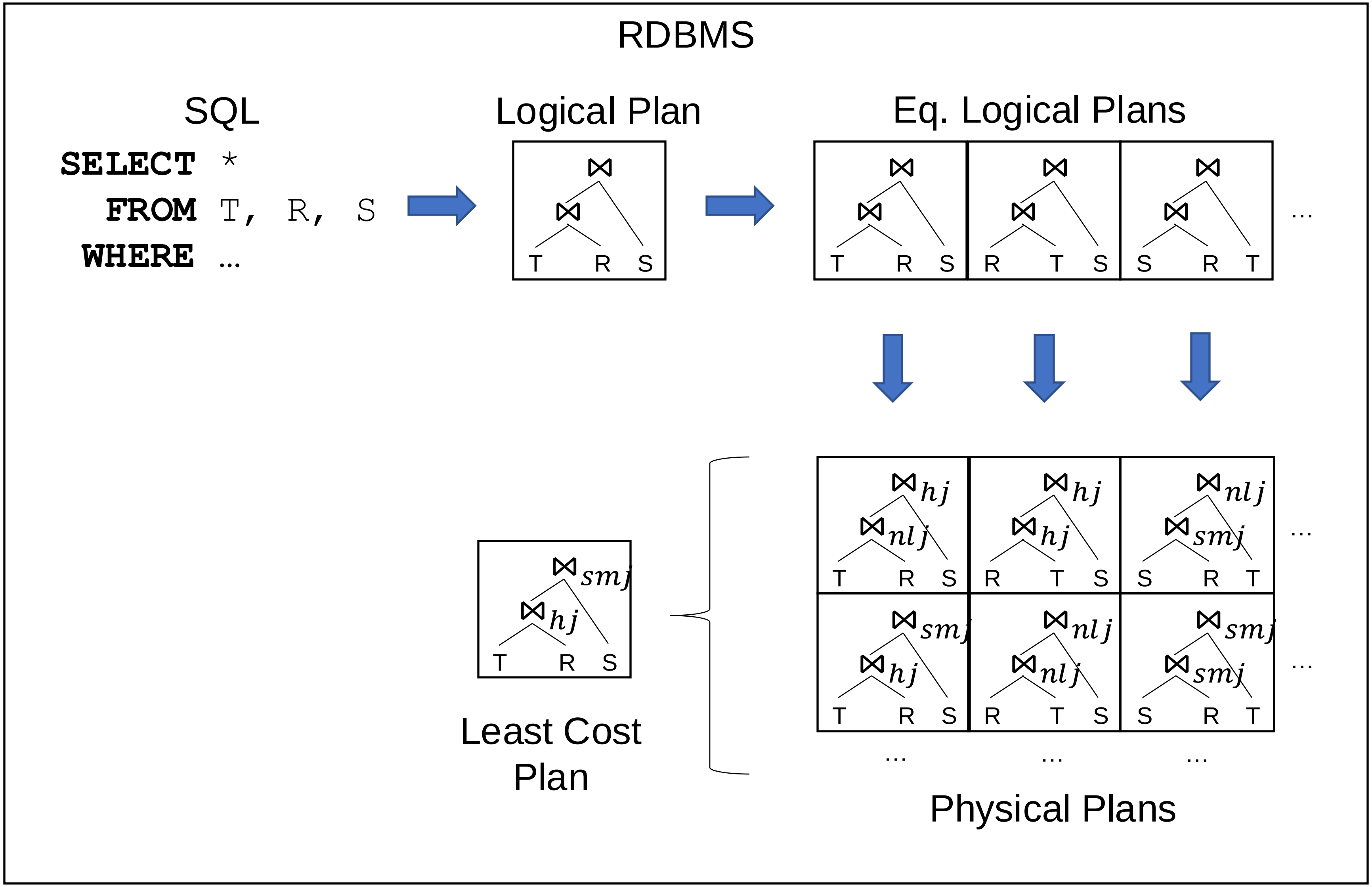
Eq. Logical Plans



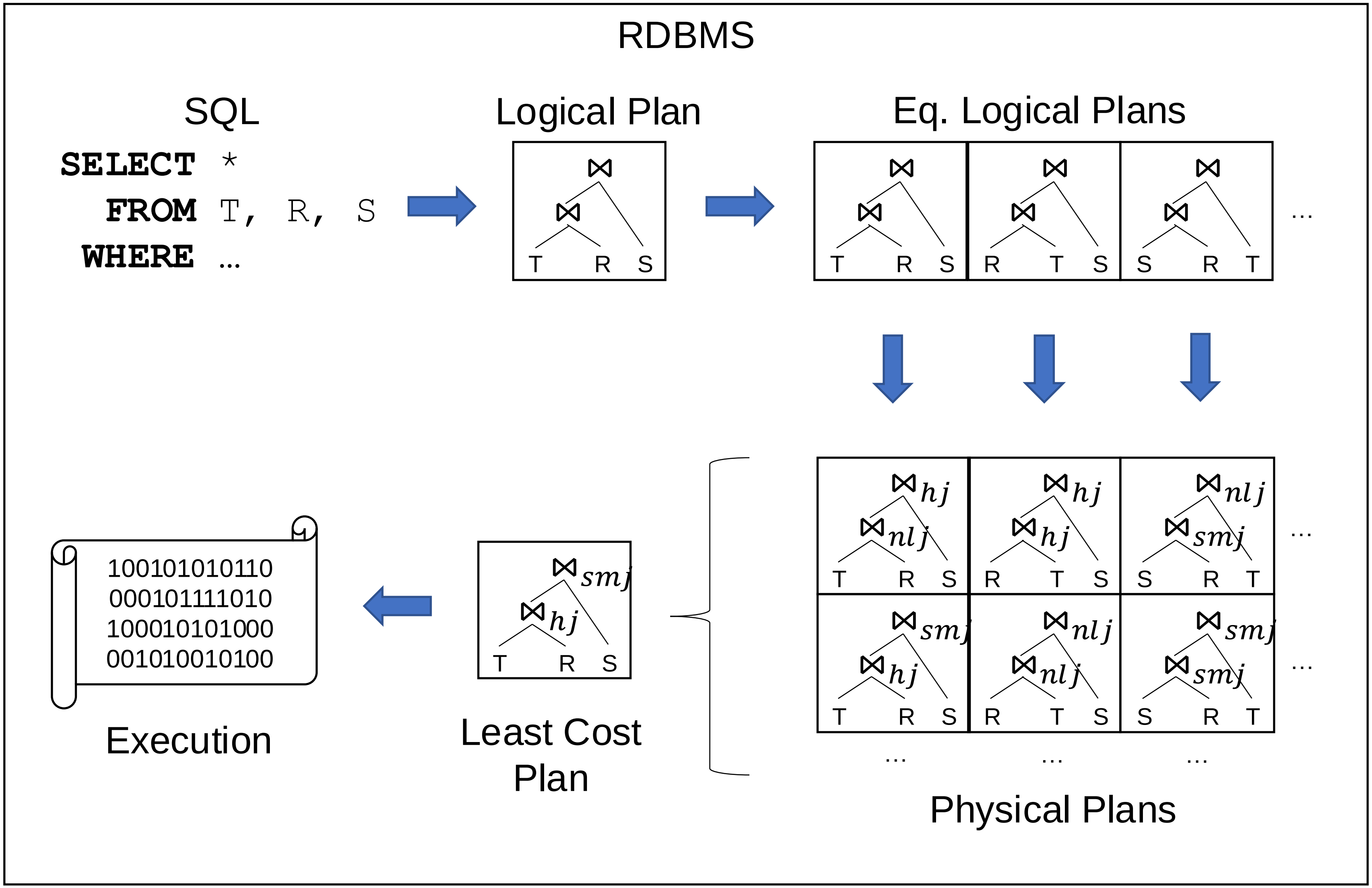
# Plan Enumeration



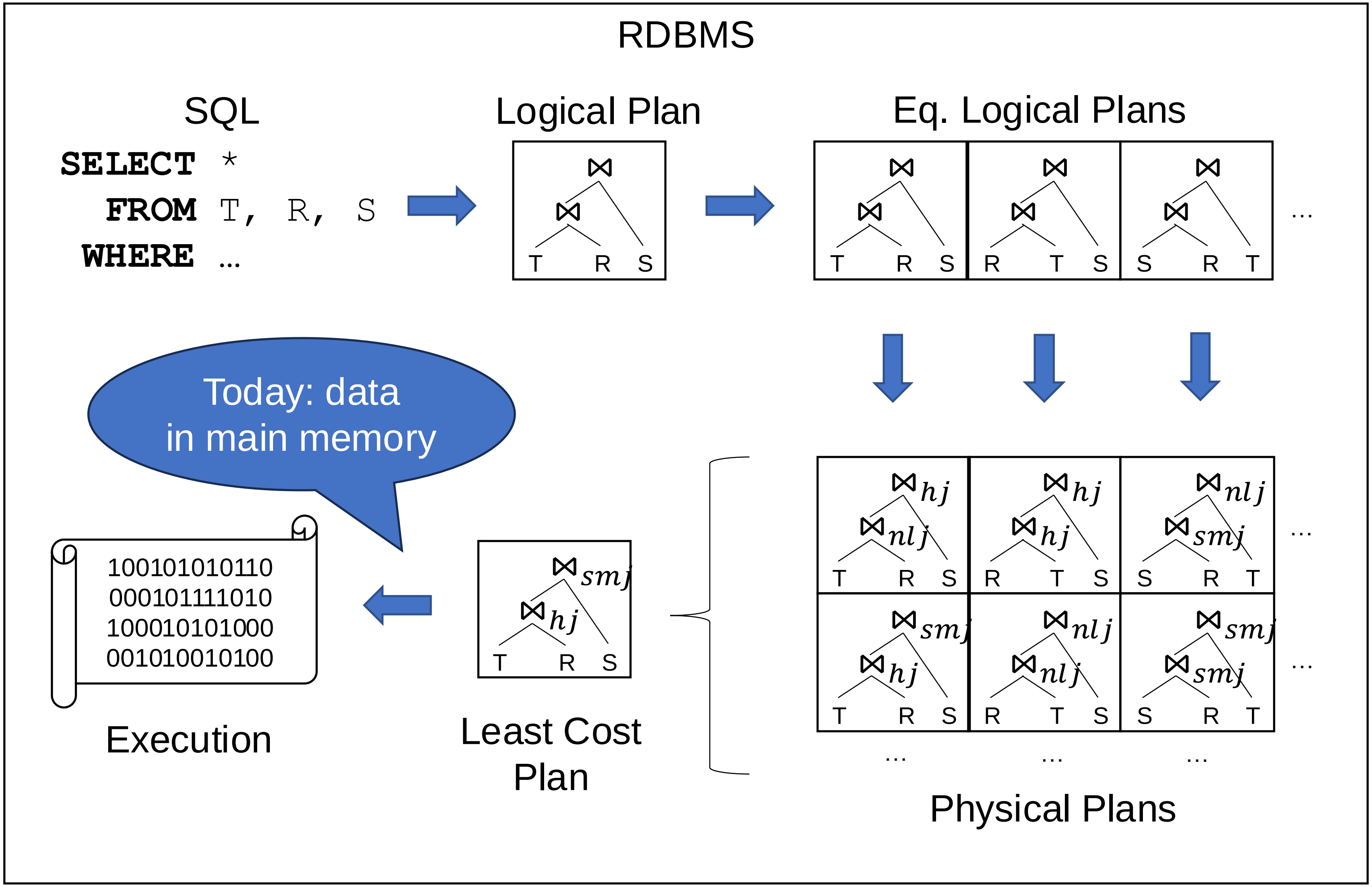
# Plan Enumeration



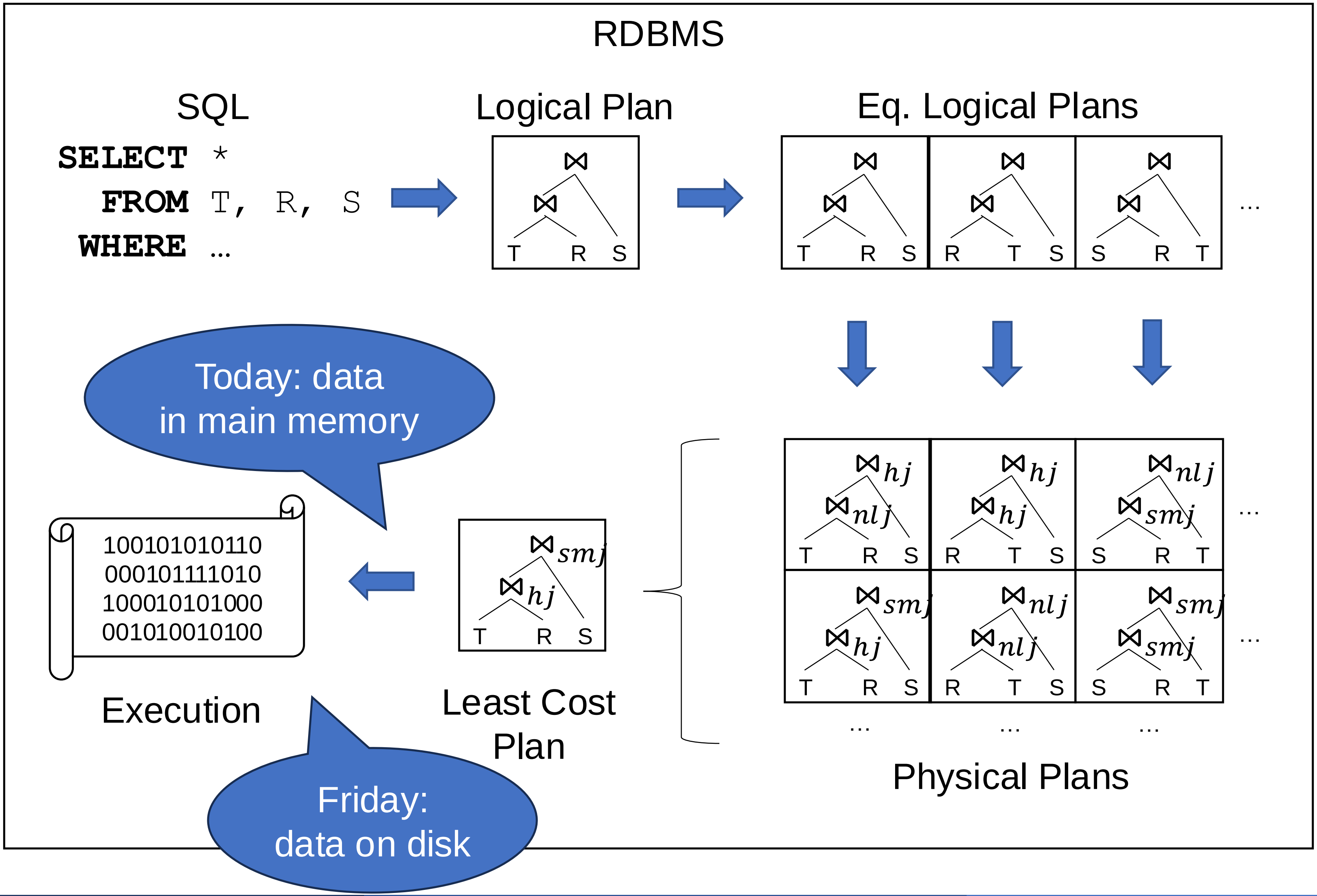
# Plan Enumeration



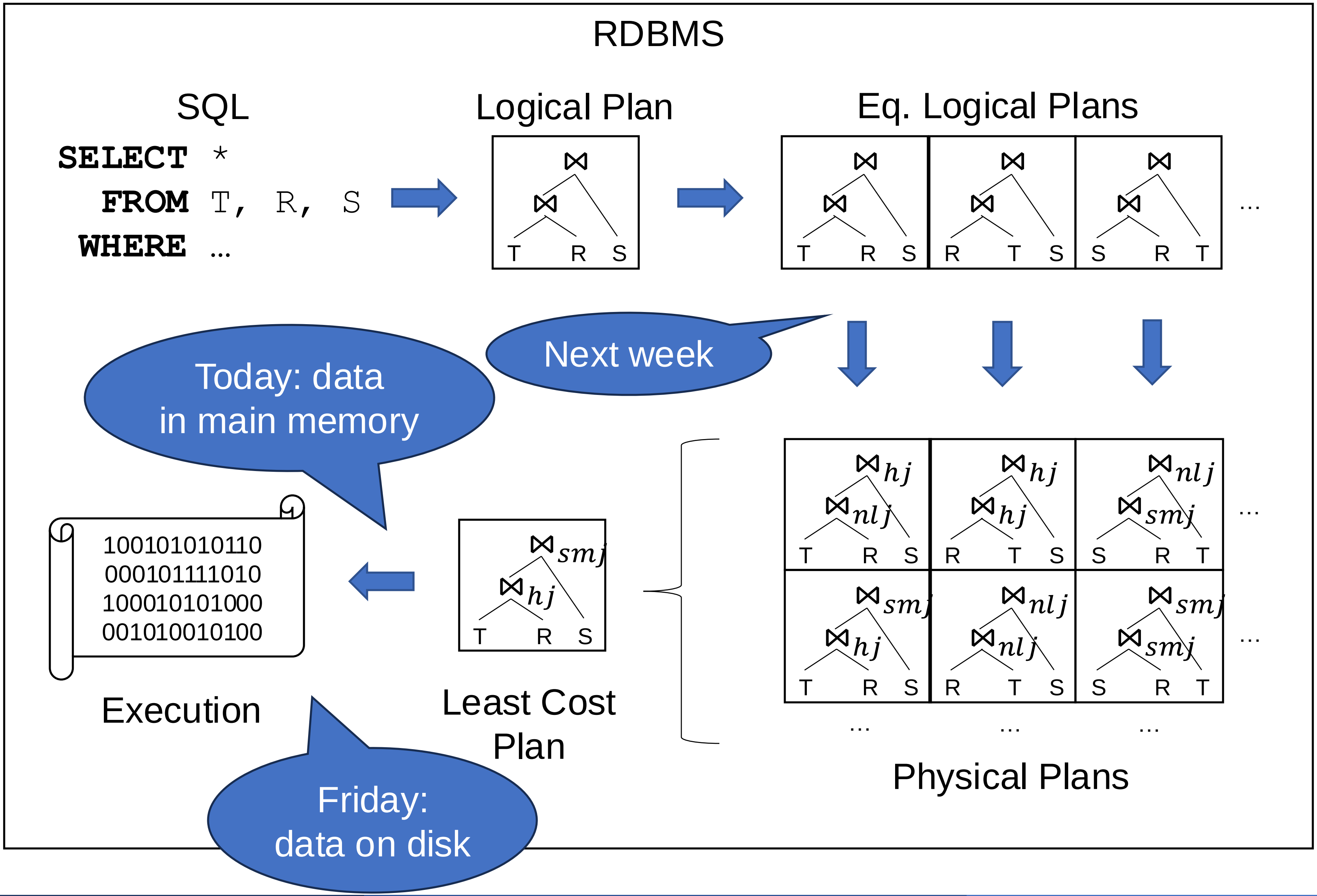
# Plan Enumeration



# Plan Enumeration



# Plan Enumeration



# Physical Operators

# Physical Operators

Physical operator is an algorithm for an RA operator

- Merge-join
- Sort-based group-by
- …

Can be a main-memory, or disk-based algorithm

# Join Algorithms

For each employee, find the cars that they drive

```
SELECT P.Name, R.Car
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID;
```

| Name | Car |
|---|---|
| Jack | Charger |
| Magda | Civic |
| Magda | Pinto |

Payroll

| <u>UserID</u> | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Regist

| UserID | Car |
|---|---|
| 123 | Charger |
| 567 | Civic |
| 567 | Pinto |

# Join Algorithms

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

# Join Algorithms

$$\text{Payroll} \bowtie_{\text{UserID=UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

How would you compute the join?

# Join Algorithms

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

How would you compute the join?

Three physical operators:

1. Nested Loops
2. Hash-join
3. Merge-join

# Join Algorithms

$\text{Payroll} \bowtie_{\text{UserID=UserID}} \text{Regist}$

Outer table

Inner table

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

How would you compute the join?

Three physical operators:

1. Nested Loops
2. Hash-join
3. Merge-join

# 1. Nested-Loop Join

# 1. Nested Loop Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for x in Payroll
  for y in Regist
    if x.UserID = y.UserID
       output(x,y)
```

# 1. Nested Loop Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for x in Payroll
  for y in Regist
    if x.UserID = y.UserID
       output(x,y)
```

If |Payroll| = |Regist| = n, what is the runtime?

