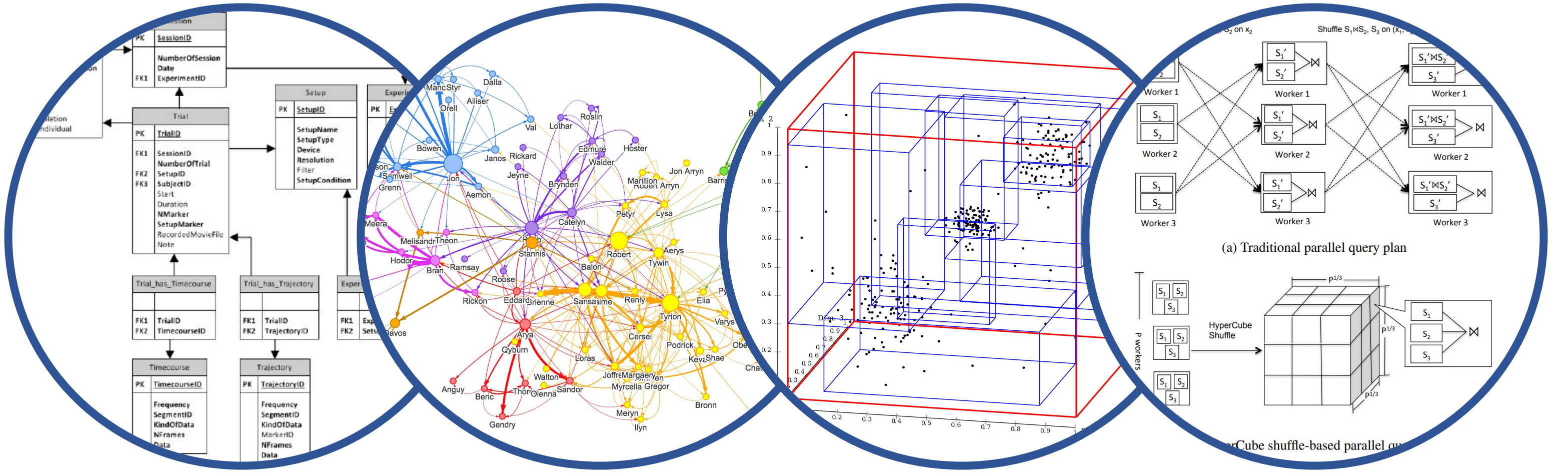


Introduction to Data Management

Query Cost Estimation

Alyssa Pittman
Based on slides by Jonathan Leang, Shana Hutchinson, Dan Suciu, et al

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

Announcements

- HW5 out
  - You know how to write the SQL queries, but it’s a lot of code!
  - Transactions can be hard to debug.

Midterm results

- Scores released today via Gradescope
- Solutions on the website
- Regrade requests open until February 26
  - Please be specific/descriptive when asking for a question to be regraded

Goals for Today

- Move to a short unit on RDBMS optimization
- Learn how an RDMS translates a logical query plan to a physical query plan and executes it

Outline

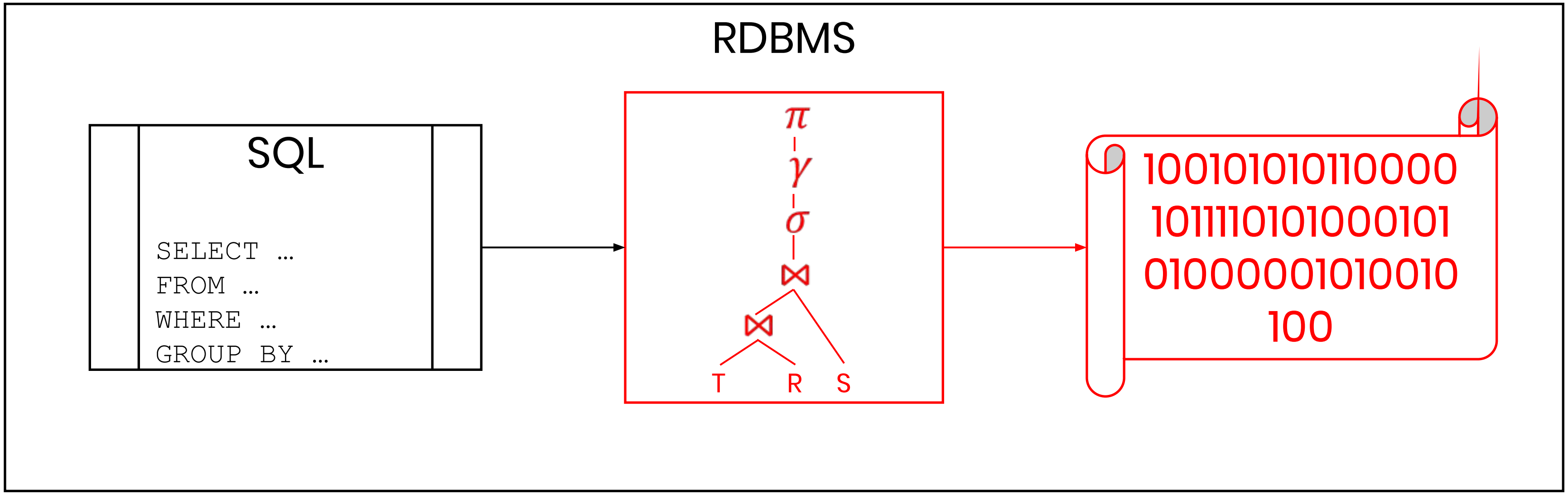
- Query execution
- Cost estimation ideas and assumptions
- Join algorithm analyses
- Basic cardinality estimation

Query Optimization

- So you wrote a SQL query…
  - SQL only tells the computer *what* you want
  - RDBMS needs to find a good way to actually do it

Logical vs Physical Plans

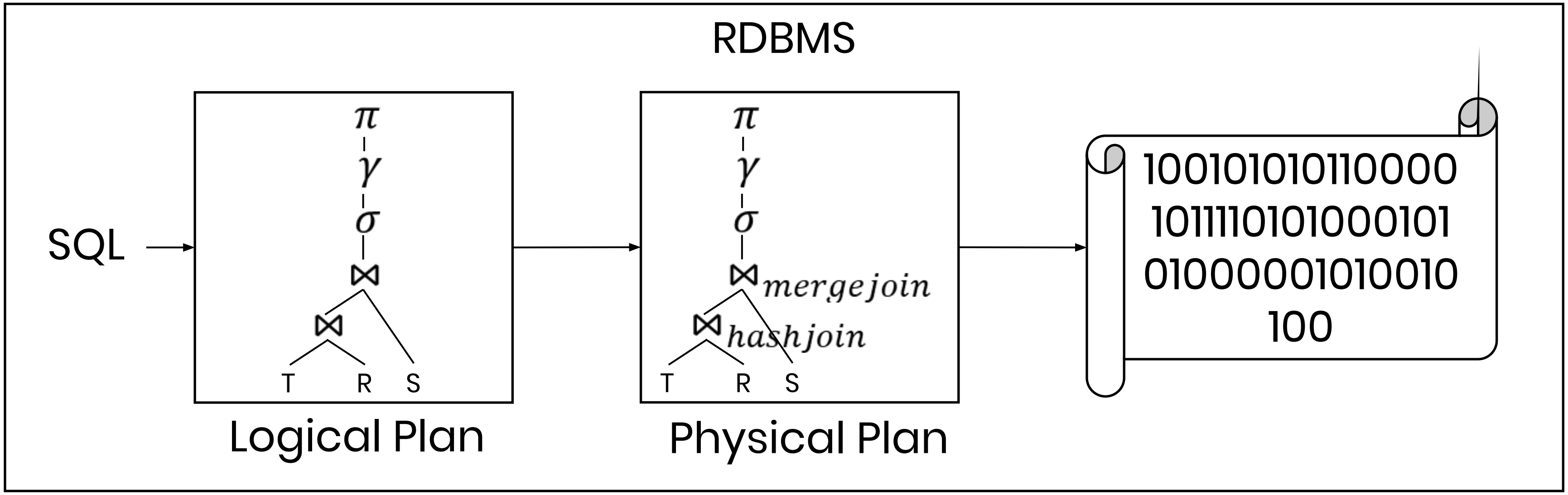
- SQL is translated into RA
- RA (logical plan) does not fully describe execution
- RA *with algorithms* (physical plan) is needed



```
SELECT …
FROM …
WHERE …
GROUP BY …
```

Logical vs Physical Plans

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



Disclaimer

- Cost estimation is an active research topic
- Equations and methods discussed in this class form a foundation of concepts, but usually cannot compare to a commercialized solution

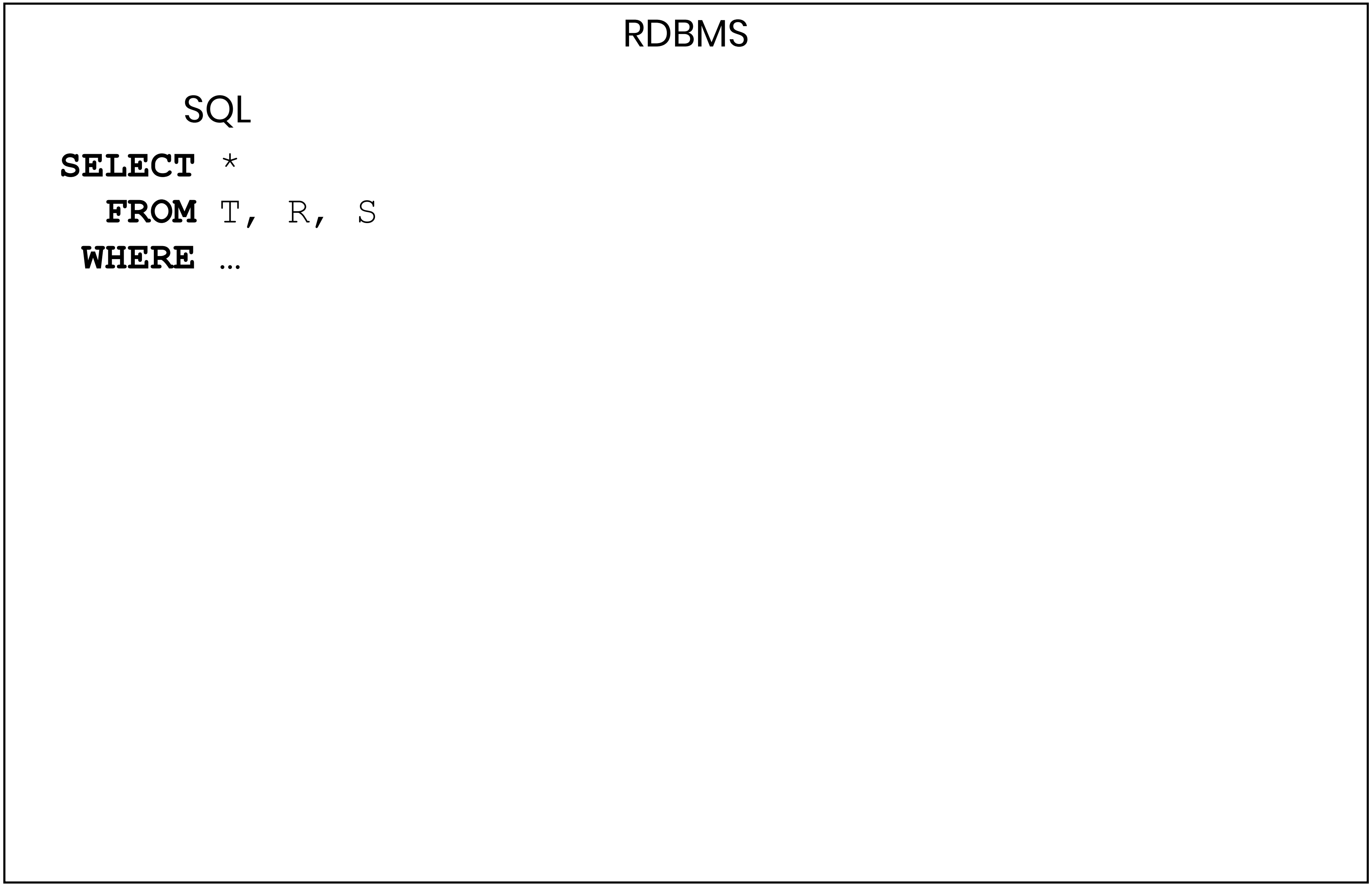
Plan Enumeration

RDBMs optimize by selecting the **least cost plan**

- SQL □ RA
- RA □ Set of eq. RA
- Set of eq. RA □ Set of physical plans
- Set of physical plans □ The least cost plan

…Execute!