# 1. Nested Loop Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for x in Payroll
  for y in Regist
    if x.UserID = y.UserID
       output(x,y)
```

If |Payroll| = |Regist| = n, what is the runtime?

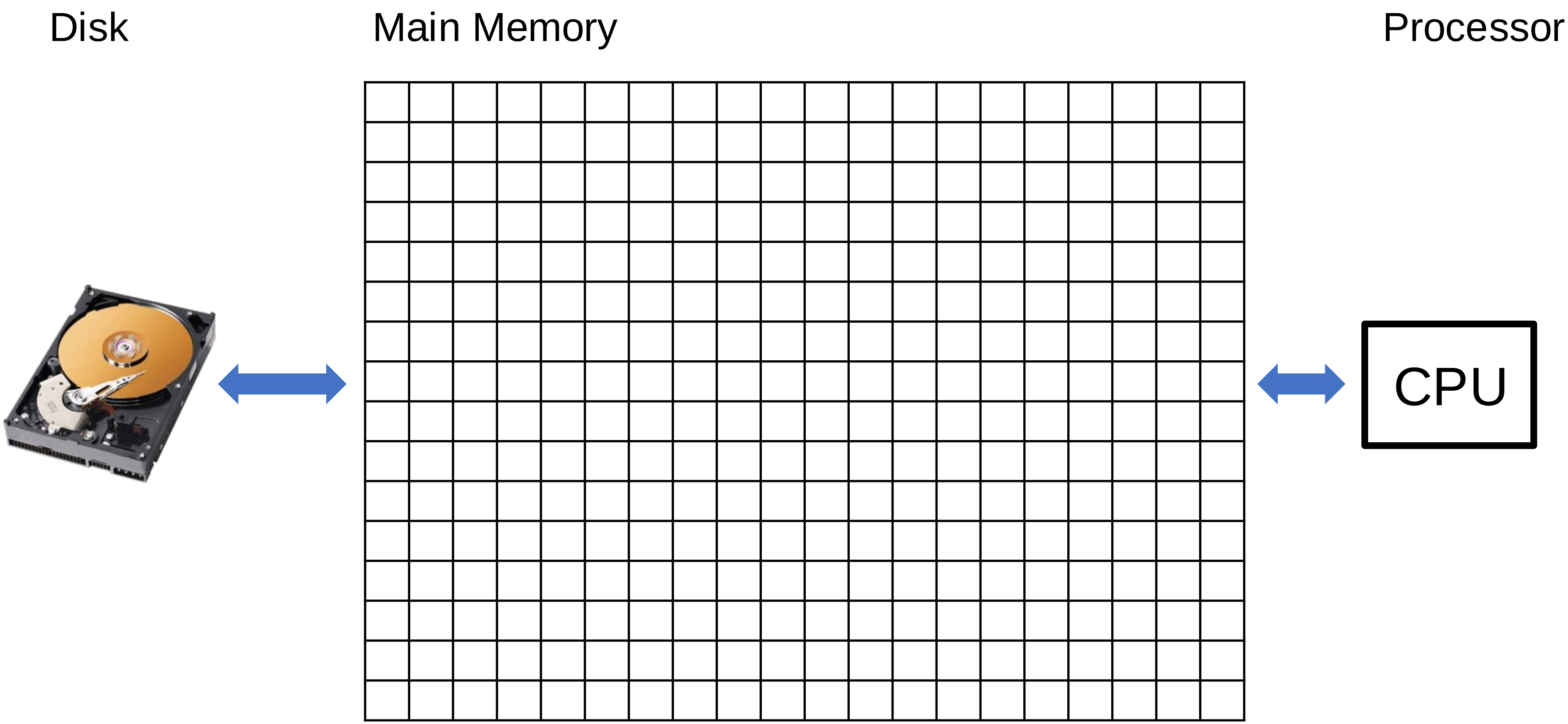
Runtime = $O(n^2)$

# Discussion

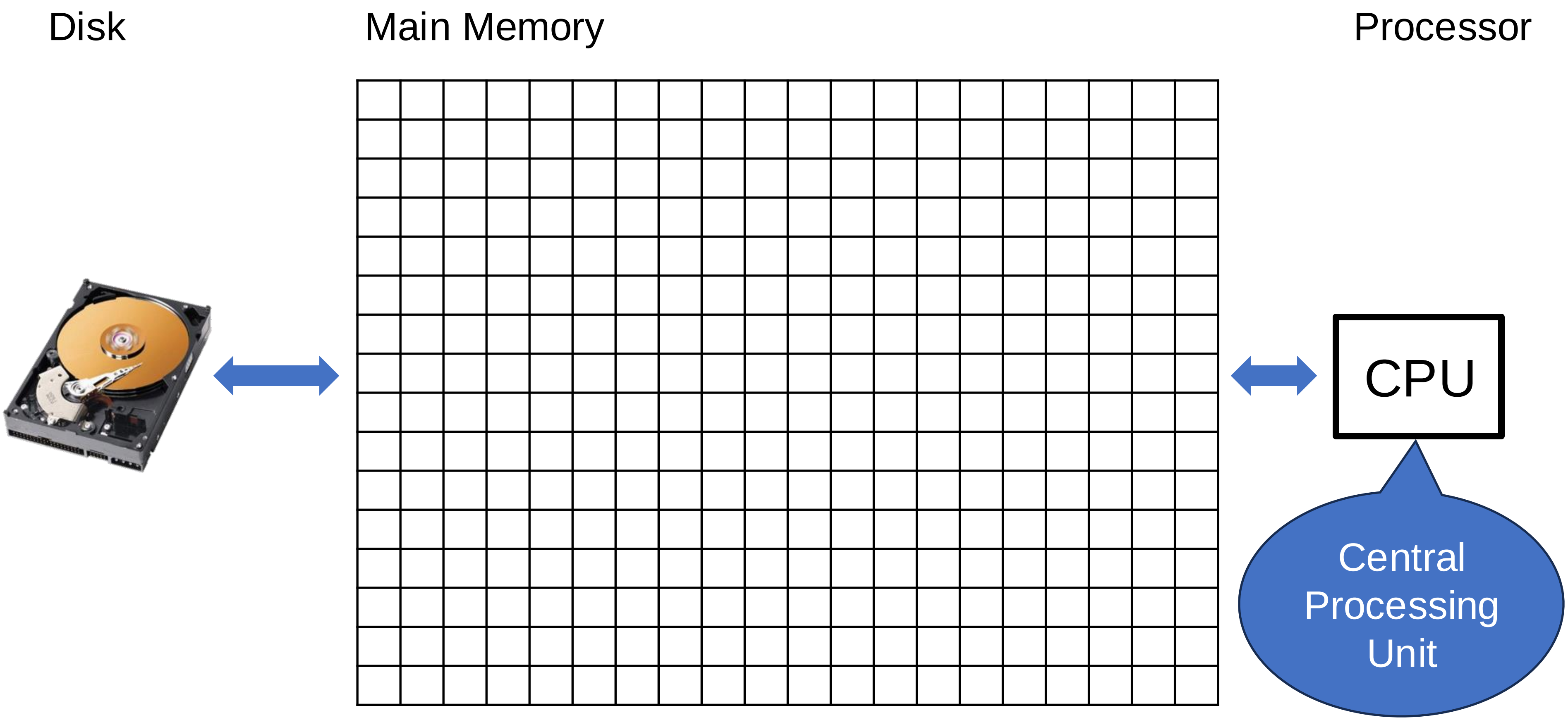
- Nested loop join: simplest implementation of join
- Works reasonably well when relation1 << relation2
  - $O(mn) \rightarrow O(n)$ when $m \ll n$
- Variations/improvements:
  - Index join: we have an index on inner table (later)
  - Block nested loop join (for data stored on disk)
- Next: hash-join and merge-join. First: a review…

# Review: Hash Tables

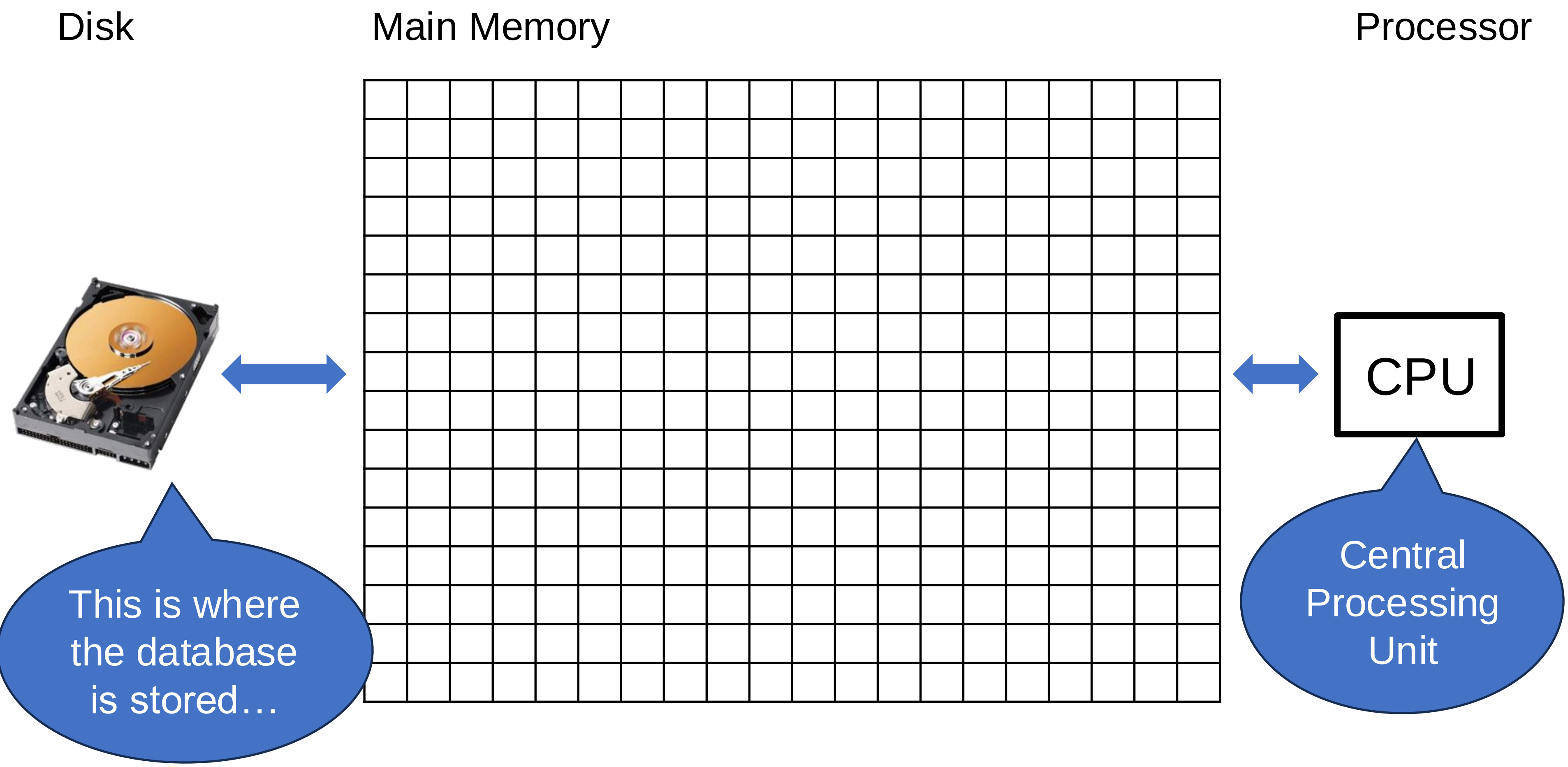
# How the Main Memory Works



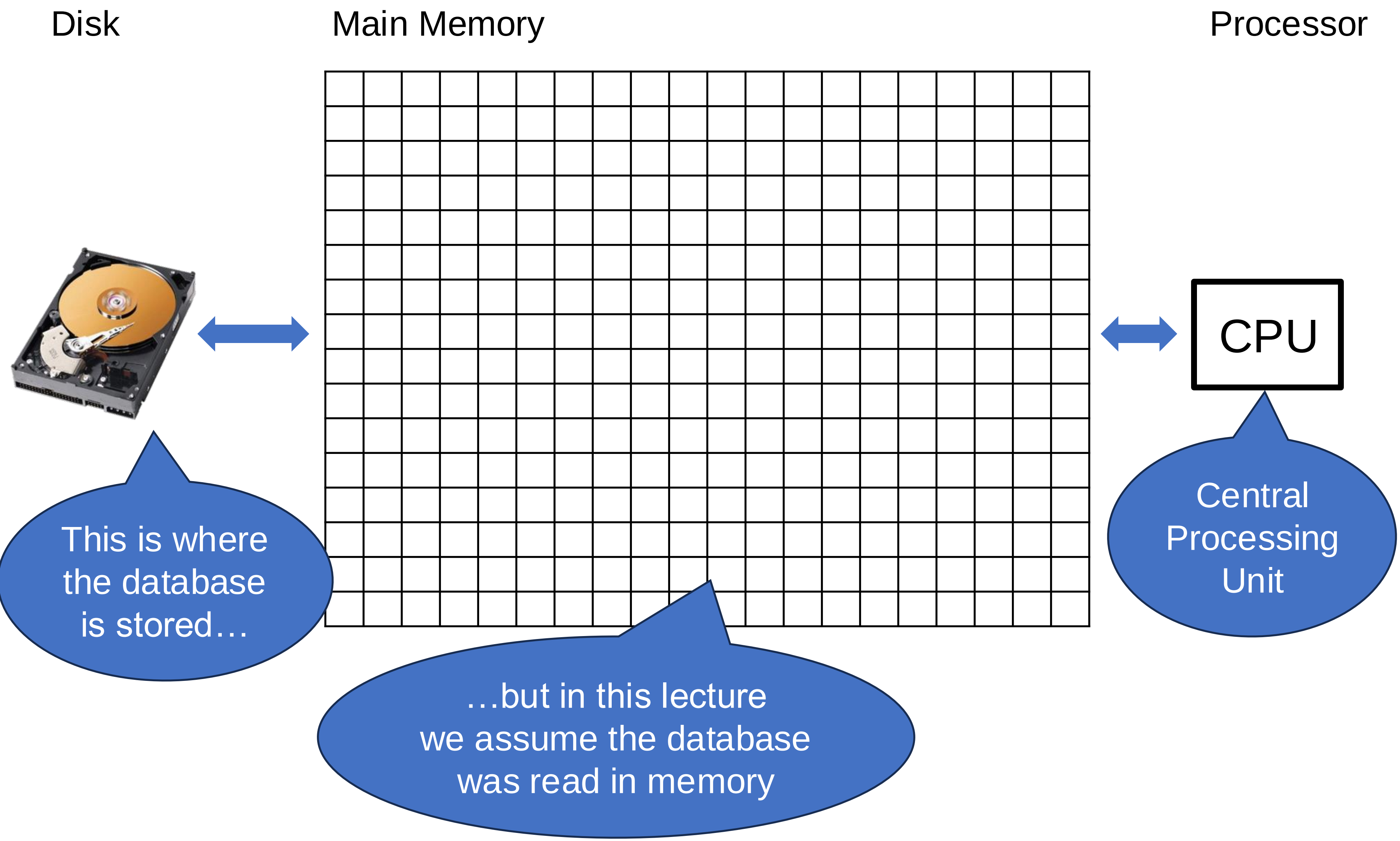
# How the Main Memory Works



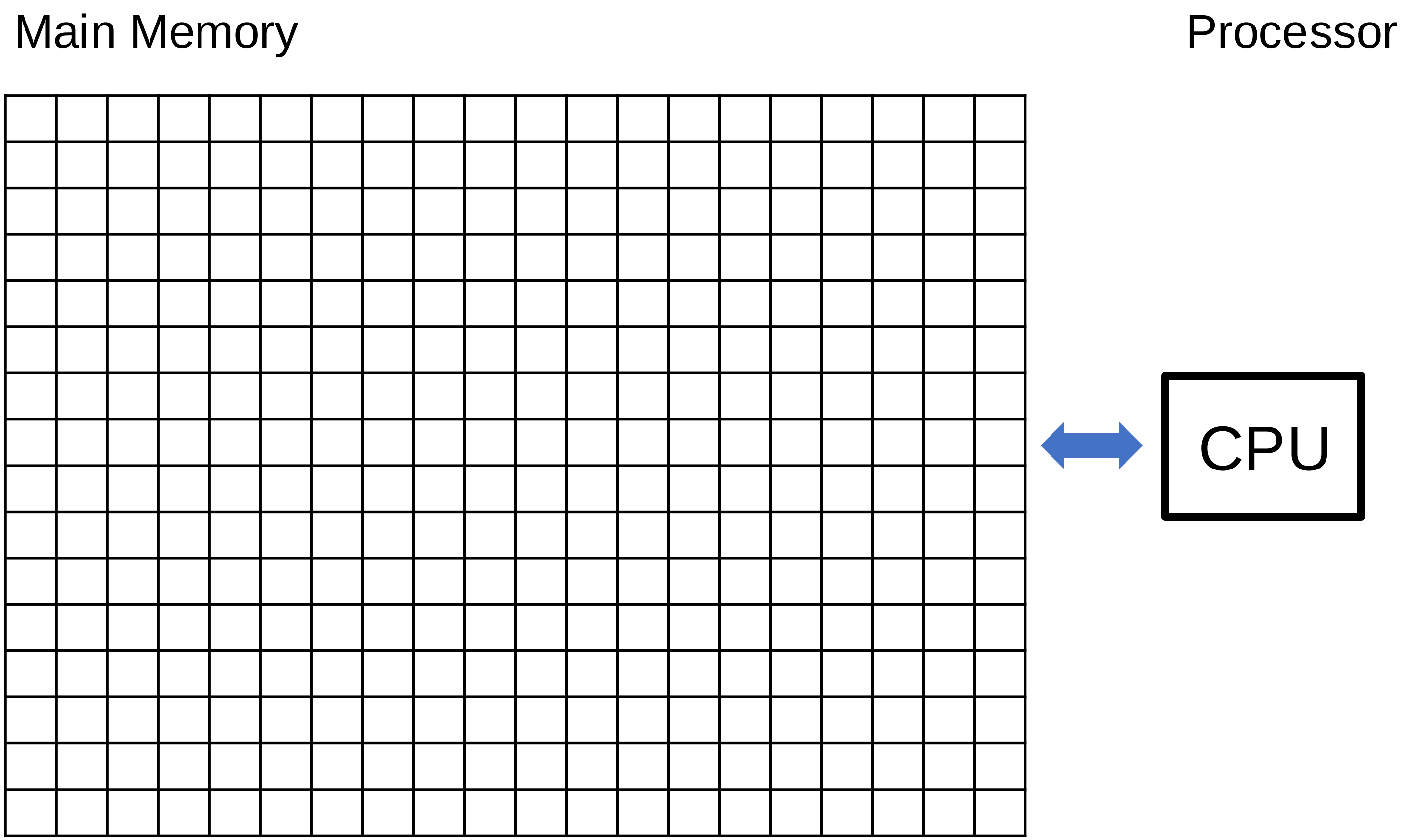
# How the Main Memory Works



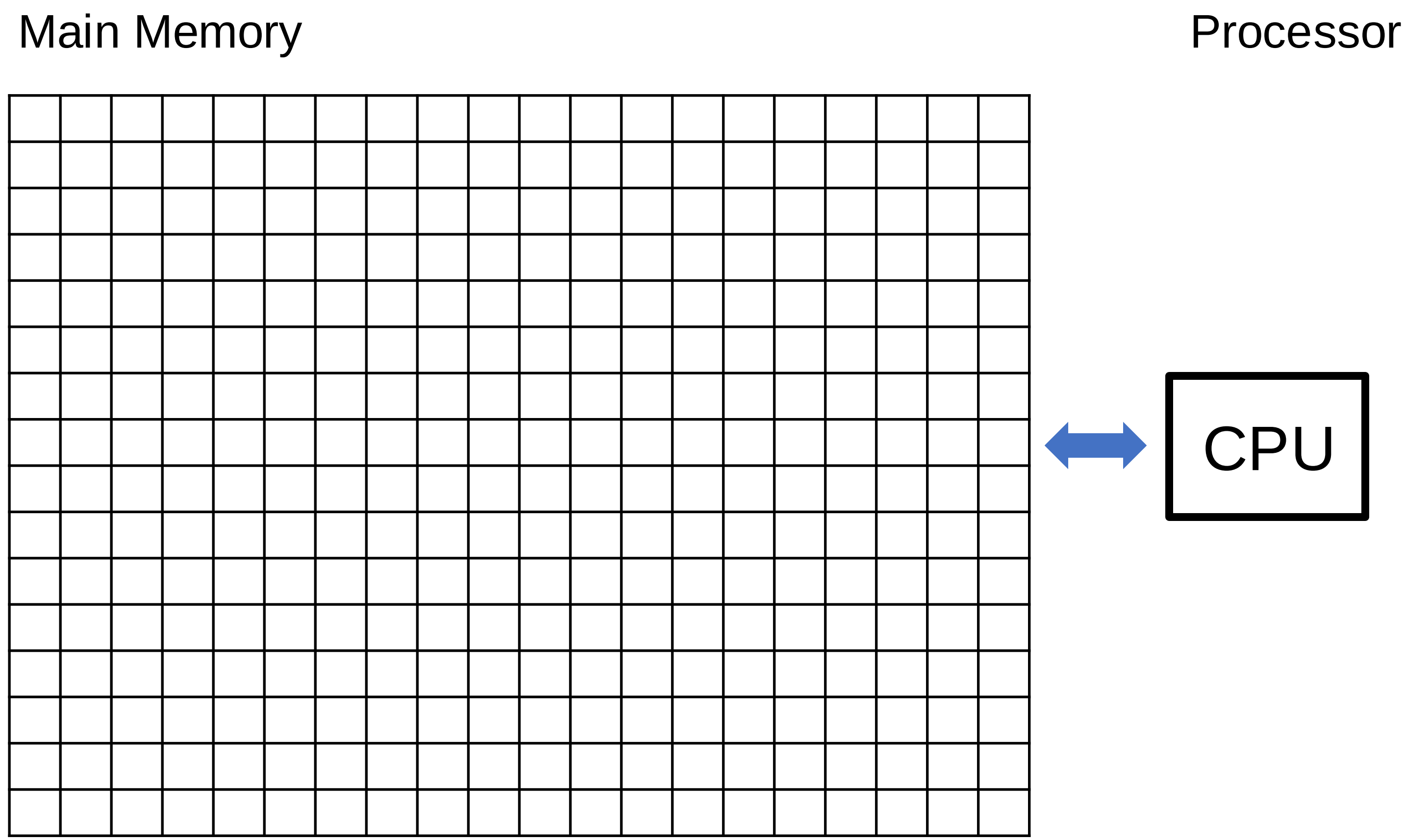
# How the Main Memory Works



# How the Main Memory Works

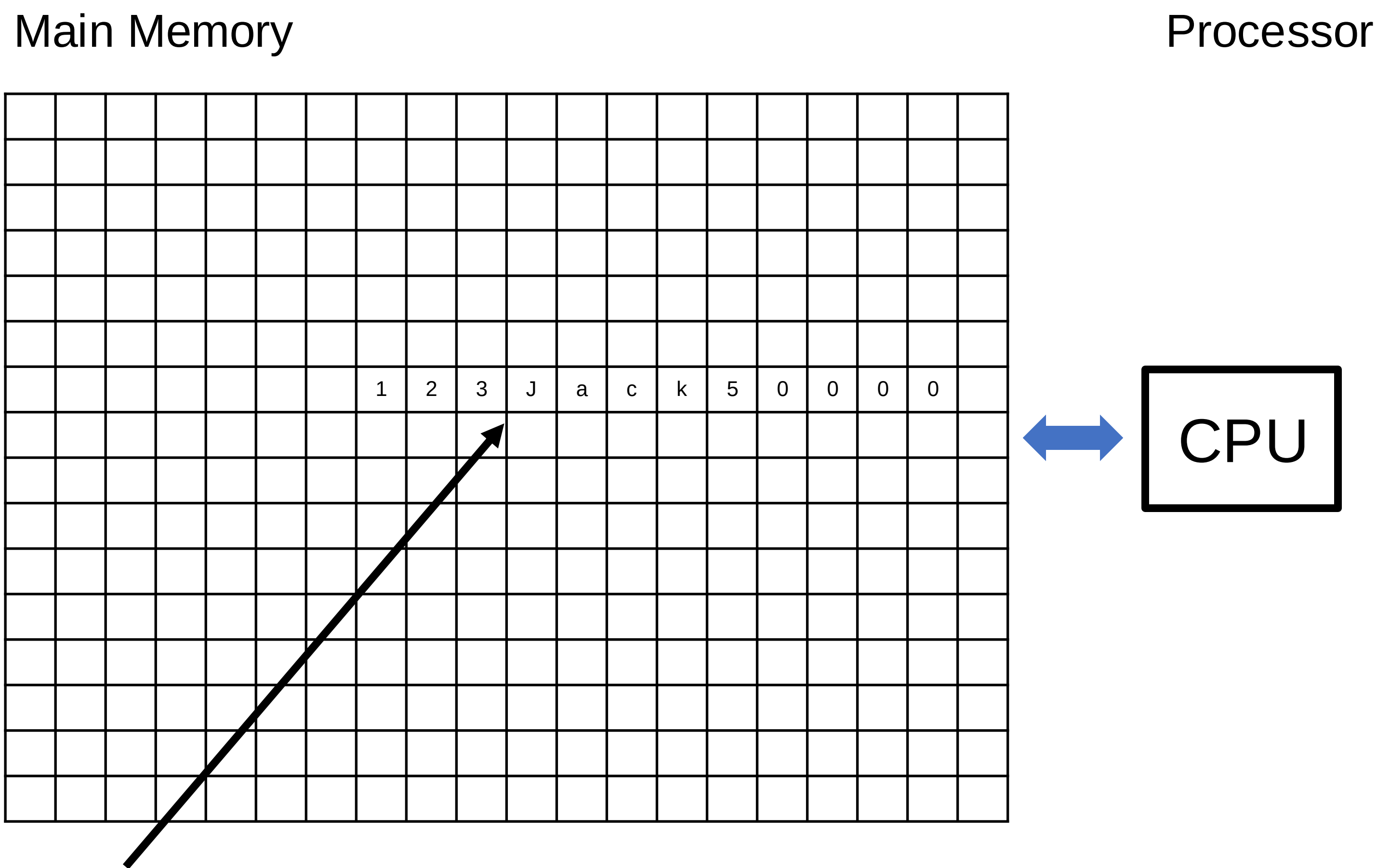


# How the Main Memory Works



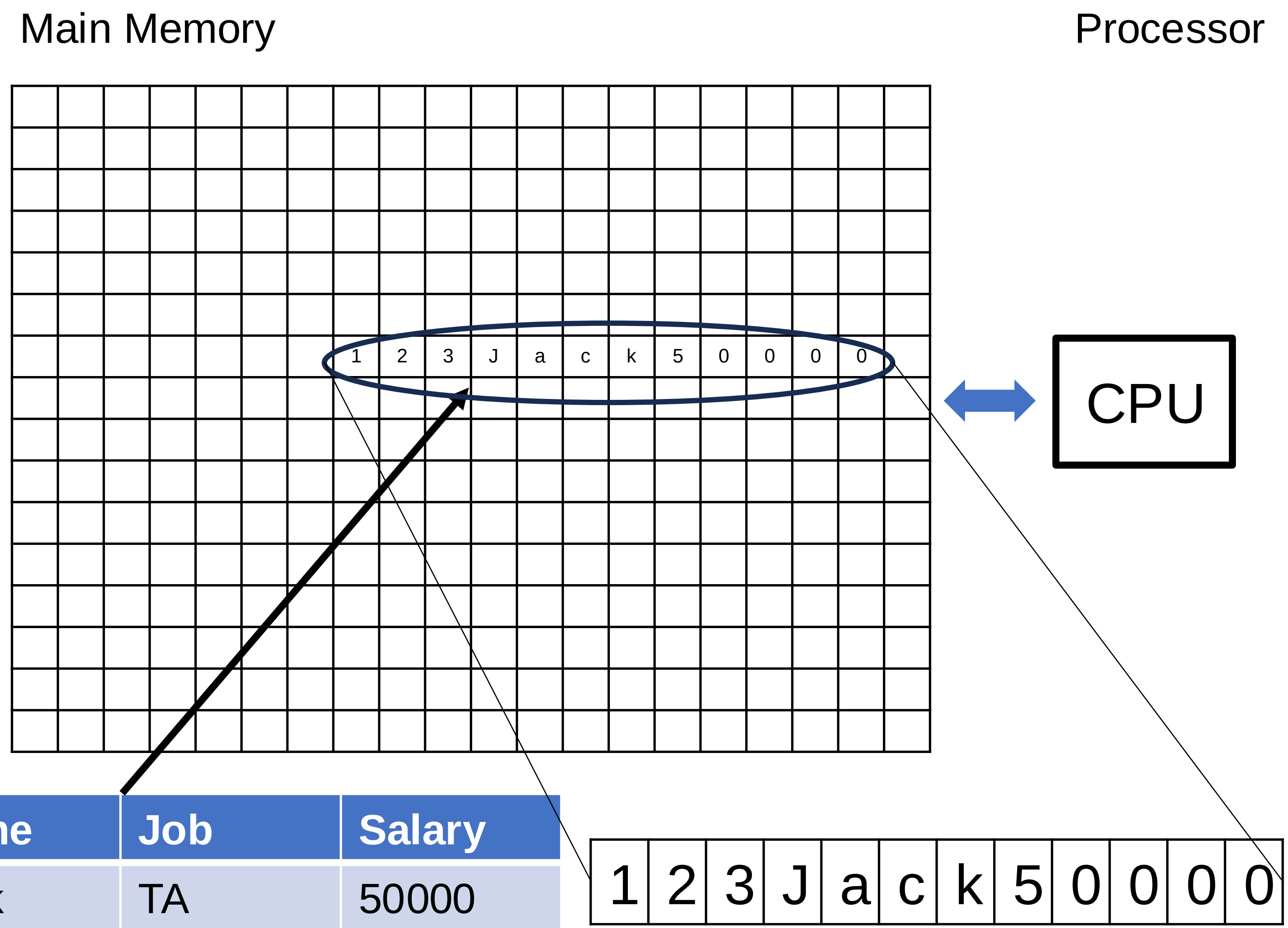

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| | … | | |

# How the Main Memory Works



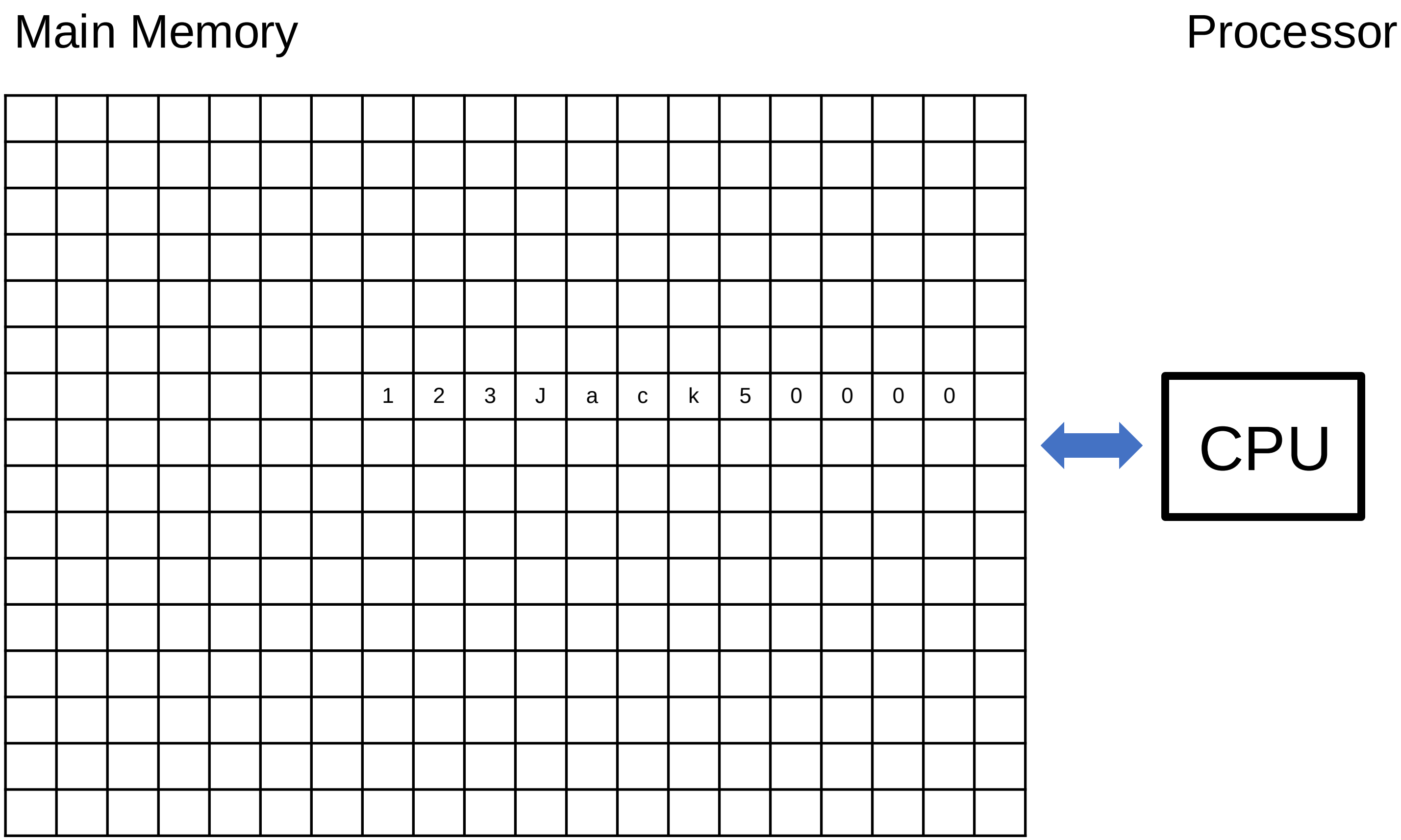

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| | … | | |

# How the Main Memory Works



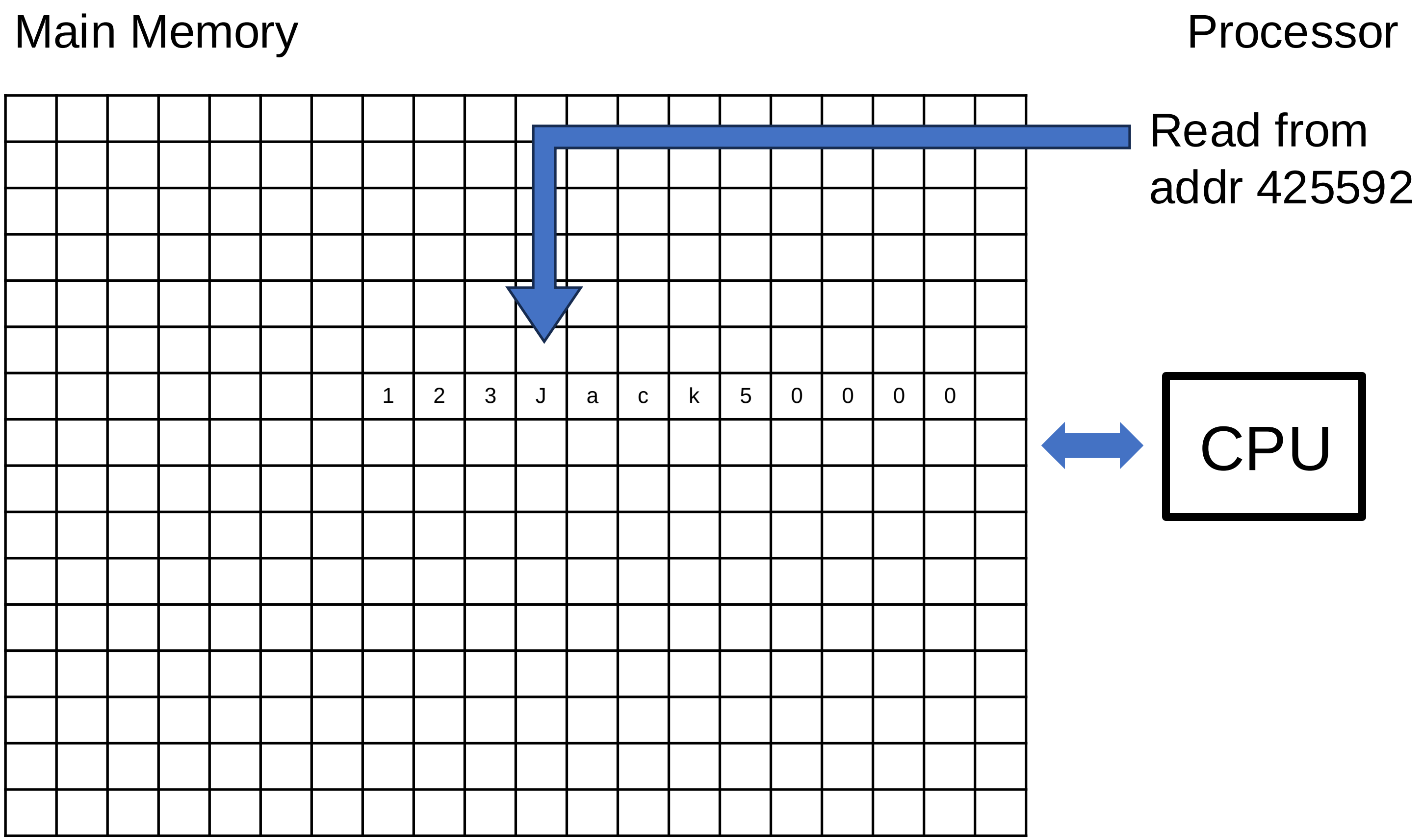

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| | … | | |

# How the Main Memory Works



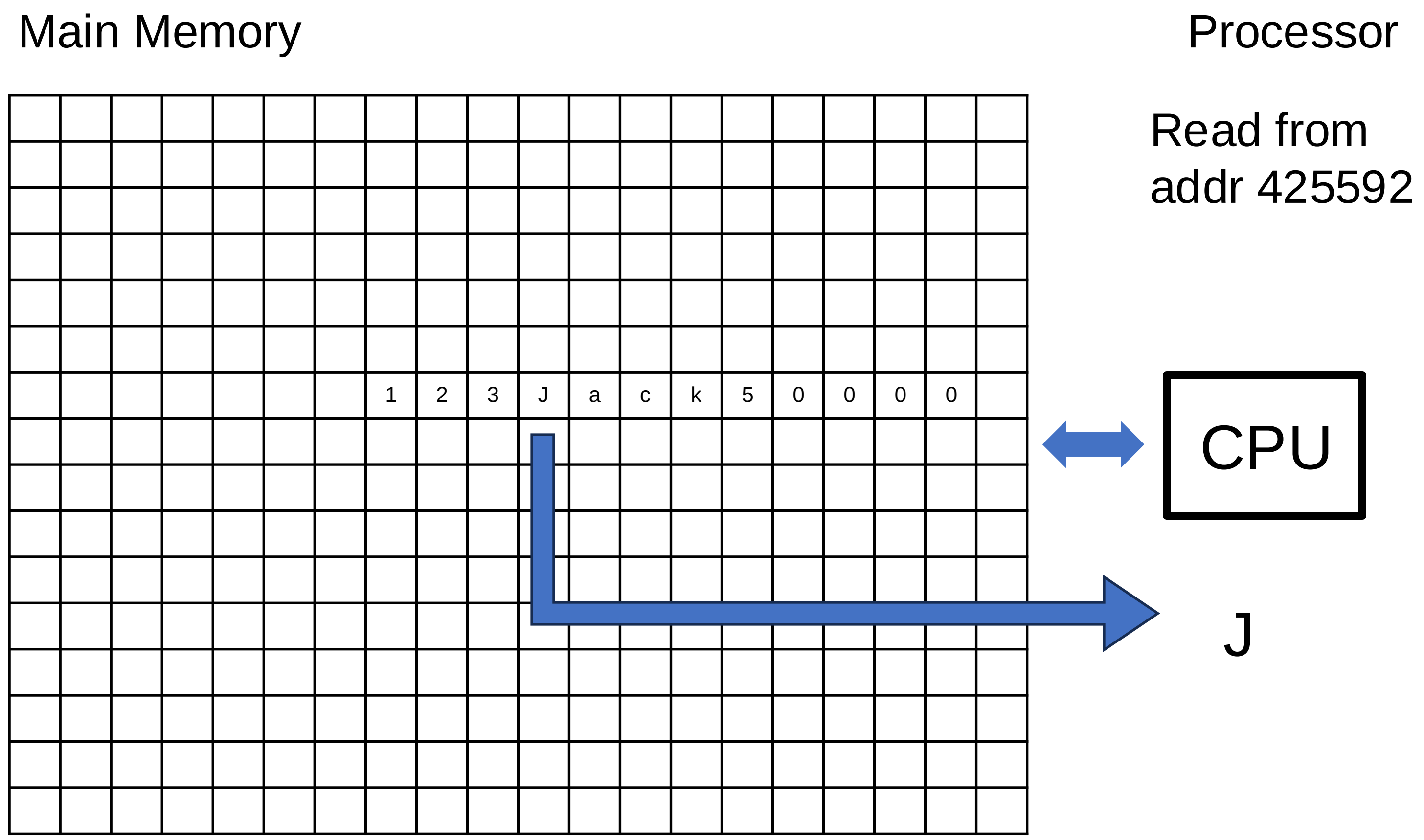

To read from Main Memory, the CPU needs to know the address

# How the Main Memory Works



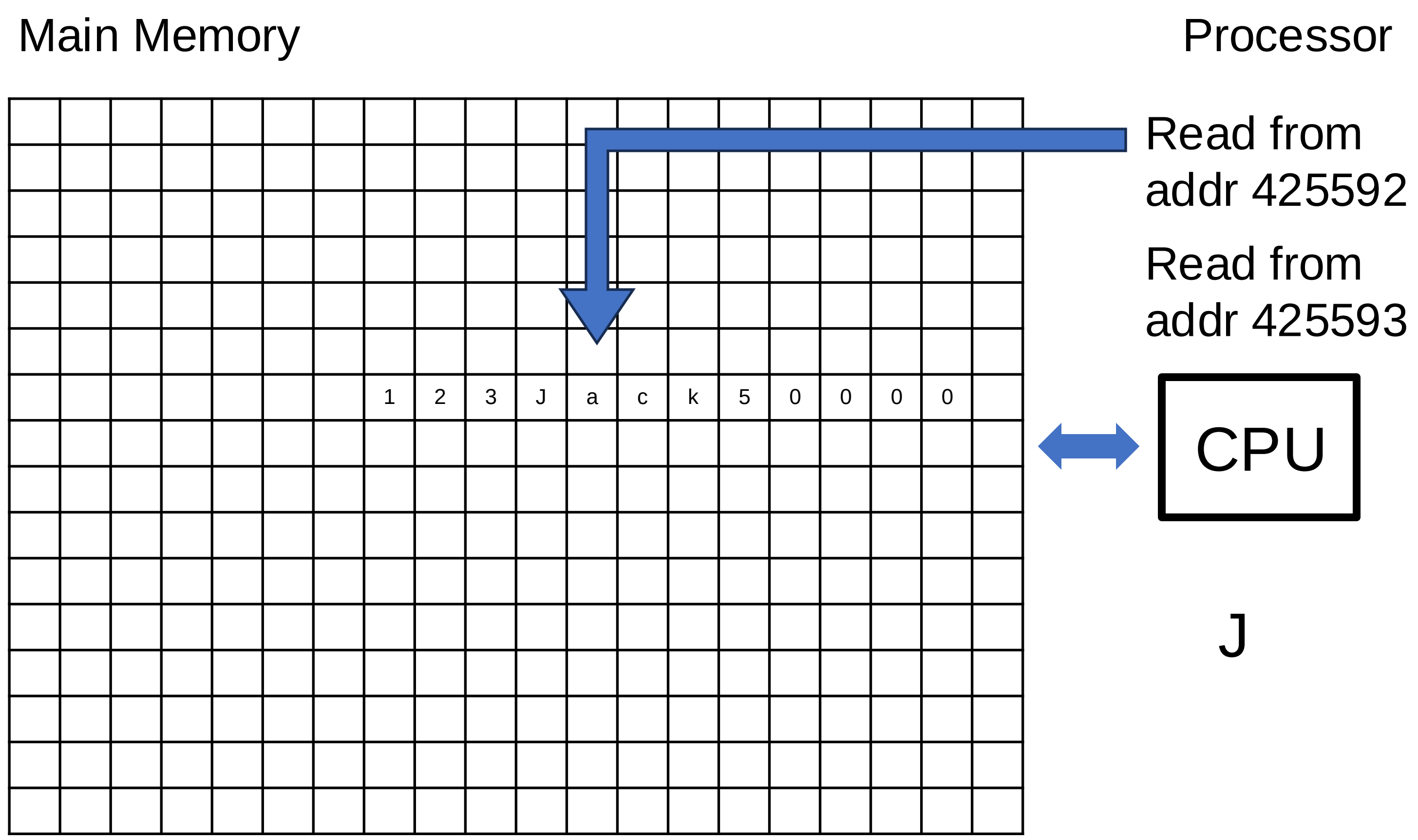

To read from Main Memory, the CPU needs to know the address

# How the Main Memory Works



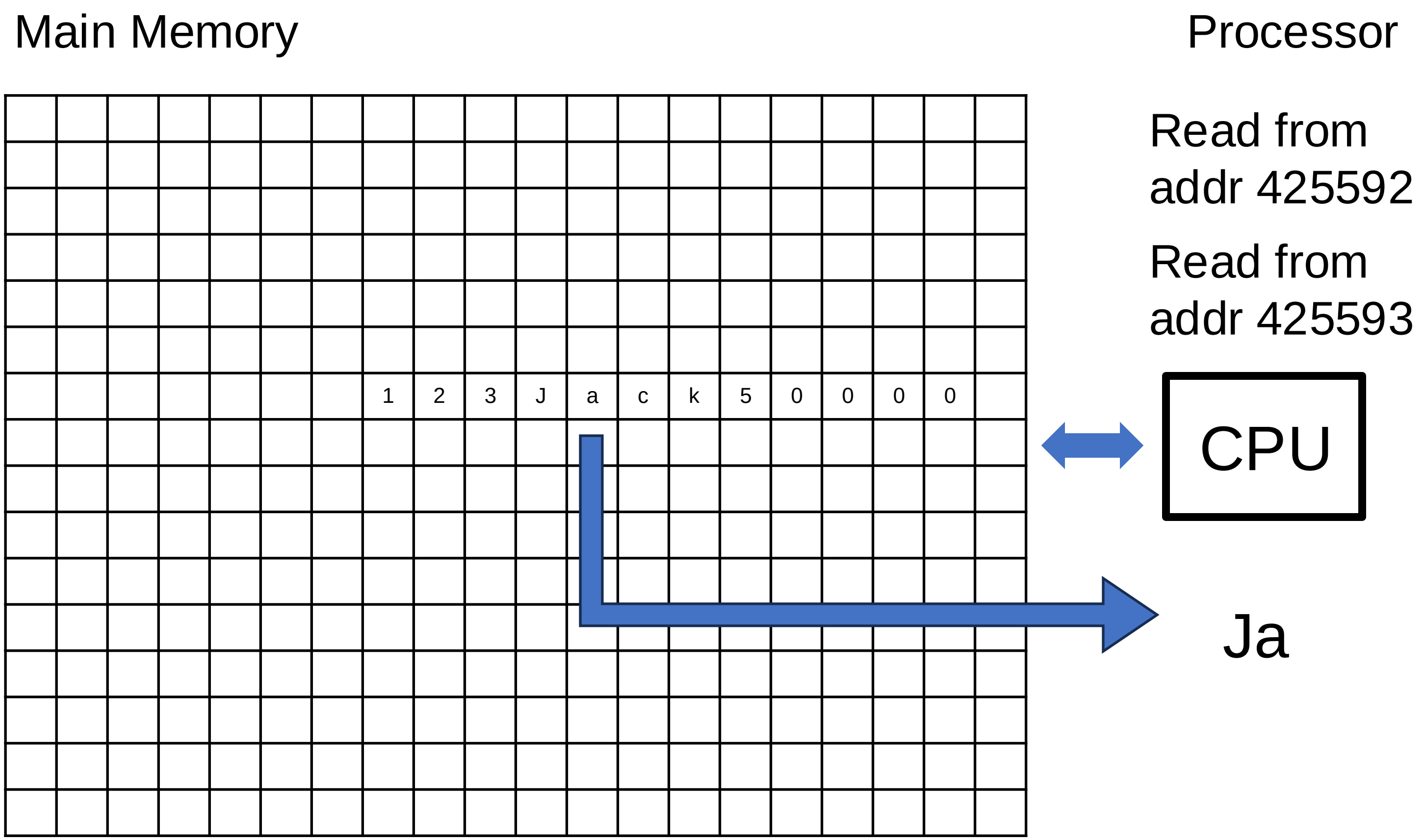

To read from Main Memory, the CPU needs to know the address

# How the Main Memory Works



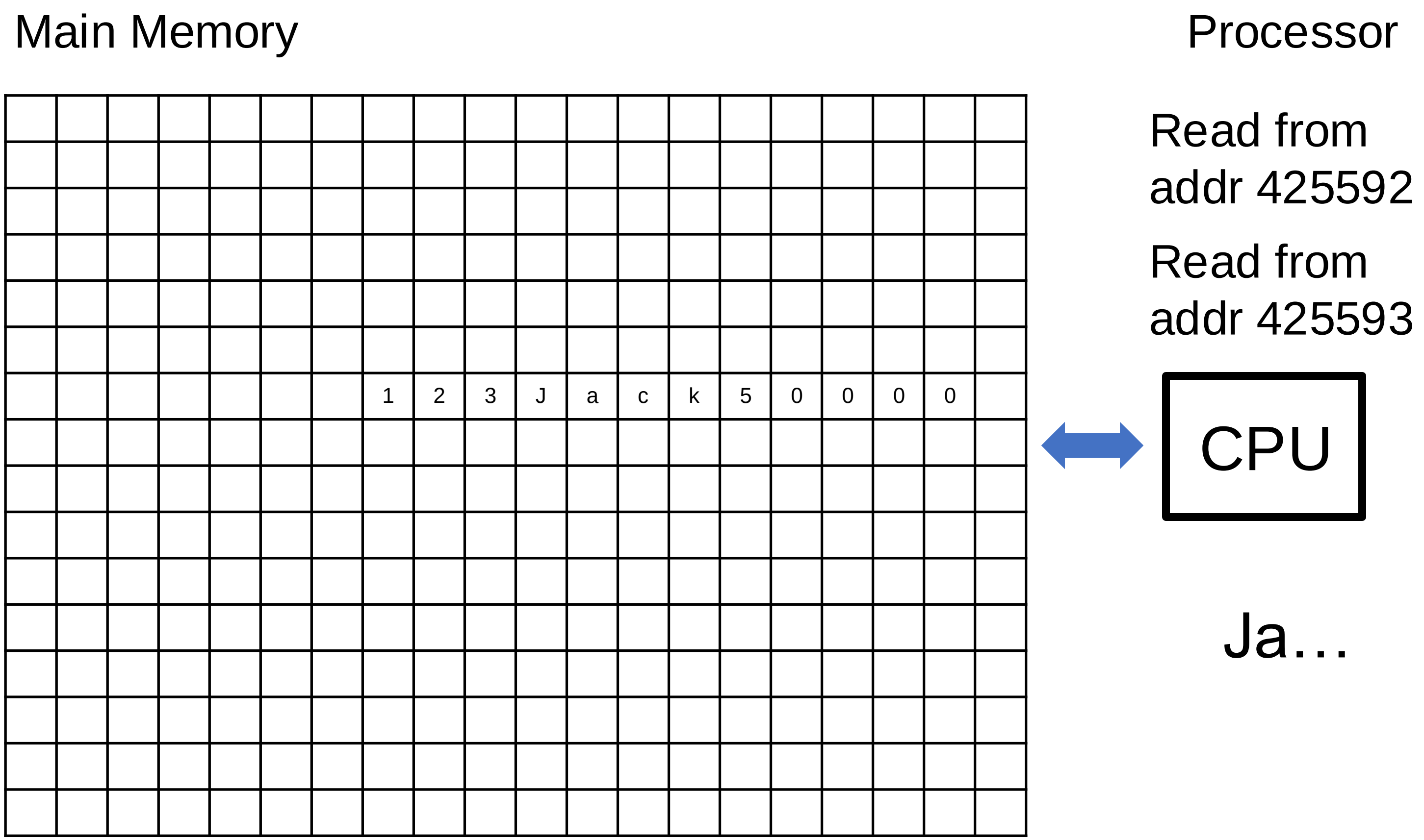

To read from Main Memory, the CPU needs to know the address

# How the Main Memory Works




To read from Main Memory, the CPU needs to know the address

# How the Main Memory Works




To read from Main Memory, the CPU needs to know the address

# Hash Tables

Array: map indices to memory locations

- A[0], A[1], A[2], … sequential in memory
- To read A[5262]: read from [address of A] + 5262

# Hash Tables

Array: map indices to memory locations

- A[0], A[1], A[2], … sequential in memory
- To read A[5262]: read from [address of A] + 5262

How to map strings to memory locations?

- A[“alice”], A[“bob”],A[“carl”]…???
- How do we read A[“jack”]?

# Hash Tables

Array: map indices to memory locations

- A[0], A[1], A[2], … sequential in memory
- To read A[5262]: read from [address of A] + 5262

How to map strings to memory locations?

- A[“alice”], A[“bob”],A[“carl”]…???
- How do we read A[“jack”]?

Hash table: simulates an array indexed by strings

# Hash Tables

A (naïve!!) hash function:

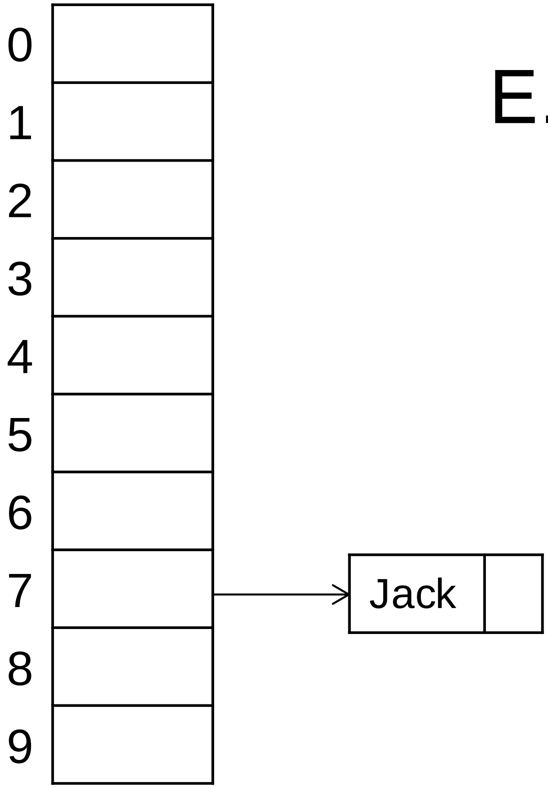
$$h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10$$

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |

# Hash Tables

A (naïve!!) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10

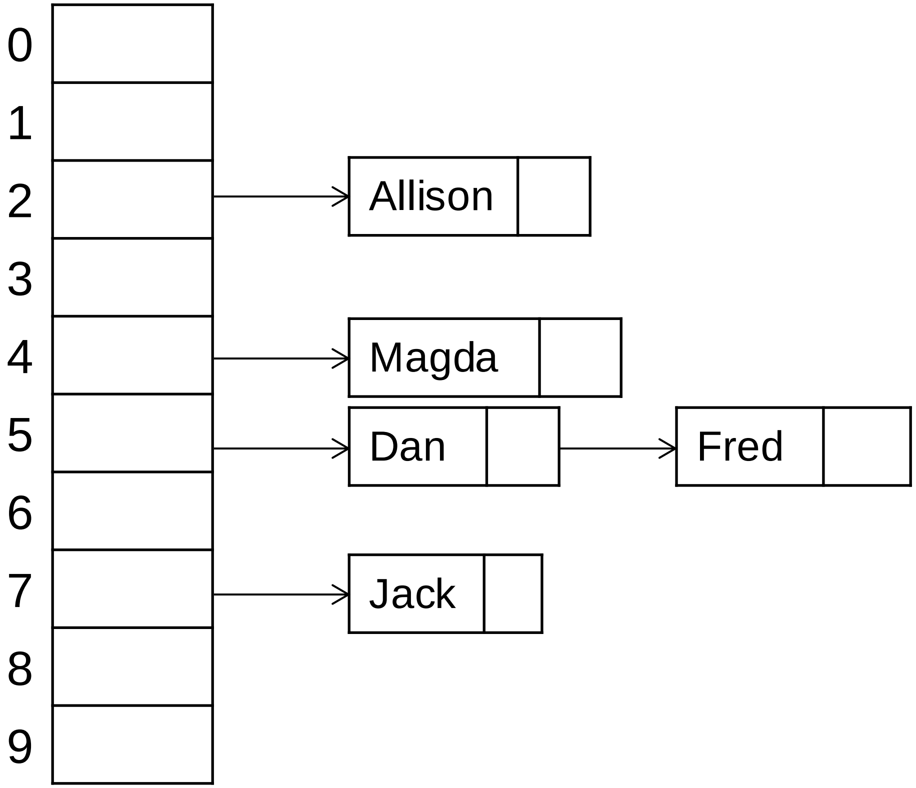



E.g. h(“Jack”) = (‘J’+’a’+’c’+’k’) mod 10
= (74+97+99+107) mod 10
= 377 mod 10 = 7

# Hash Tables

A (naïve!!) hash function:

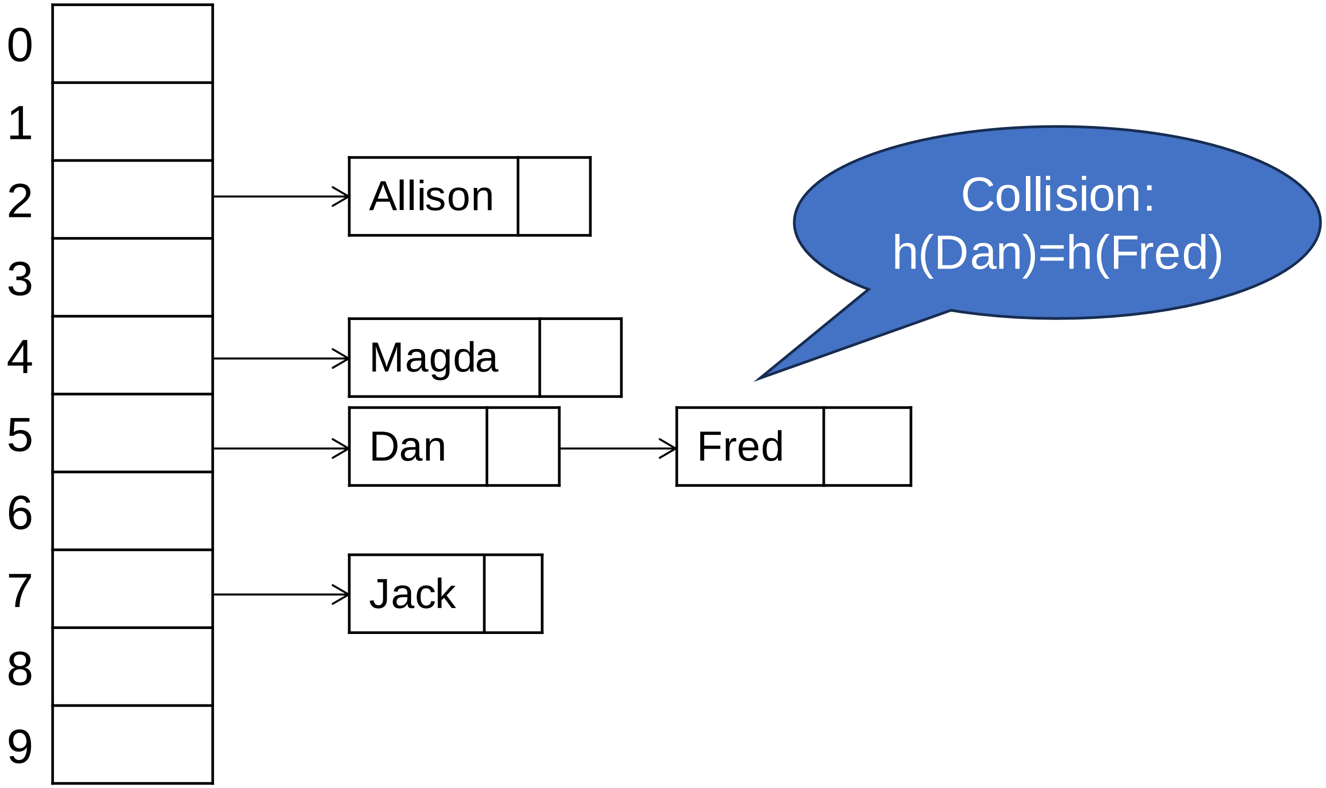
h(“abc”) = (‘a’+’b’+’c’) mod 10



# Hash Tables

A (naïve!!) hash function:

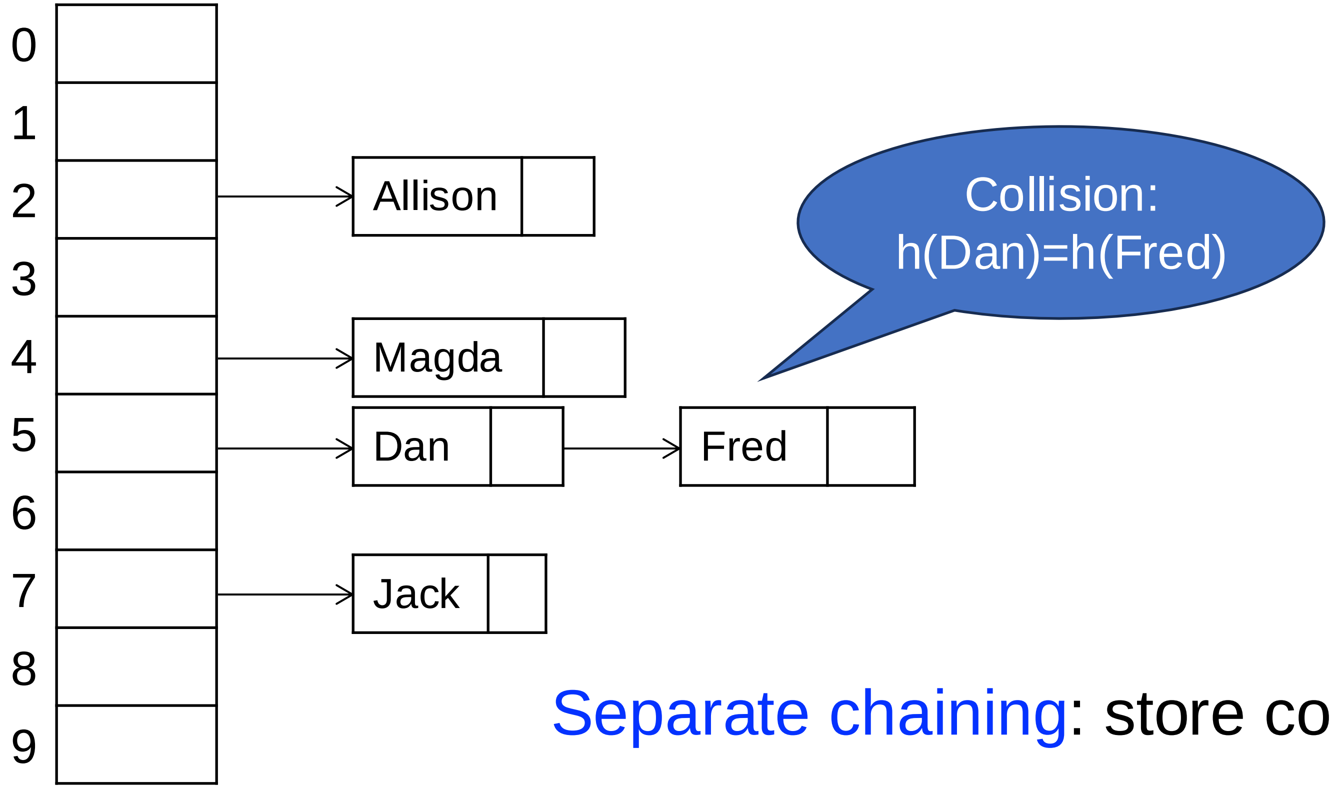
$$h(\text{"abc"}) = (\text{'a'}+\text{'b'}+\text{'c'}) \bmod 10$$



# Hash Tables

A (naïve!!) hash function:

$$h(\text{"abc"}) = (\text{'a'}+\text{'b'}+\text{'c'}) \bmod 10$$

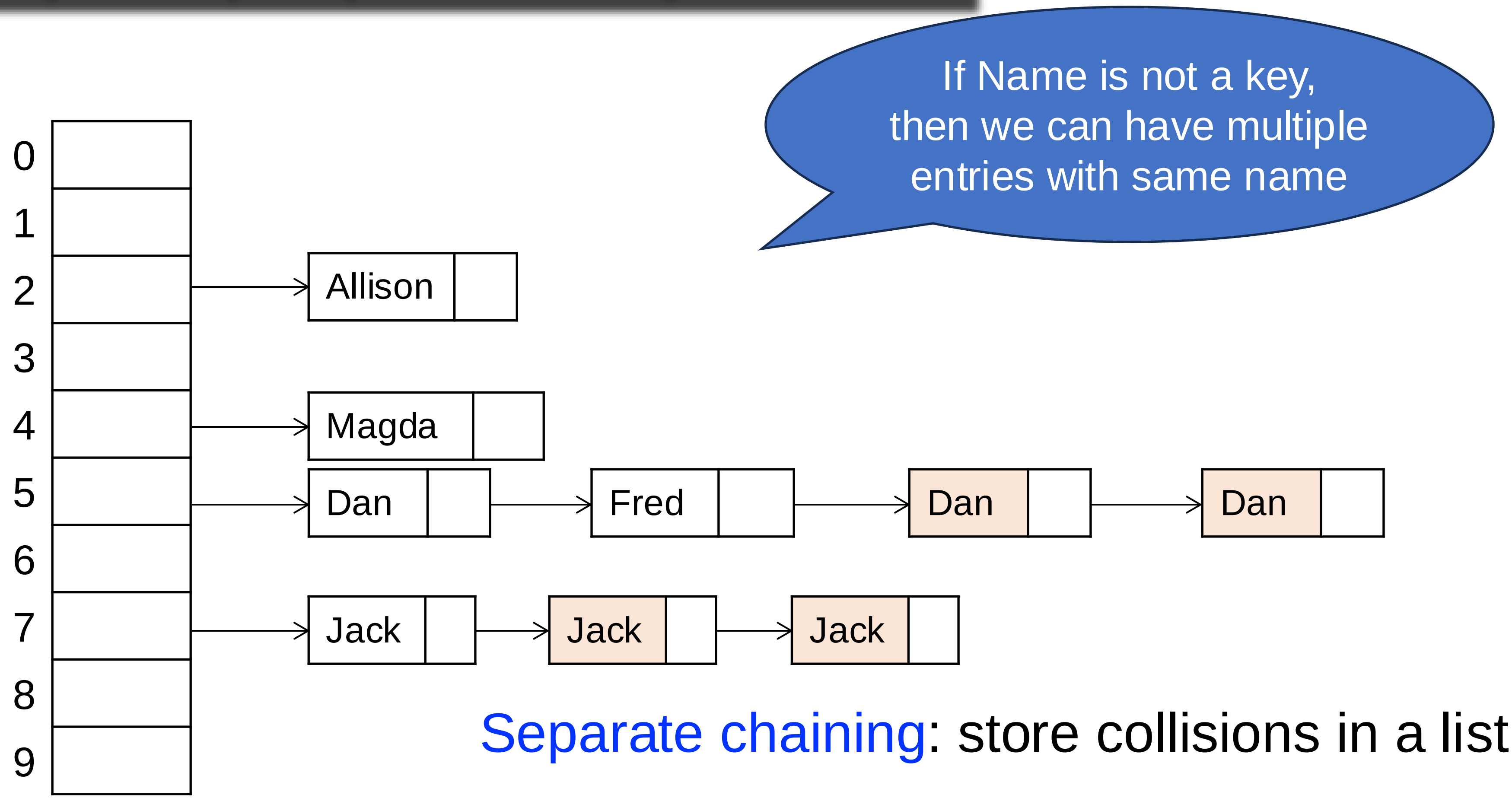



Separate chaining: store collisions in a list

# Hash Tables

A (naïve!!) hash function:

$$h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10$$



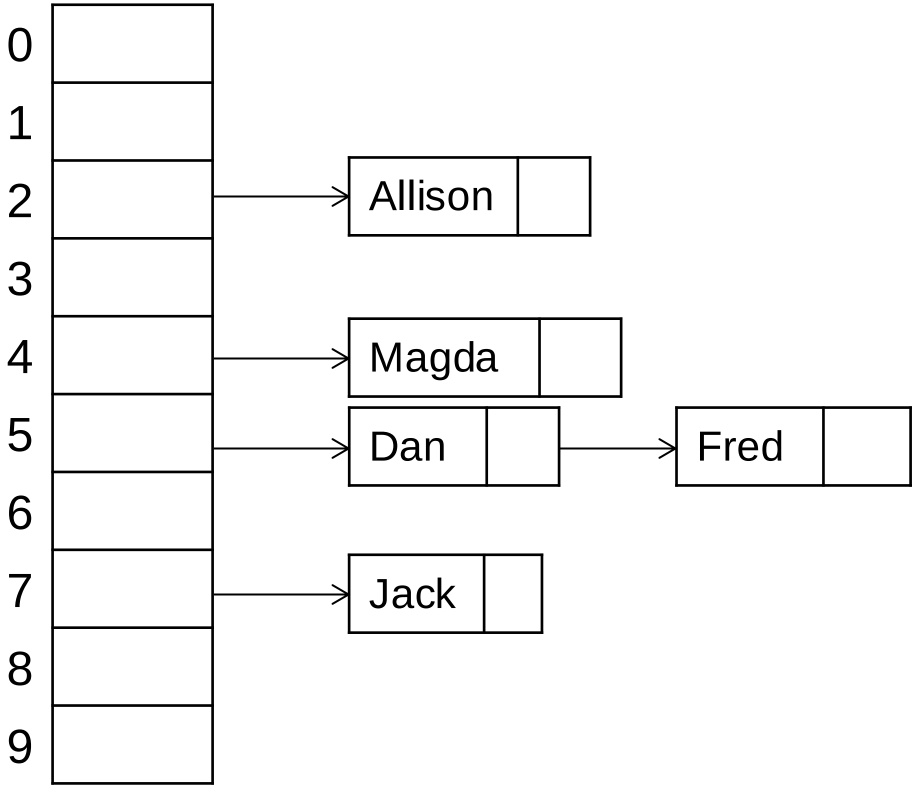

Separate chaining: store collisions in a list

# Hash Tables

A (naïve!!) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10

find(“Magda”)

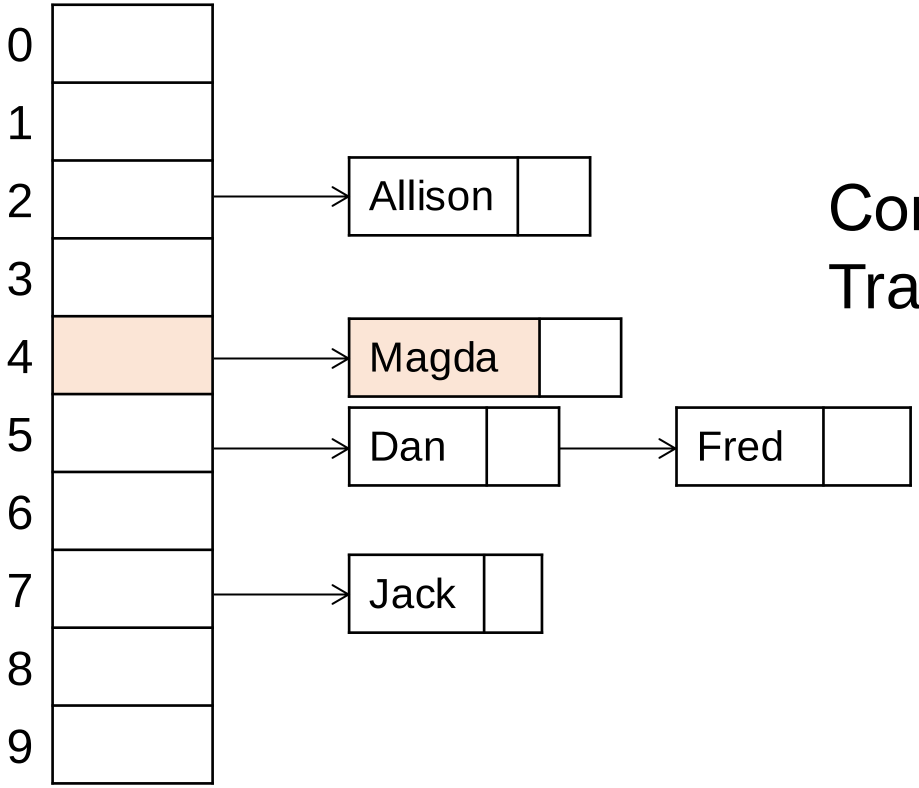


# Hash Tables

A (naïve!!) hash function:

$$h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10$$