Plan Enumeration



RDBMS

SQL

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

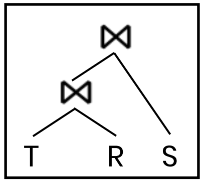
Plan Enumeration

RDBMS

SQL

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

Logical Plan



Plan Enumeration

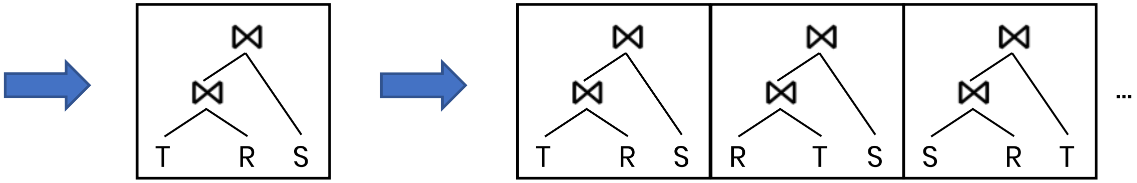
RDBMS

SQL

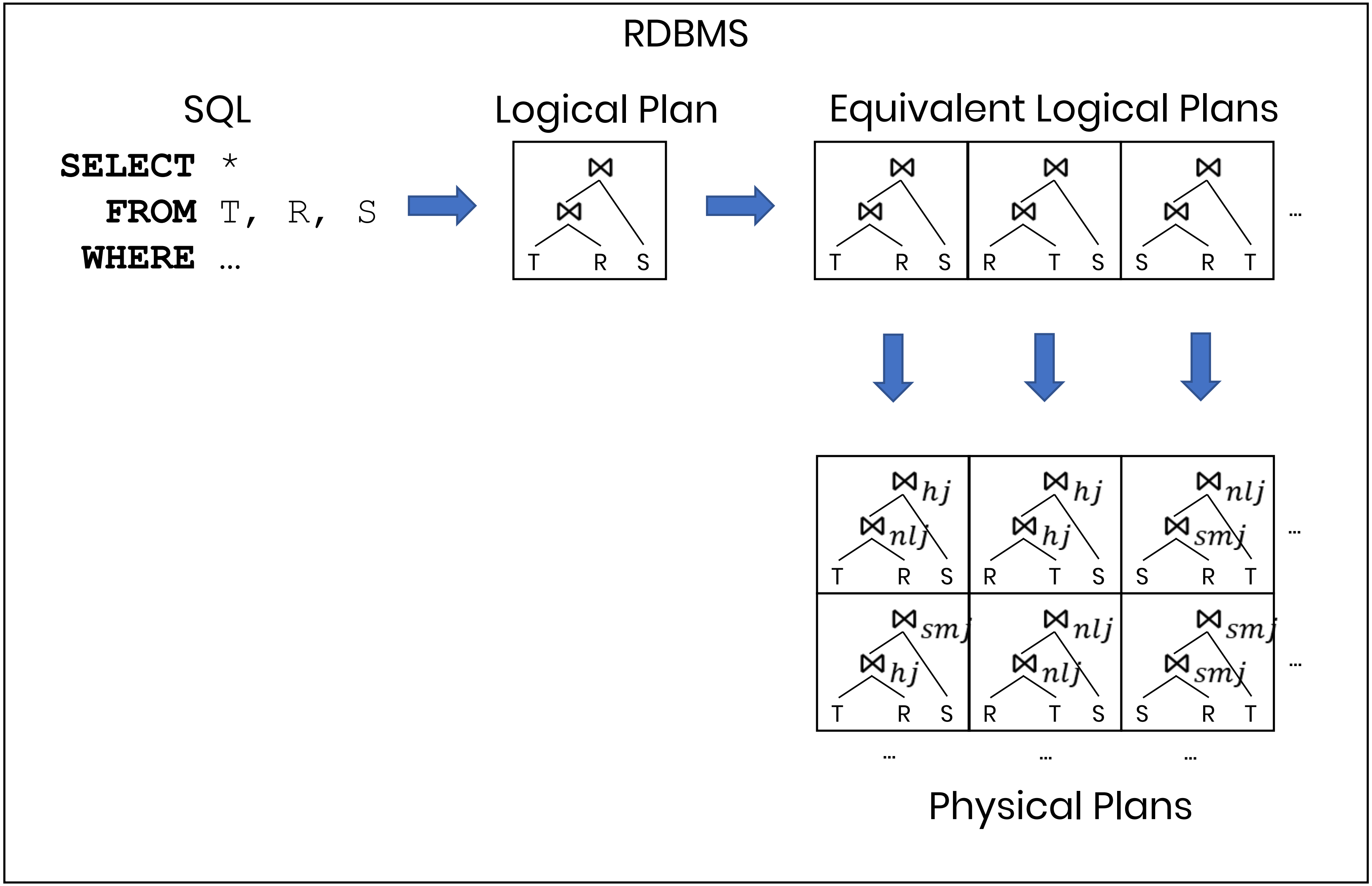
```
SELECT *
  FROM T, R, S
 WHERE …
```

Logical Plan

Equivalent Logical Plans



Plan Enumeration



RDBMS

SQL

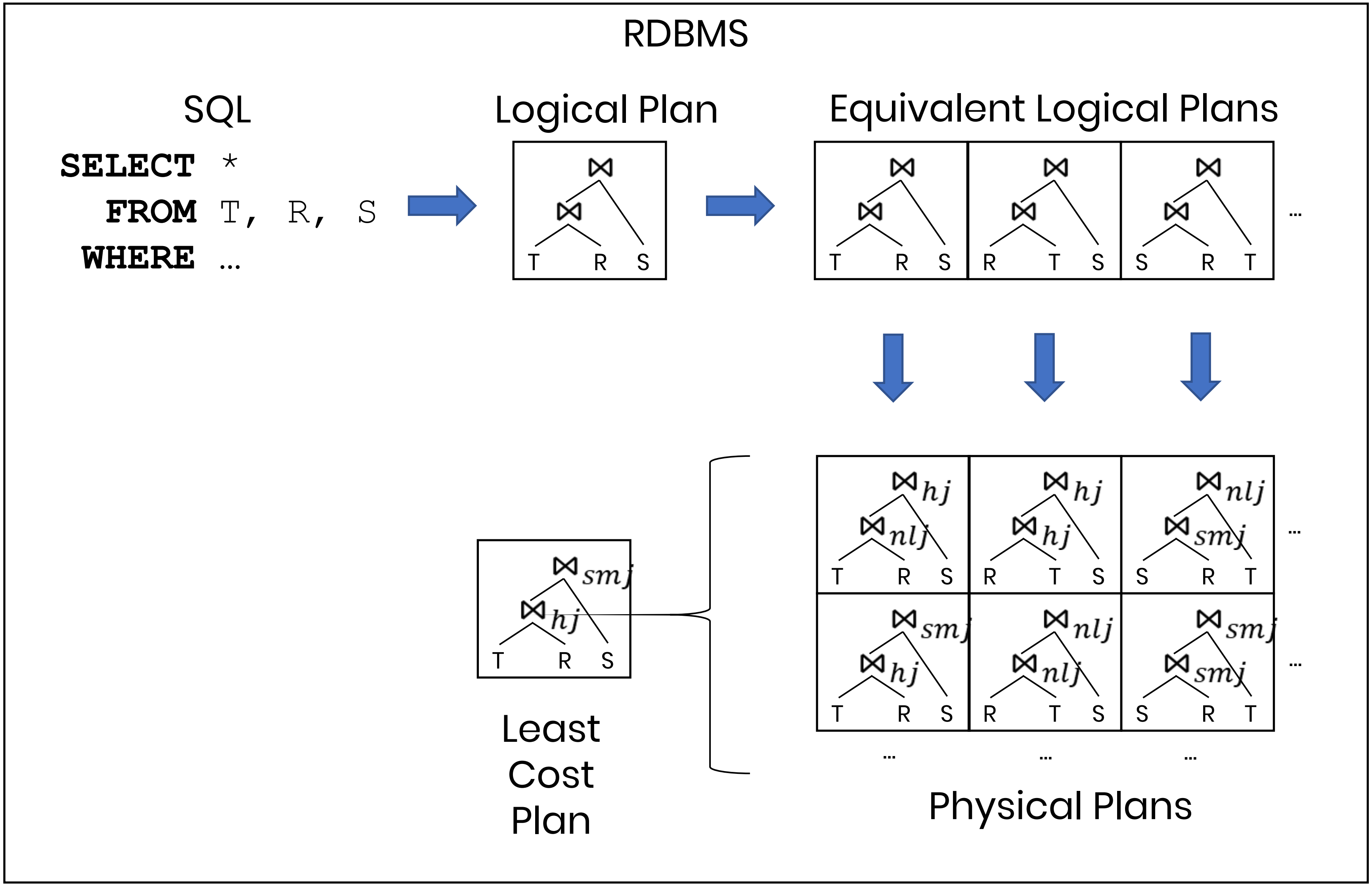
```
SELECT *
  FROM T, R, S
 WHERE ...
```