find(“Magda”)




Compute h(“Magda”)=4

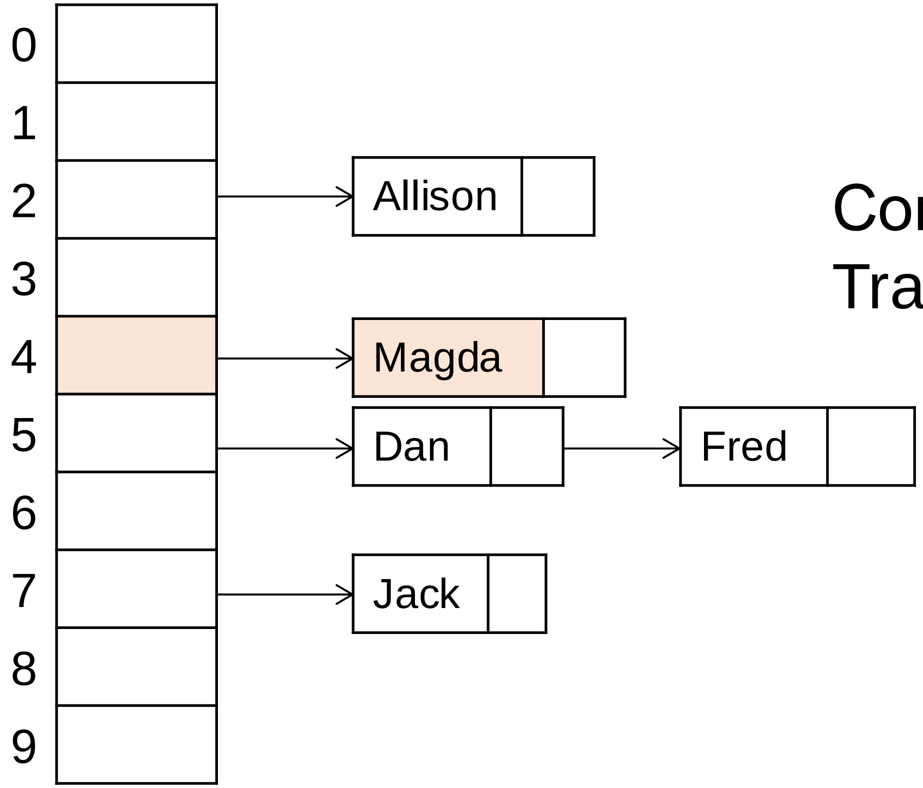
Traverse the short list at offset 4

# Hash Tables

A (naïve!!) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10

find(“Magda”)




Compute h(“Magda”)=4
Traverse the short list at offset 4

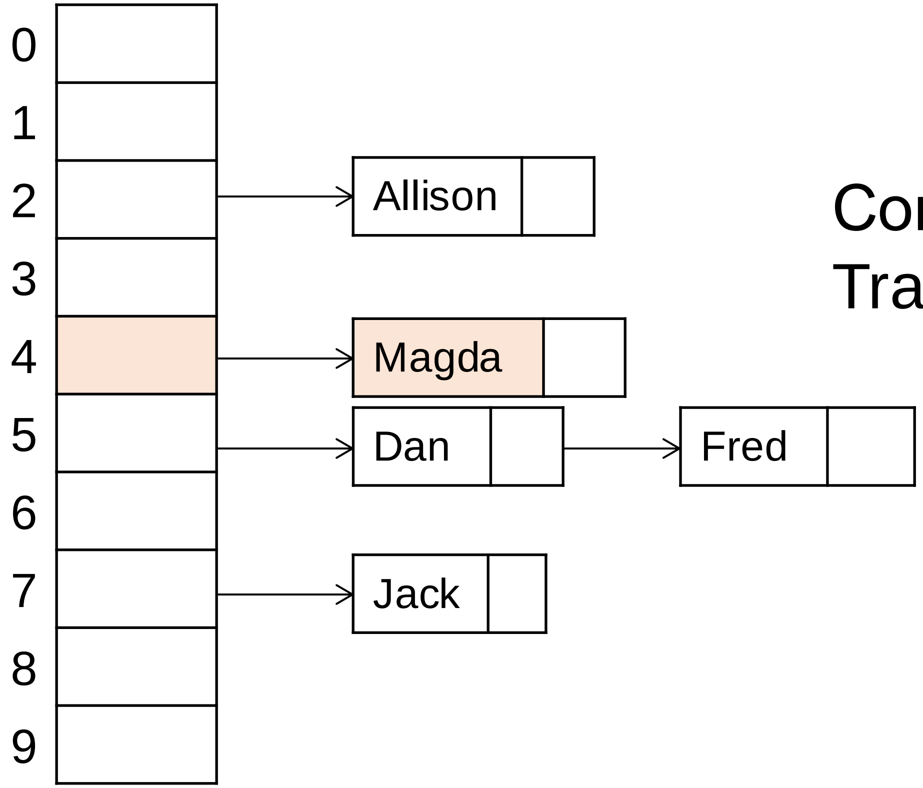
Time = O(1)

# Hash Tables

A (naïve!!) hash function:

h("abc") = ('a'+'b'+'c') mod 10

find("Magda")




Compute h("Magda")=4
Traverse the short list at offset 4

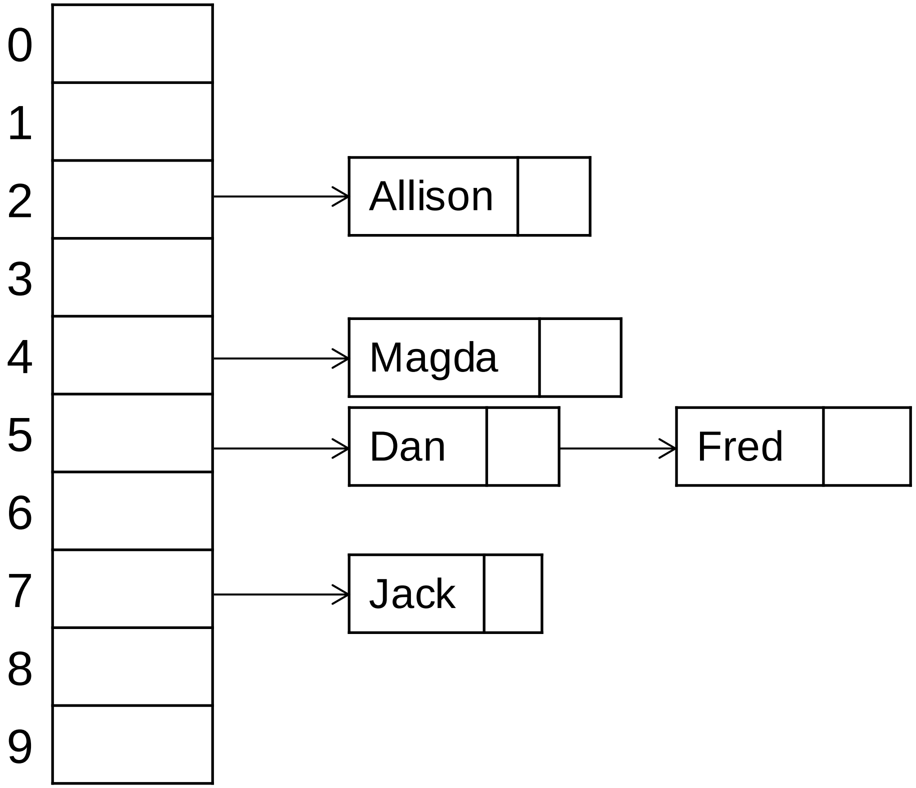
Time = O(1)

…but can becomes O(n)
if the collision list is long

# Hash Tables

A (naïve!!) hash function:

$$h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10$$



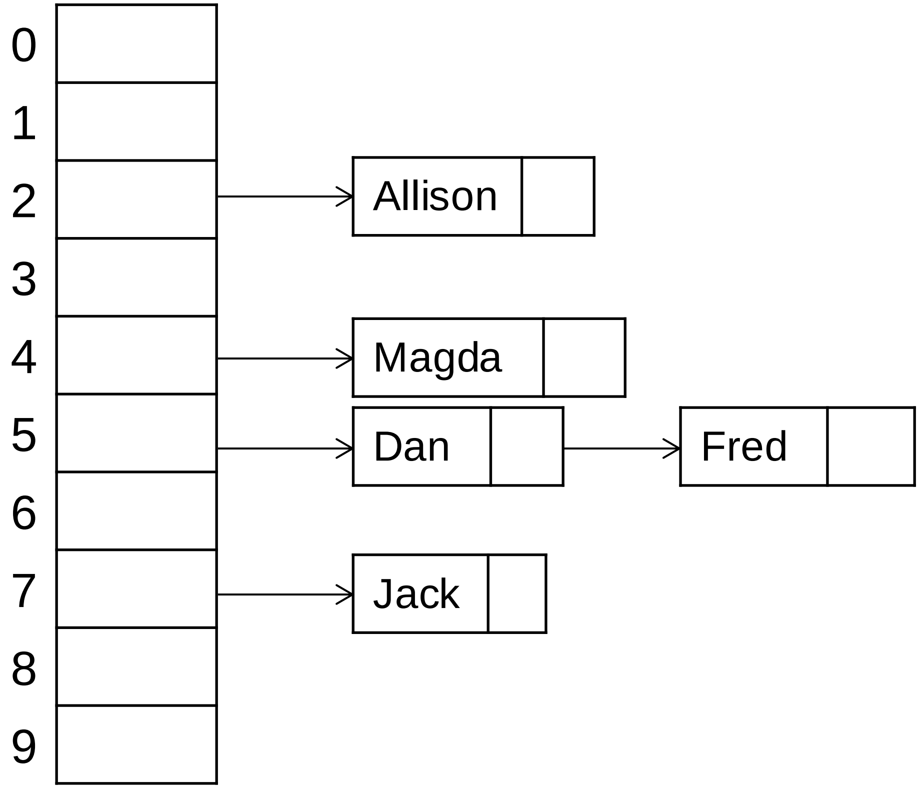

find(“Magda”)

insert(“Jane”)

# Hash Tables

A (naïve!!) hash function:

$$h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10$$




find(“Magda”)

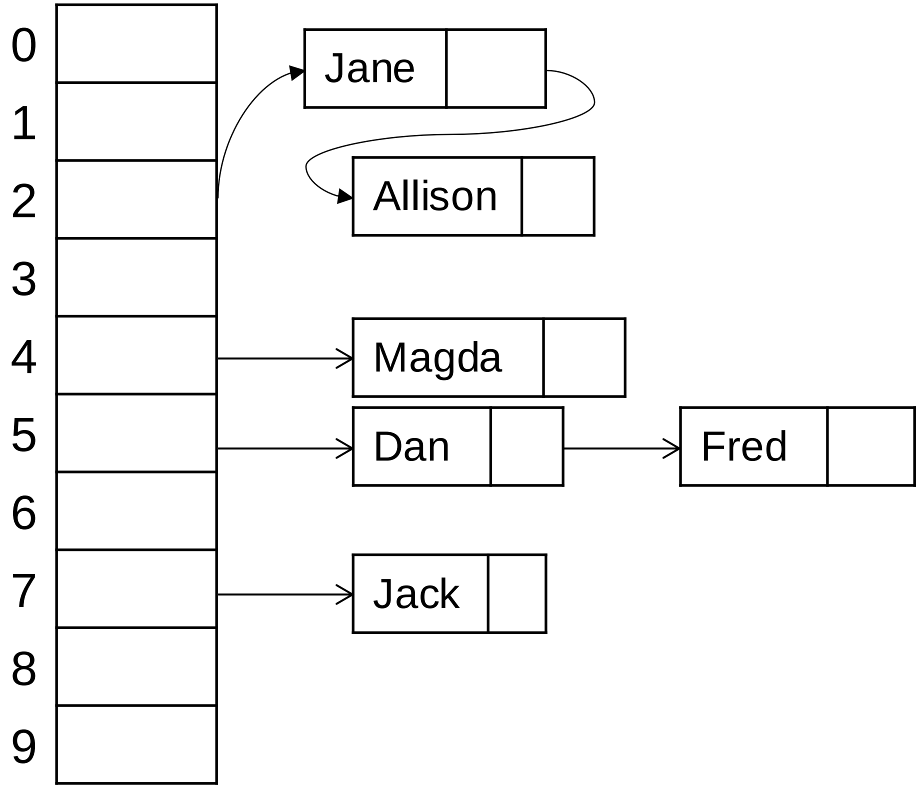
insert(“Jane”)

Compute h(“Jane”)=2
Insert into that list.
Easiest: insert at beginning

# Hash Tables

A (naïve!!) hash function:

$$h(\text{"abc"}) = (\text{'a'}+\text{'b'}+\text{'c'}) \bmod 10$$




find("Magda")

insert("Jane")

Compute h("Jane")=2
Insert into that list.
Easiest: insert at beginning

# Hash Table Takeaways

- Use good hash function, never your own. E.g. https://15445.courses.cs.cmu.edu/fall2023/slides/07-hashtables.pdf
  - Low probability of collision
- Don't use a cryptographic hash function! Why?
- The vector needs to be pre-allocated:
  - Too big: waste space
  - To small: long chains
  - Reallocation (java, python, …): expensive
- If one value occurs much more than average, we say that the data is skewed:
  - O(1)→O(N)

# 2. Hash-Join

# 2. Hash Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

# 2. Hash Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

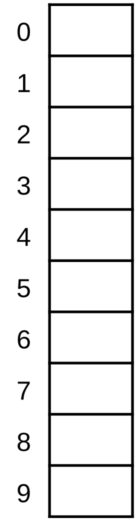
```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for x in Payroll
    insert(x.UserID,x)
```

# 2. Hash Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

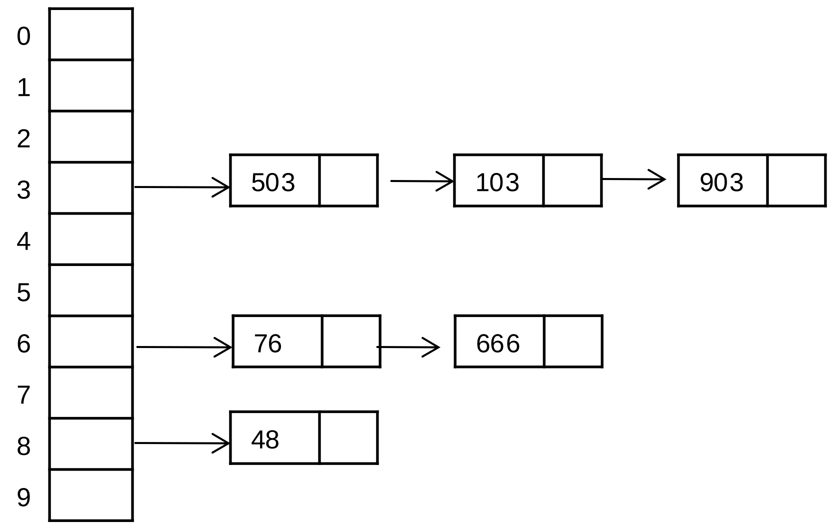