Logical Plan

Equivalent Logical Plans

Physical Plans

Plan Enumeration



RDBMS

SQL

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

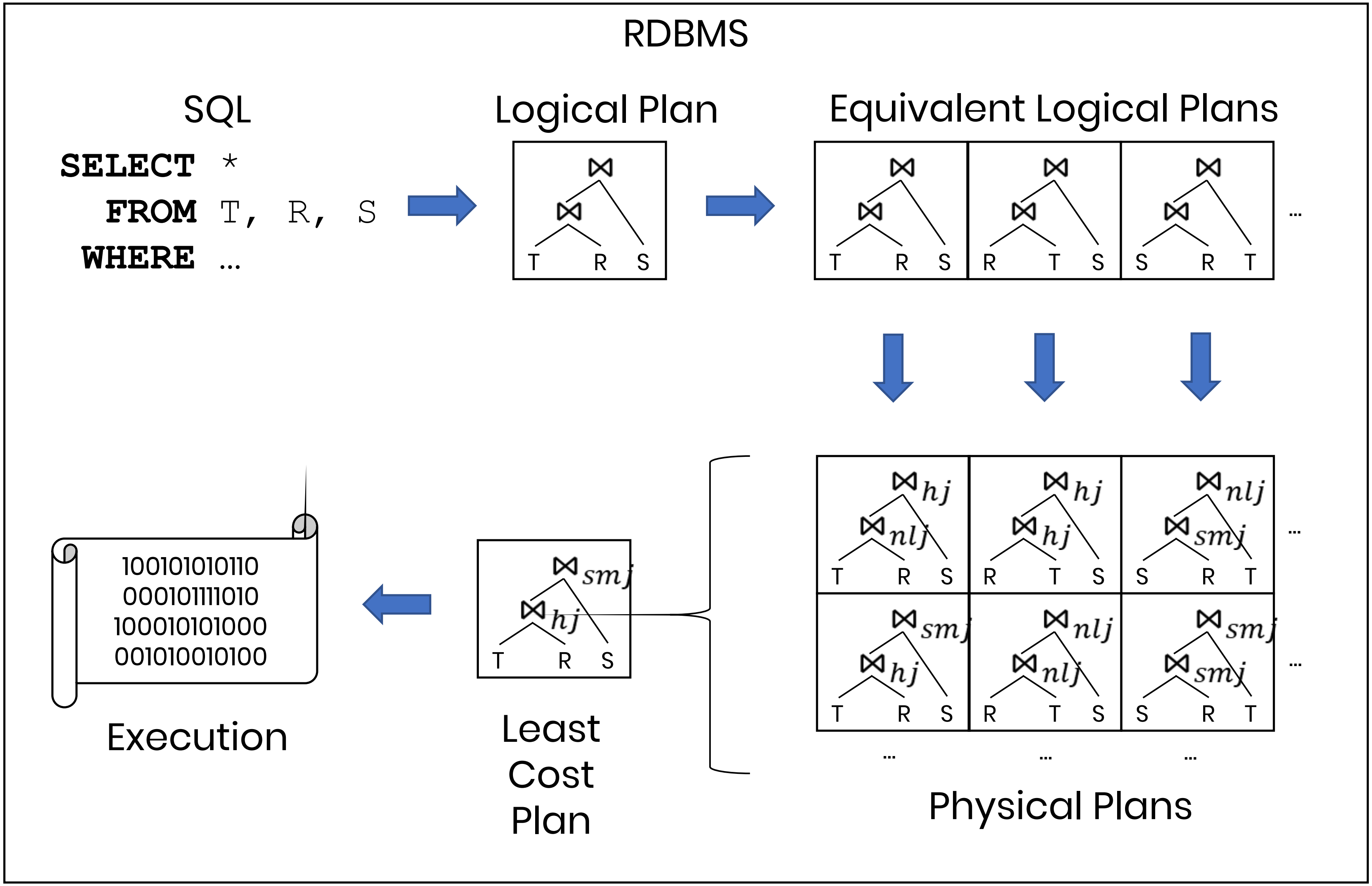
Logical Plan

Equivalent Logical Plans

Physical Plans

Least Cost Plan

Plan Enumeration



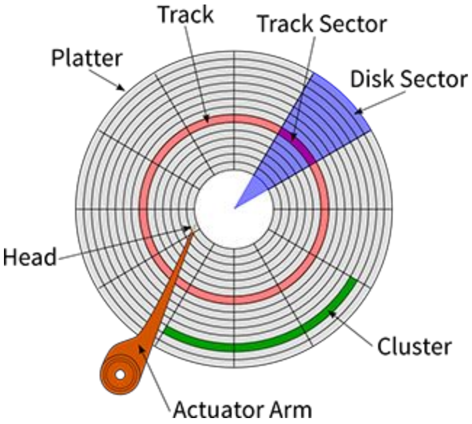
Assumptions

For this class we make a lot of assumptions

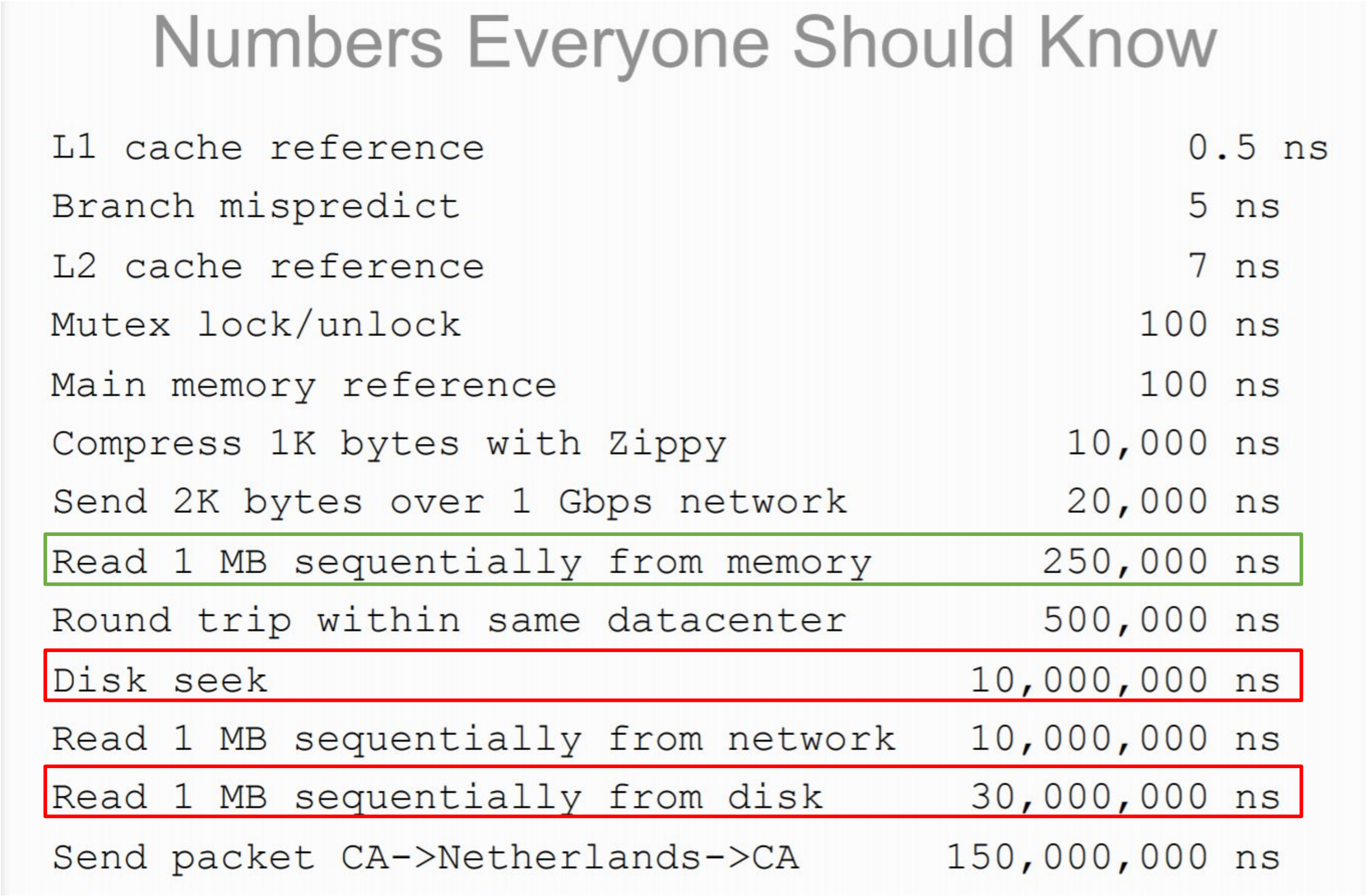
- **Disk-based storage**
  - HDD not SDD
- **Row-based storage**
  - Tuples are stored contiguously
- **IO cost** (reading from disk) only considered
  - Comprehensive cost estimation involves many factors
    - Network, disk, and CPU cost
    - Cache (main mem., L1 cache, L2 cache, disk cache, ...)
  - Reading from disk is usually the biggest component
    - One IO access is ~100000x more expensive than one main memory access
- **Cold cache** (no data preloaded)

Disk Storage

- Mechanical hard drive
- Smallest unit of memory that can be read at once is a **block**
  - Usually 512B to 4kB
- DBMS will attempt to store table files in **contiguous chunks of memory** on disk
- Sequential disk reads are faster than random ones



Disk Storage



Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Jeff Dean’s “Numbers Everyone Should Know”

Disk Storage

- Tables are stored as files
  - **Heap file** □ Unsorted tuples (this lecture)
  - **Sequential file** □ Sorted tuples (next lecture)
    - Attribute(s) sorted on is called a key (because that term isn’t overloaded…)

Making Cost Estimations

- RDBMS keeps statistics about our tables
  - **B(R)** = **# of blocks** in relation R
  - **T(R)** = **# of tuples** in relation R
  - **V(attr, R)** = **# of distinct values** of attr in R
- We only discuss **join algorithms** because they are usually the most expensive part of a query
- We only discuss **nested-loop** and **single-pass** join algorithms because cost equations get complex

Join Algorithm Summary

- Nested-Loop Join
  - Versatile
- Hash Join (single pass)
  - Fast
  - Needs at least one input to be small
- Sort-Merge Join (single pass)
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small

Join Algorithm Summary

- **Nested-Loop Join**
    - Versatile
- Hash Join (single pass)
    - Fast
    - Needs at least one input to be small
- Sort-Merge Join (single pass)
    - Fast
    - Sorts data at the same time!
    - Needs both inputs to be small

Nested Loop Join Algorithm

- Similar execution logic as nested-loop semantics

```
for each tuple t1 in R:
    for each tuple t2 in S:
        if t1 and t2 can join:
            output (t1,t2)
```

Nested Loop Join Algorithm

- Similar execution logic as nested-loop semantics

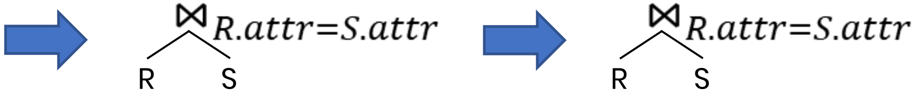
```
for each tuple t1 in R:
    for each tuple t2 in S:
        if t1 and t2 can join:
            output (t1,t2)
```

To save time, we'll read tuples from disk to memory in blocks. For fixed-size tuples, each block will have the same number of tuples.

Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



(block-at-a-time nested loop join)

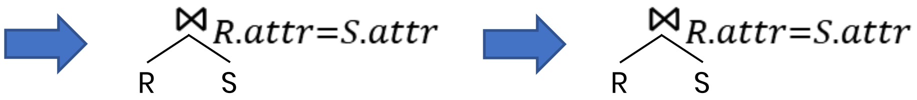
Block-at-a-time nested loop join:

```
for each block bR in R:
    for each block bS in S:
        for each tuple tR in bR:
            for each tuple tS in bS:
                if tR and tS can join:
                    output (tR,tS)
```

Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



(block-at-a-time nested loop join)

Block-at-a-time nested loop join:

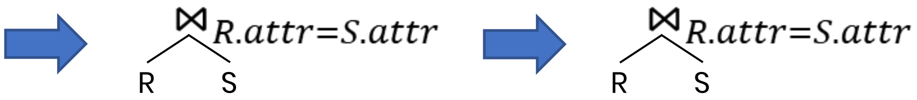
```
for each block bR in R:
    for each block bS in S:
        for each tuple tR in bR:
            for each tuple tS in bS:
                if tR and tS can join:
                    output (tR,tS)
```

Read blocks from disk to memory

Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



(block-at-a-time nested loop join)

Block-at-a-time nested loop join:

```
for each block bR in R:
    for each block bS in S:
        for each tuple tR in bR:
            for each tuple tS in bS:
                if tR and tS can join:
                    output (tR,tS)
```

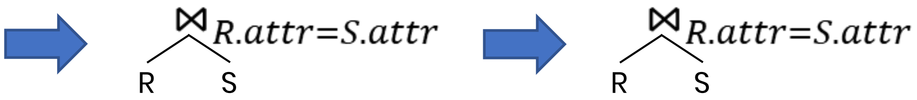
Blocks are joined in memory

Nested Loop Join Algorithm

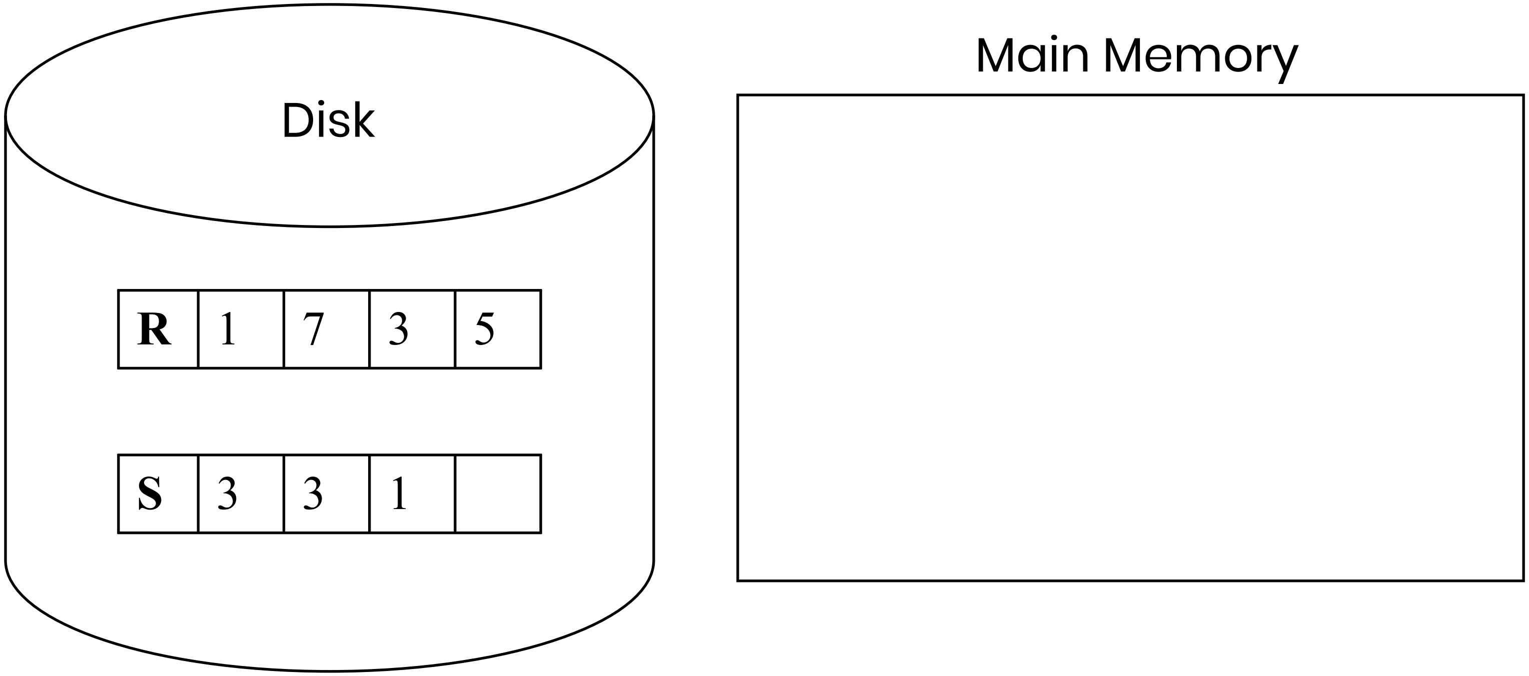
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```

(block-at-a-time nested loop join)



x □ A tuple where x is the join attribute value



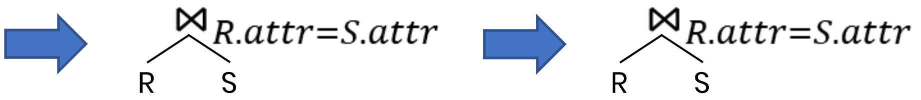
R	1	7	3	5
S	3	3	1	

Assume block size = 2 tuples

Nested Loop Join Algorithm

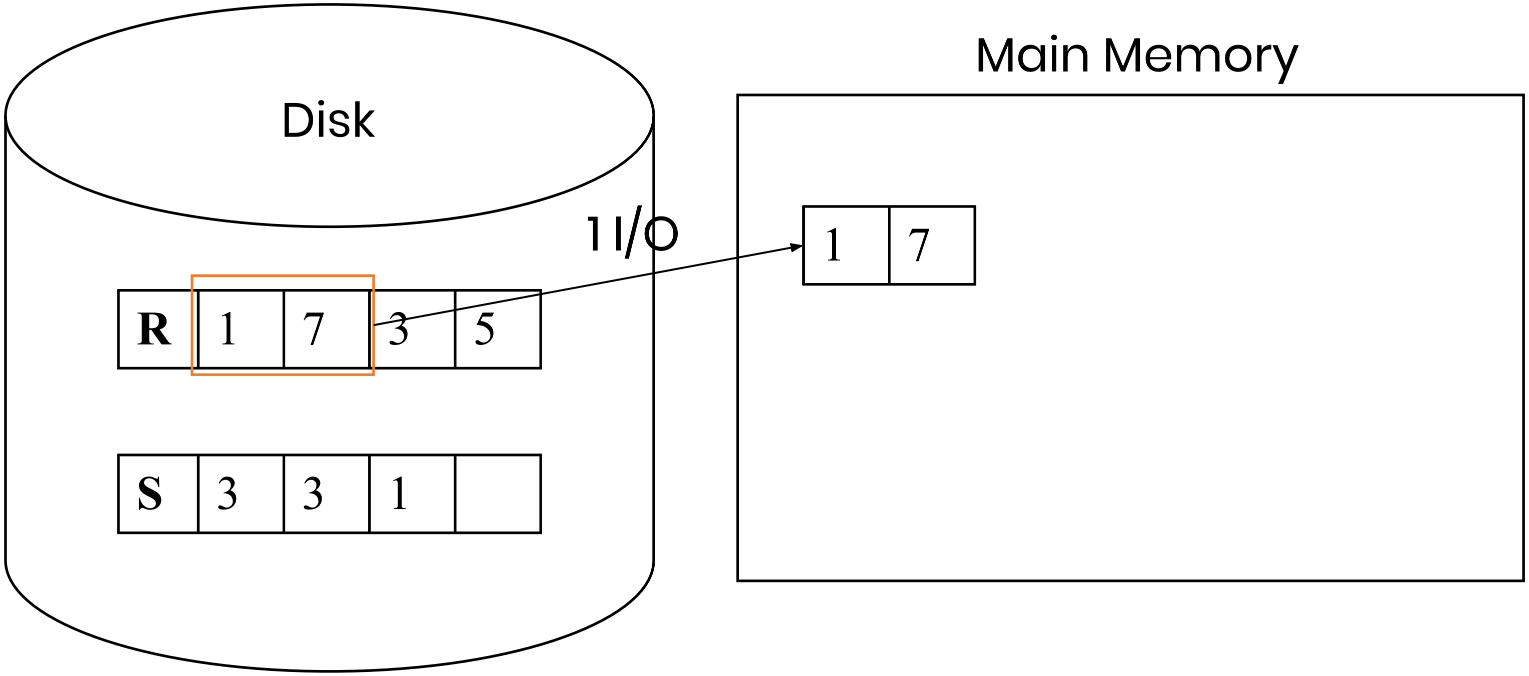
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



(block-at-a-time nested loop join)

x □ A tuple where x is the join attribute value

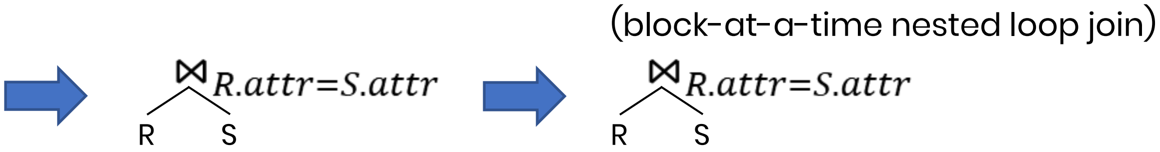


Assume block size = 2 tuples

Nested Loop Join Algorithm

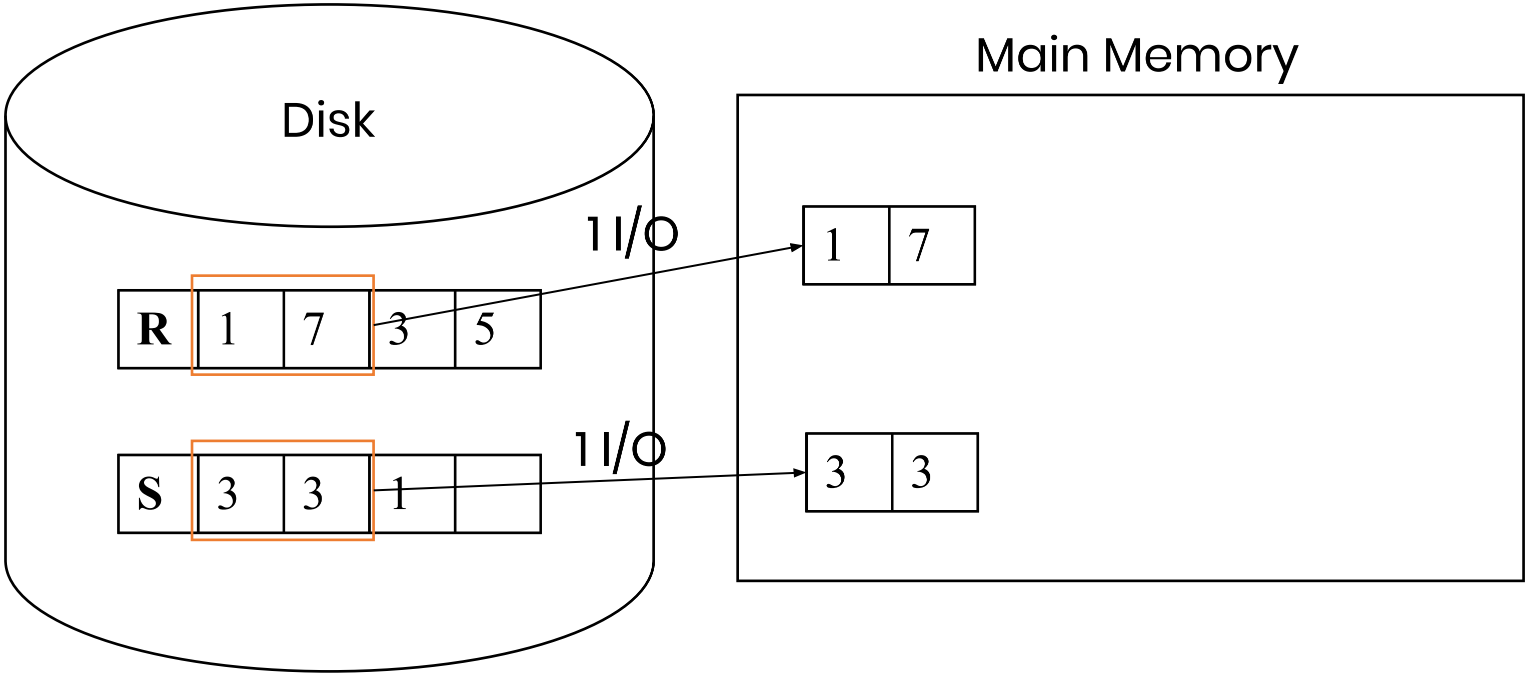
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