```
for x in Payroll
    insert(x.UserID,x)
```

# 2. Hash Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

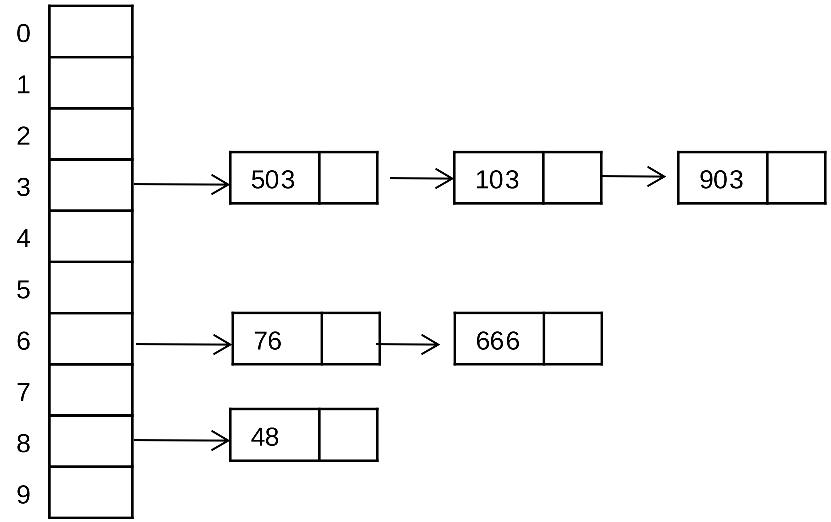



```
for x in Payroll
    insert(x.UserID,x)
```

# 2. Hash Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

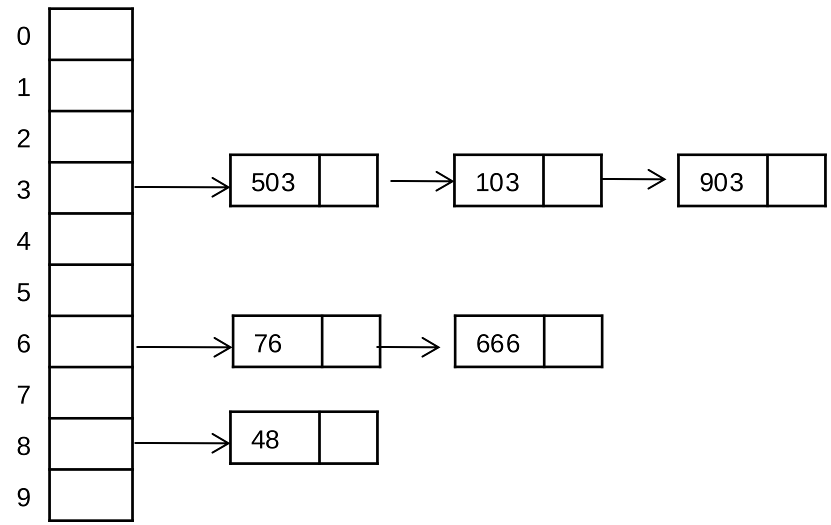