$\bowtie_{R.attr=S.attr}$
R
S
(block-at-a-time nested loop join)
$\bowtie_{R.attr=S.attr}$
R
S

x □ A tuple where x is the join attribute value



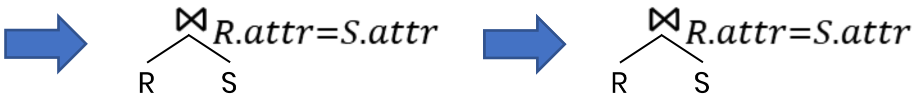
Assume block size = 2 tuples

Nested Loop Join Algorithm

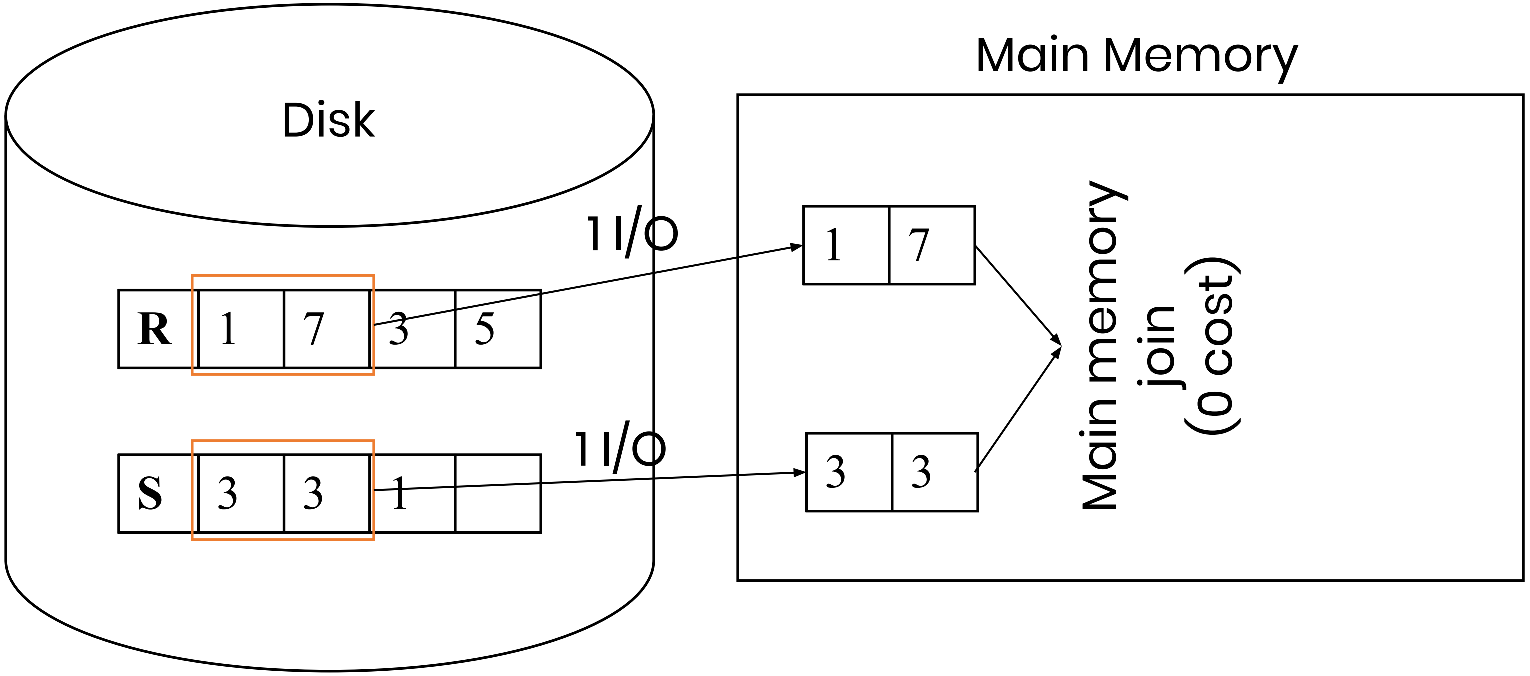
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```

(block-at-a-time nested loop join)



x □ A tuple where x is the join attribute value



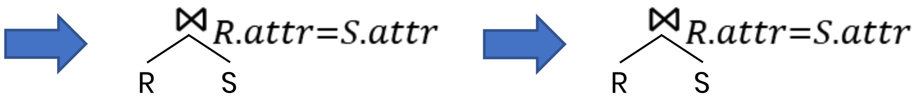
Assume block size = 2 tuples

Nested Loop Join Algorithm

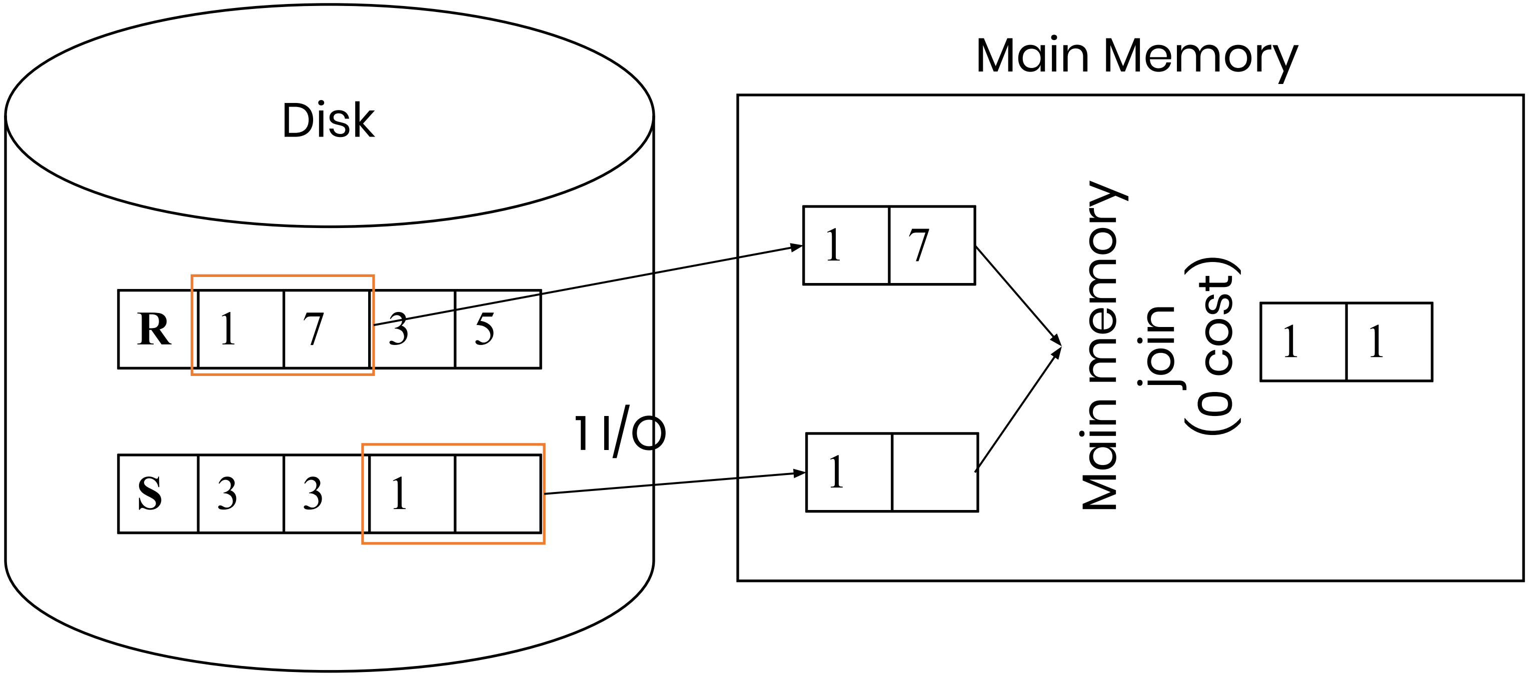
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```

(block-at-a-time nested loop join)



x □ A tuple where x is the join attribute value

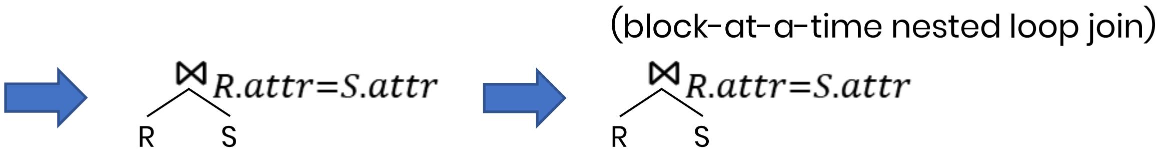


Assume block size = 2 tuples

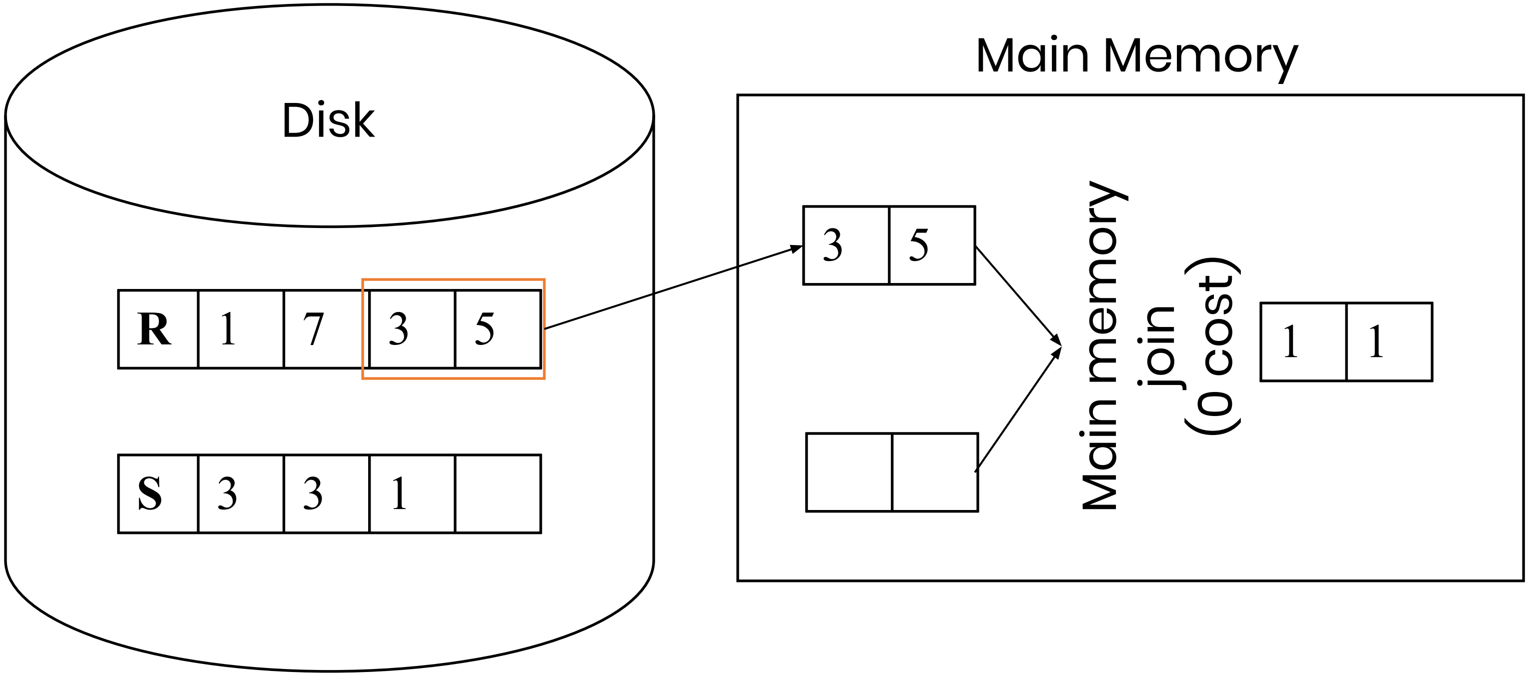
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



x □ A tuple where x is the join attribute value



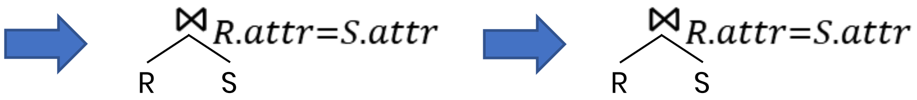
Assume block size = 2 tuples

Nested Loop Join Algorithm

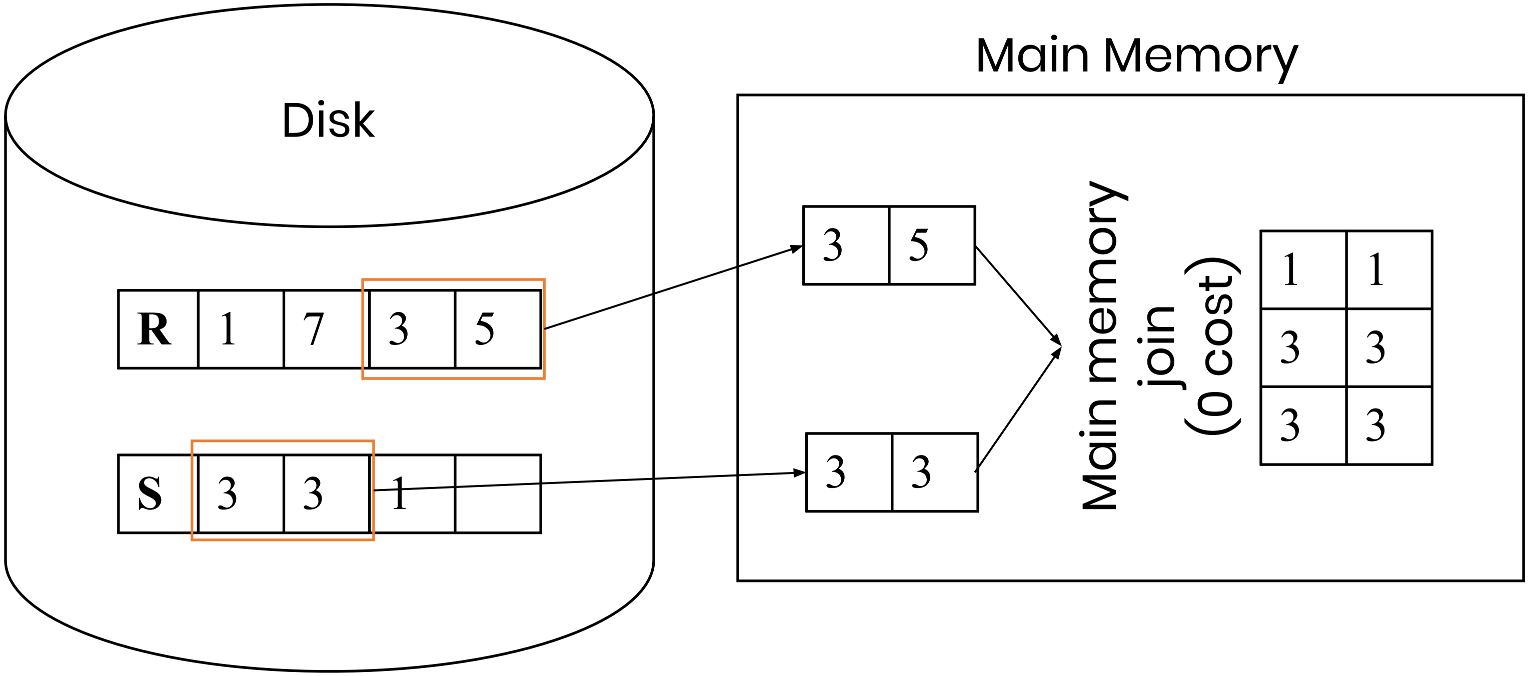
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```

(block-at-a-time nested loop join)



x □ A tuple where x is the join attribute value

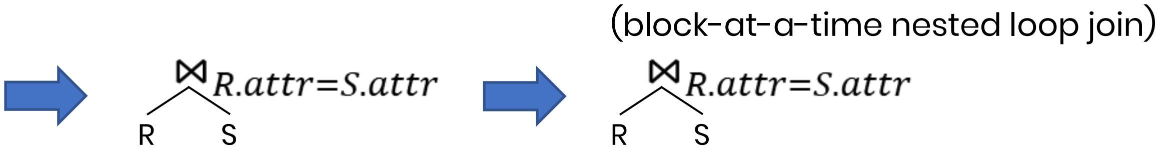


Assume block size = 2 tuples

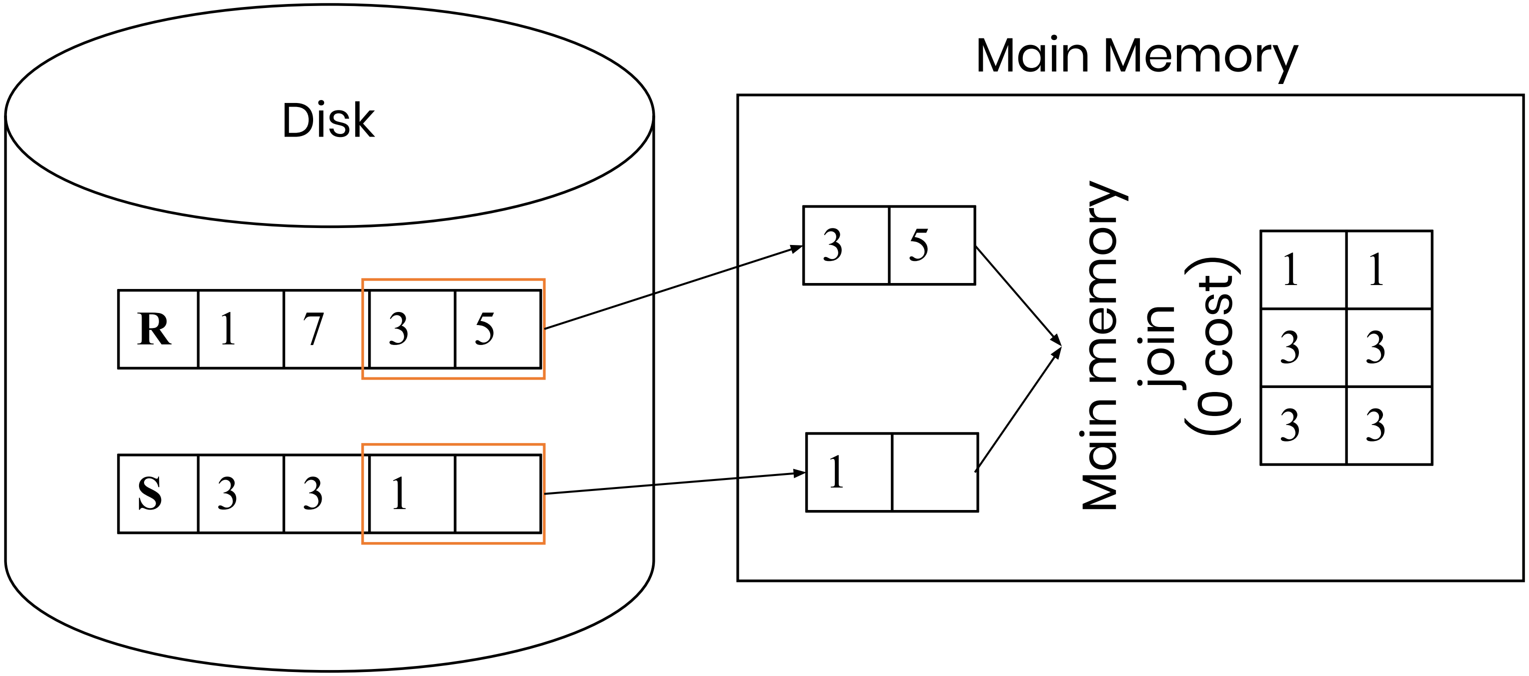
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



x □ A tuple where x is the join attribute value

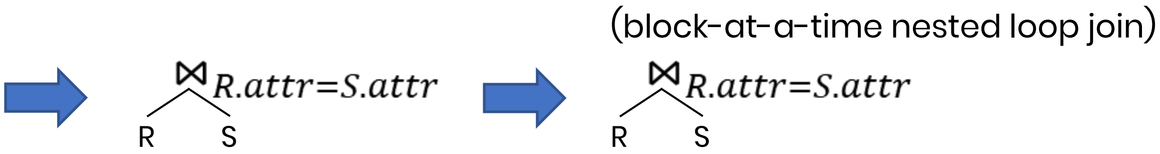


Assume block size = 2 tuples

Nested Loop Join Algorithm

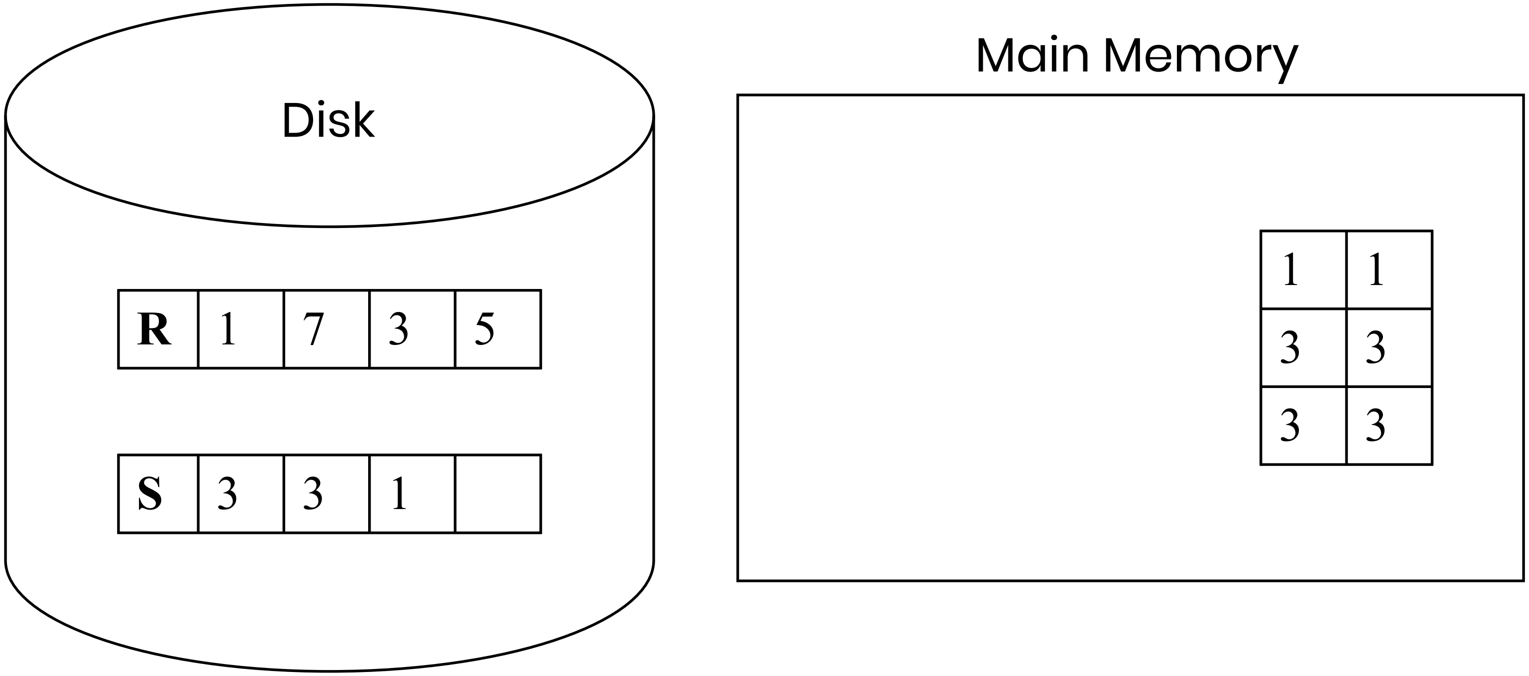
Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