```
for x in Payroll
    insert(x.UserID,x)

for y in Regist
    x = find(y.UserId)
    output(x,y)
```

# 2. Hash Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```




”Build phase”

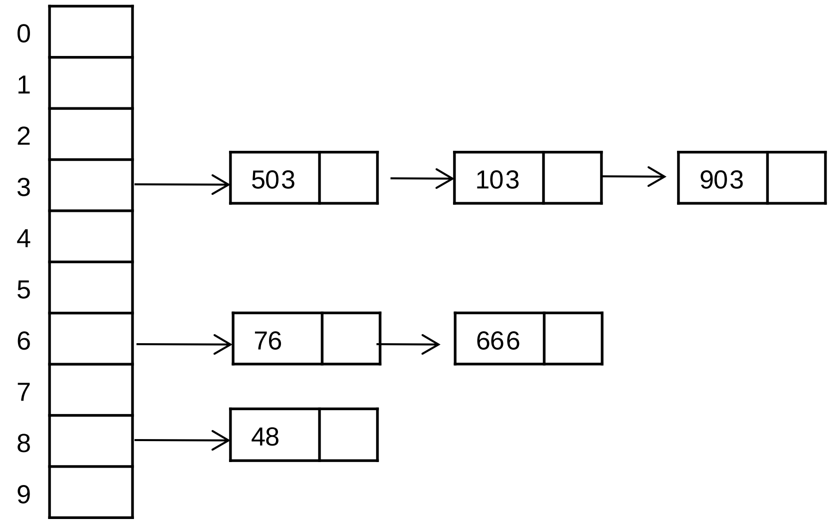
```
for x in Payroll
   insert(x.UserID,x)

for y in Regist
   x = find(y.UserId)
   output(x,y)
```

”Probe phase”

# 2. Hash Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```




"Build phase"

```
for x in Payroll
   insert(x.UserID,x)

for y in Regist
   x = find(y.UserId)
   output(x,y)
```

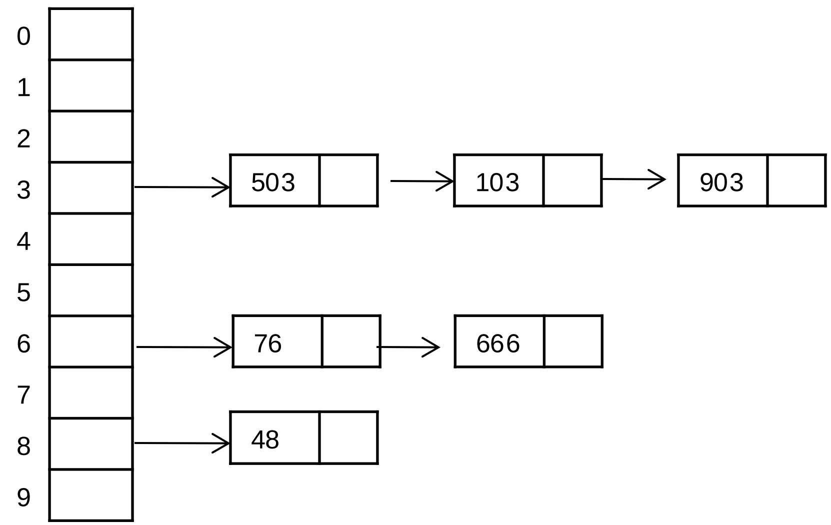
"Probe phase"

**If |Payroll| = |Regist| = n, what is the runtime?**

# 2. Hash Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```




"Build phase"

```
for x in Payroll
   insert(x.UserID,x)

for y in Regist
   x = find(y.UserId)
   output(x,y)
```

"Probe phase"

If |Payroll| = |Regist| = n, what is the runtime?

O(n)

# 2. Hash Join

$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$

Switch join order

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

# 2. Hash Join

$$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

Switch join order

```
for y in Regist
    insert(y.UserID,x)
```

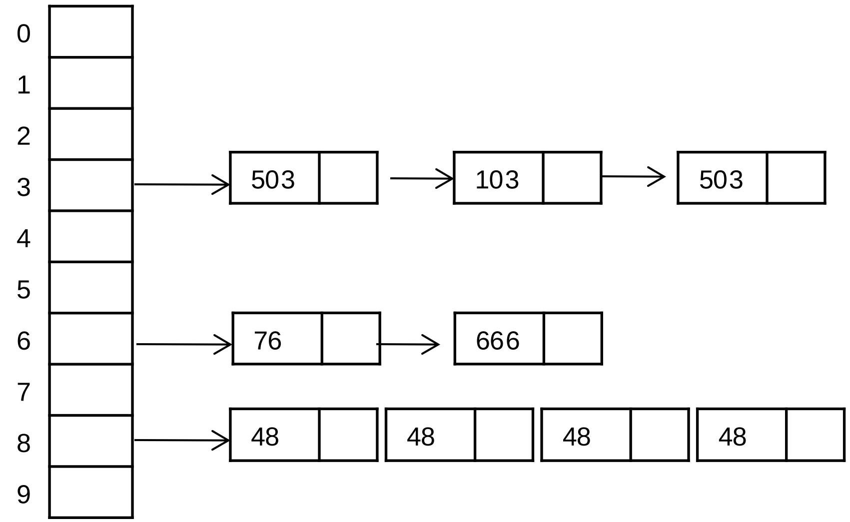
# 2. Hash Join

$$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$$

Switch join order

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for y in Regist
    insert(y.UserID,x)
```



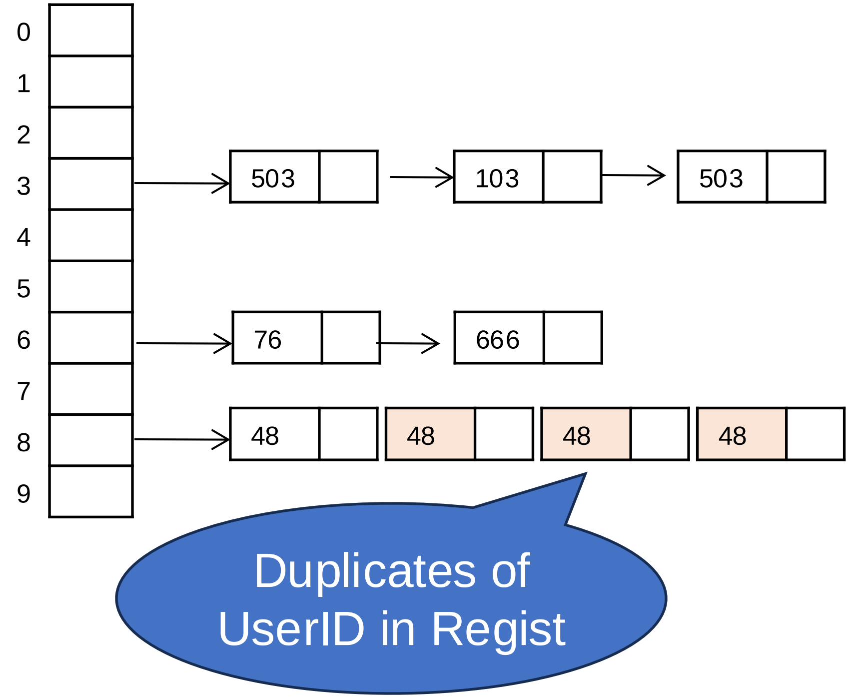
# 2. Hash Join

$$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$$

Switch join order

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for y in Regist
    insert(y.UserID,x)
```

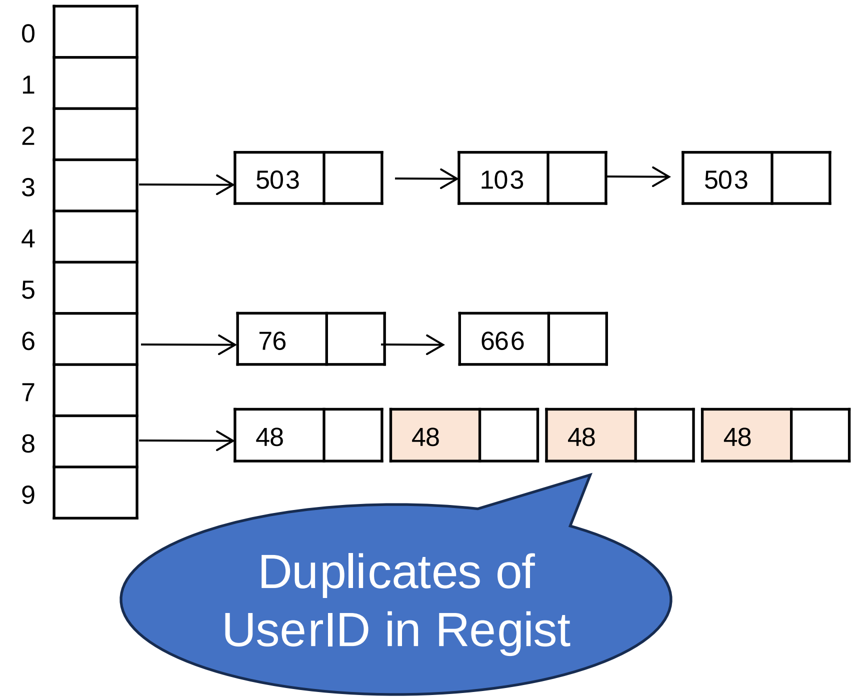


# 2. Hash Join

$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$

Switch join order

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```




```
for y in Regist
    insert(y.UserID,x)

for x in Payroll
  for y in find(y.UserId)
    output(x,y)
```

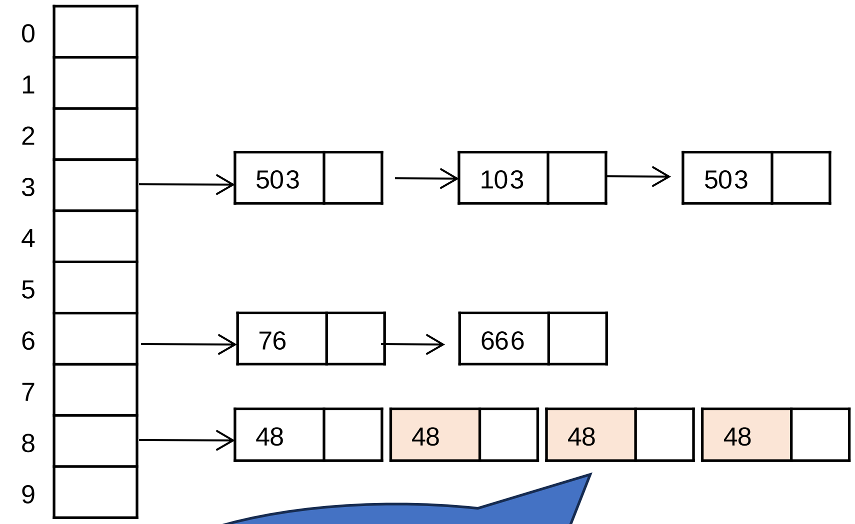
# 2. Hash Join

Regist $\bowtie_{\text{UserID=UserID}}$ Payroll

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

Switch join order

```
for y in Regist
    insert(y.UserID,x)

for x in Payroll
   for y in find(y.UserId)
      output(x,y)
```




Duplicates of UserID in Regist

Runtime: O(n)

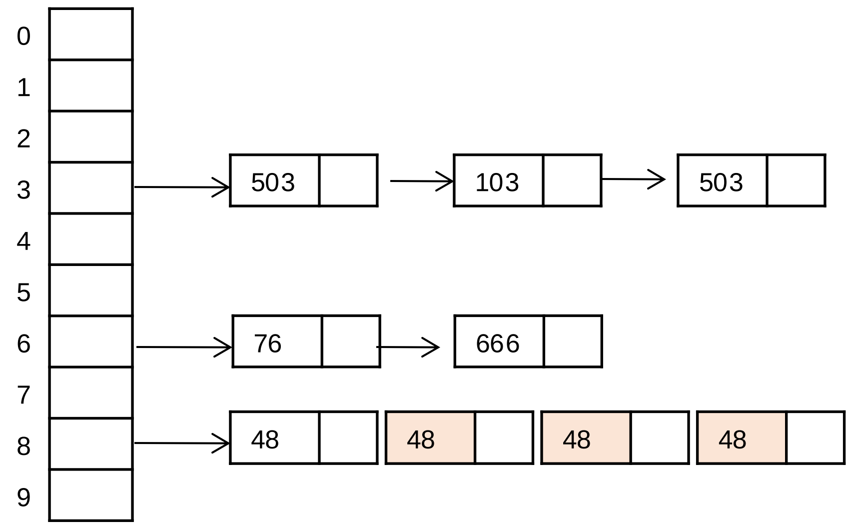
But may be O($n^2$)

# 2. Hash Join

$$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$$

Payroll(<u>UserID</u>,Name,Job,Salary)
Regist(UserID,Car)

Switch join order




```
for y in Regist
   insert(y.UserID,x)

for x in Payroll
  for y in find(y.UserId)
    output(x,y)
```

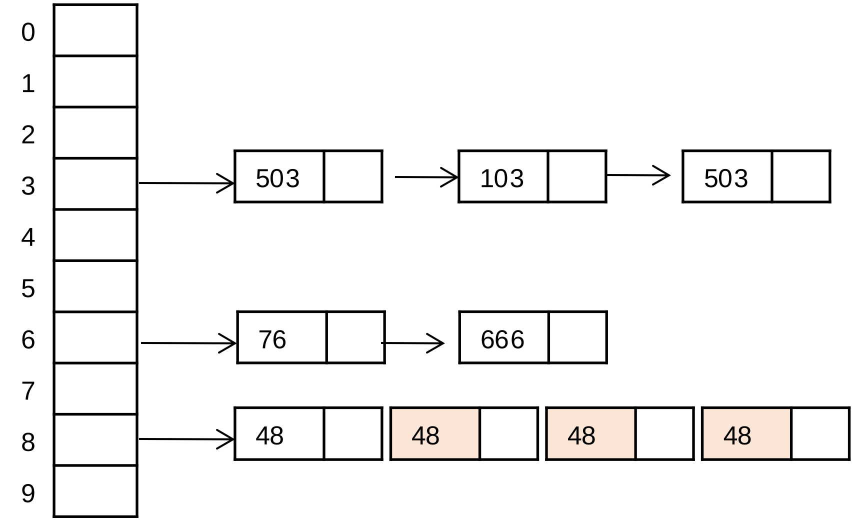
What could be the advantage of this join order?

# 2. Hash Join

$\text{Regist} \bowtie_{\text{UserID}=\text{UserID}} \text{Payroll}$

Switch join order

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```




```
for y in Regist
    insert(y.UserID,x)

for x in Payroll
   for y in find(y.UserId)
      output(x,y)
```

What could be the advantage of this join order?

Is better than previous order when |Regist| << |Payroll|

# Discussion

Hash-Join: very efficient, usually runs in time O(n)

Performance degrades when:

- Data is skewed, e.g. join on non-key
- Outer table (for which we build the hash table) does not fit in main memory

# 3. Merge-Join

# Review: Sorting

- Given an array, sort it in increasingly

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 74 | 7 | 45 | 99 | 2 | 90 | 19 | 87 | 61 | 82 |

# Review: Sorting

- Given an array, sort it in increasingly

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 74 | 7 | 45 | 99 | 2 | 90 | 19 | 87 | 61 | 82 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 19 | 45 | 61 | 74 | 82 | 87 | 90 | 99 |

# Review: Sorting

- Given an array, sort it in increasingly

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 74 | 7 | 45 | 99 | 2 | 90 | 19 | 87 | 61 | 82 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 19 | 45 | 61 | 74 | 82 | 87 | 90 | 99 |

Simple algorithms: $O(N^2)$
Quicksort: $O(N \log N)$
Mergesort: $O(N \log N)$

# 3. Merge Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
```

# 3. Merge Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
while y!=NULL do:
```

# 3. Merge Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
while y!=NULL do:
  case:
    x.UserID < y.UserID: ??????
    x.UserID = y.UserID: ??????
    x.UserID > y.UserID: ??????
```

# 3. Merge Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
while y!=NULL do:
  case:
    x.UserID < y.UserID: x.next();
    x.UserID = y.UserID: ??????
    x.UserID > y.UserID: ??????
```

# 3. Merge Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
while y!=NULL do:
  case:
    x.UserID < y.UserID: x.next();
    x.UserID = y.UserID: output(x,y); y.next();
    x.UserID > y.UserID: ??????
```

# 3. Merge Join

$$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
while y!=NULL do:
  case:
    x.UserID < y.UserID: x.next();
    x.UserID = y.UserID: output(x,y); y.next();
    x.UserID > y.UserID: y.next();
```

# 3. Merge Join

$\text{Payroll} \bowtie_{\text{UserID}=\text{UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

If |Payroll| = |Regist| = n
then runtime = O(n log n)

```
sort(Payroll); sort(Regist);
x = Payroll.first()
y = Regist.first()
while y!=NULL do:
  case:
    x.UserID < y.UserID: x.next();
    x.UserID = y.UserID: output(x,y); y.next();
    x.UserID > y.UserID: y.next();
```

# Summary

Three join algorithms:

- Nested loop: always $O(n^2)$
- Hash join: usually $O(n)$, but can be $O(n^2)$
- Merge join: $O(n \log n)$