(block-at-a-time nested loop join)
$\bowtie_{R.attr=S.attr}$
R
S
$\bowtie_{R.attr=S.attr}$
R
S

x □ A tuple where x is the join attribute value



R	1	7	3	5
S	3	3	1	

1	1
3	3
3	3

Assume block size = 2 tuples

Nested Loop Join Algorithm

Block-at-a-time nested loop join

Cost = B(R)+B(R)*B(S)

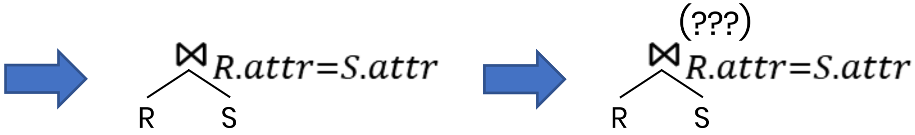
Reading all of R…

… for each block of R read all of S

Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```

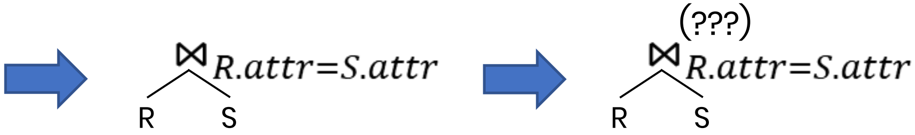


Can I do it faster?

Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



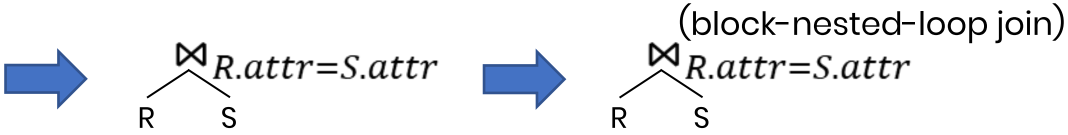
Can I do it faster?
Yeah… if you’re willing to use more memory

Algorithms 101:
Time complexity vs space complexity tradeoff

Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



$\bowtie_{R.attr=S.attr}$

R S

(block-nested-loop join)

$\bowtie_{R.attr=S.attr}$

R S

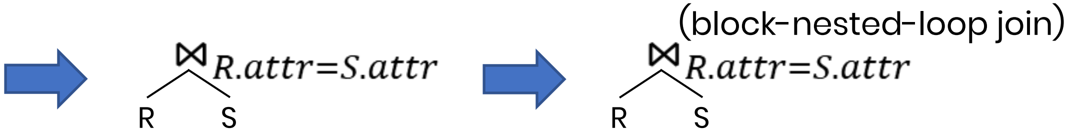
Optimized block-nested-loop join:

```
for each group of N blocks bR in R:
    for each block bS in S:
        for each tuple tR in bR:
            for each tuple tS in bS:
                if tR and tS can join:
                    output (tR,tS)
```

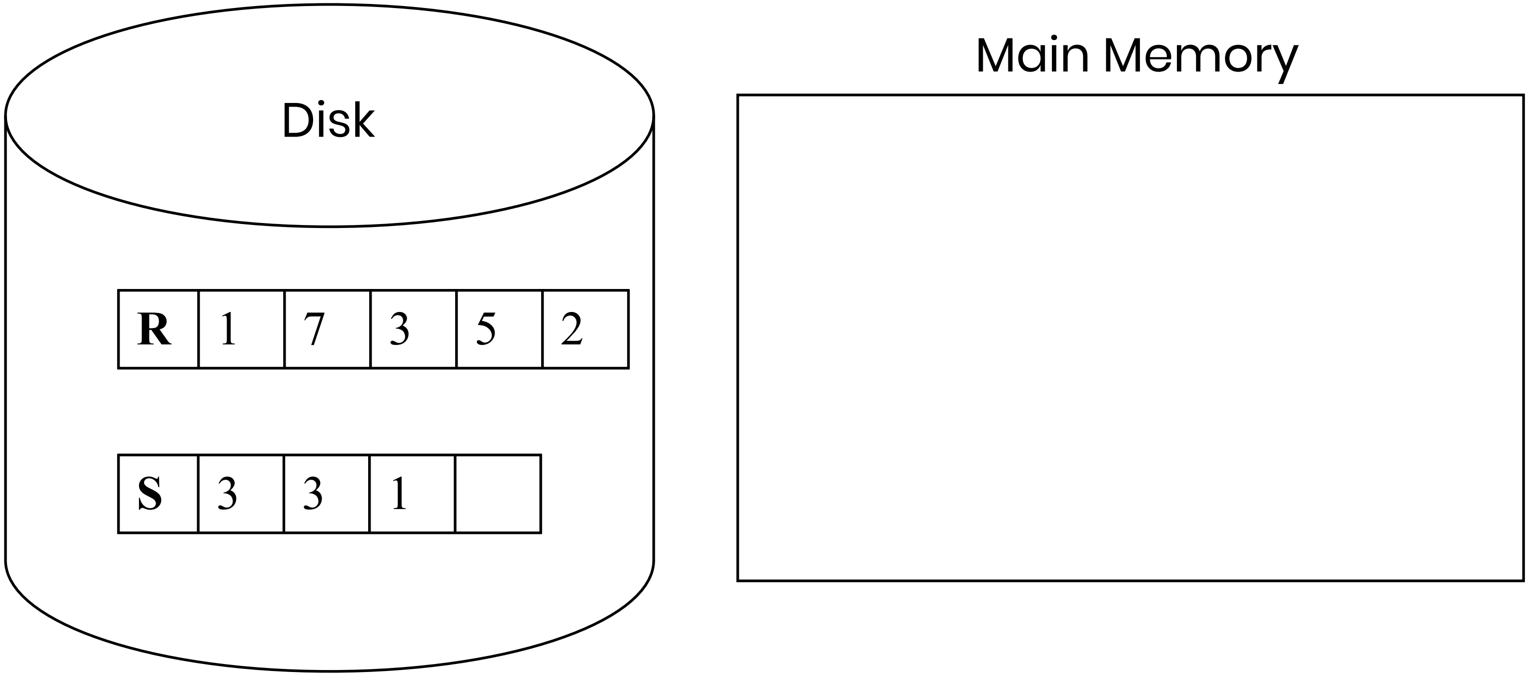
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



N = 2 blocks



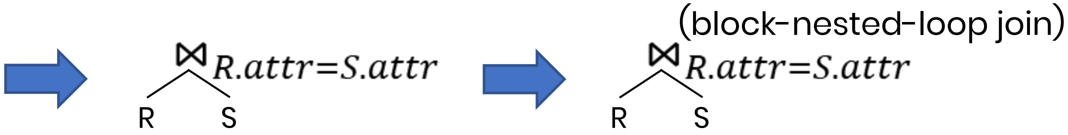
R	1	7	3	5	2
S	3	3	1		

Assume block size = 2 tuples

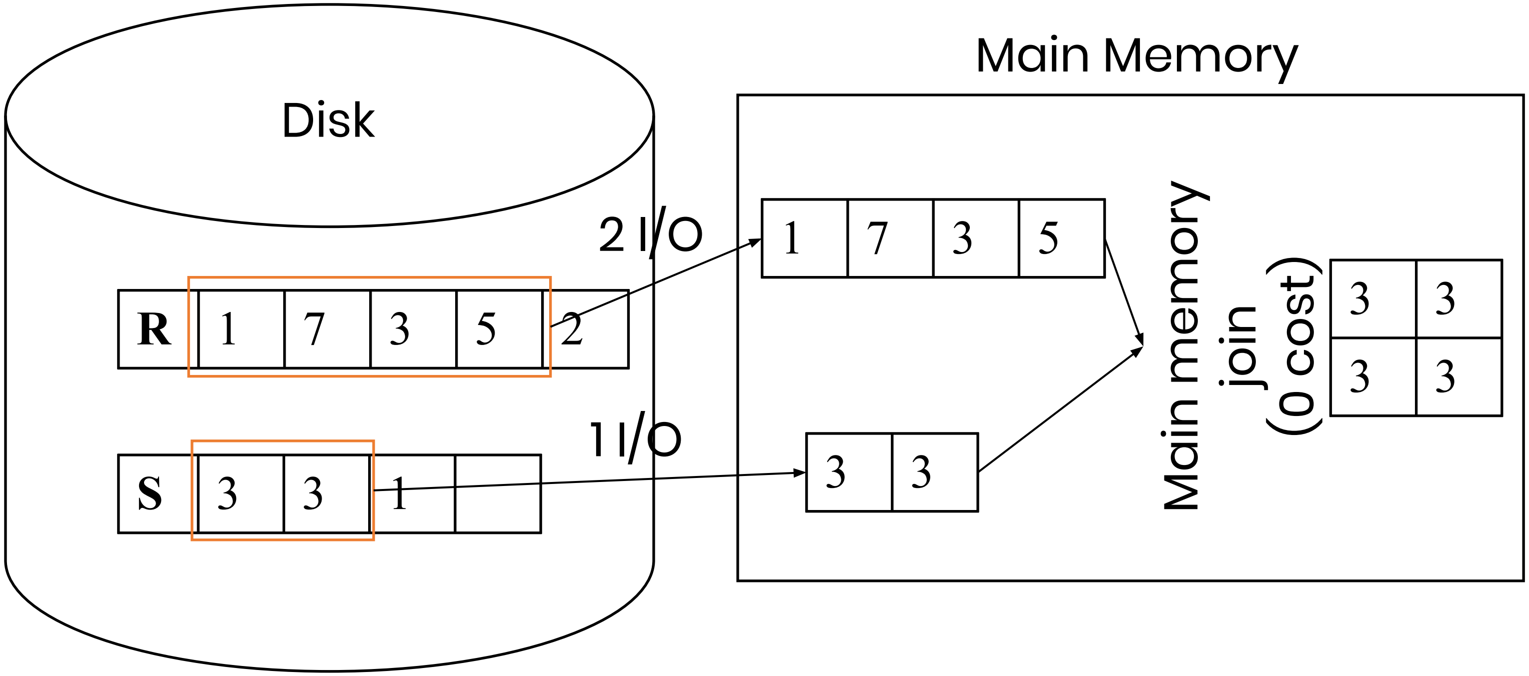
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



N = 2 blocks

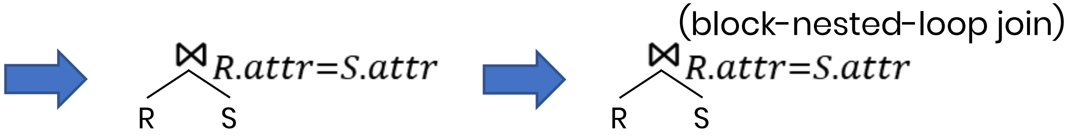


Assume block size = 2 tuples

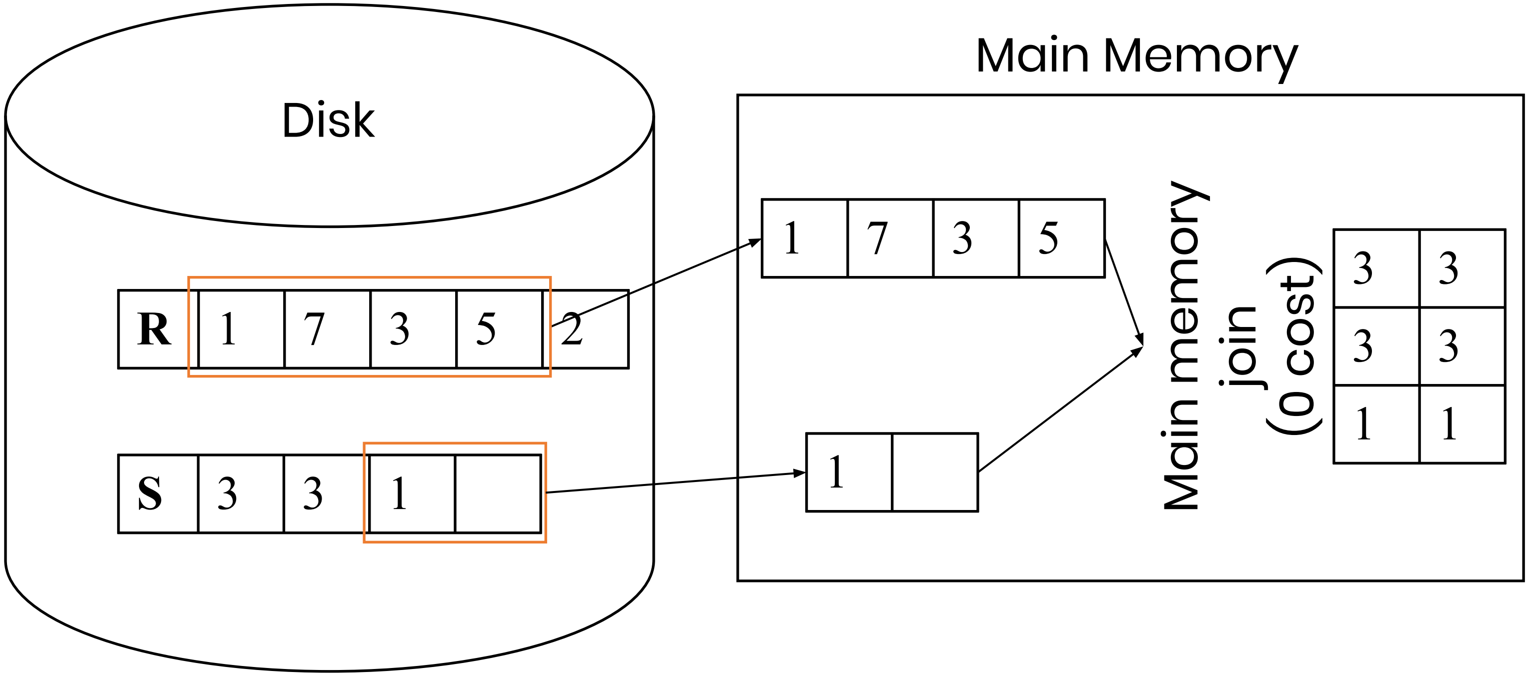
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



N = 2 blocks



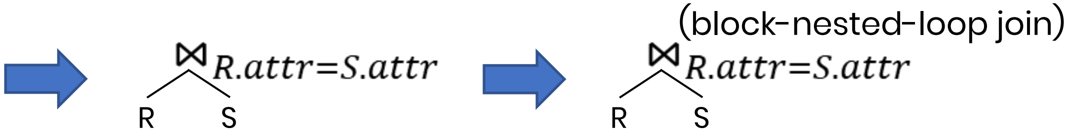
3	3
3	3
1	1

Assume block size = 2 tuples

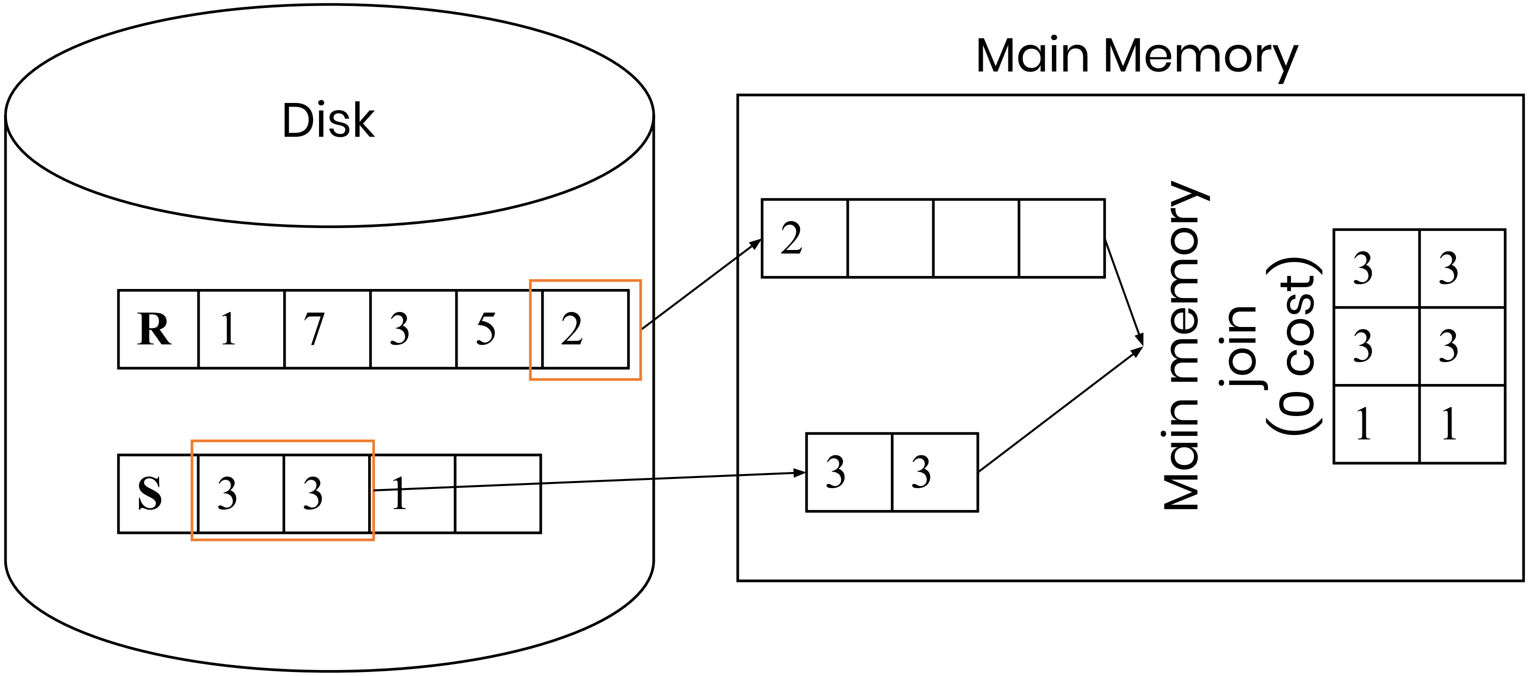
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



N = 2 blocks

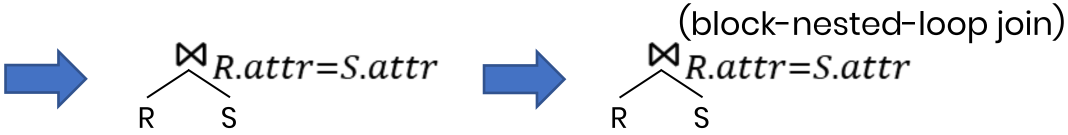


Assume block size = 2 tuples

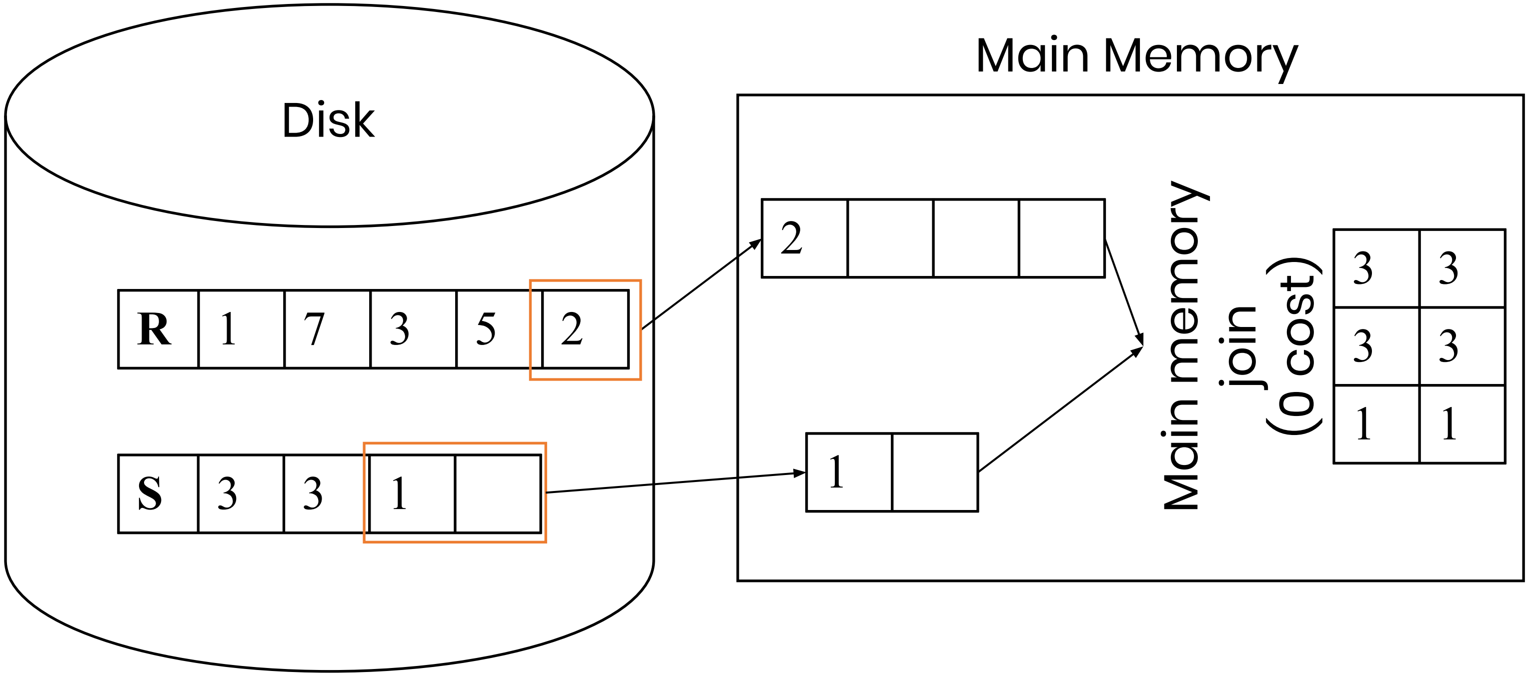
Nested Loop Join Algorithm

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



N = 2 blocks



Assume block size = 2 tuples

Nested Loop Join Algorithm

Block-nested-loop join

$$\textbf{Cost} = B(R)+B(R)/N*B(S)$$

Reading all of R…

… for each group of N blocks of R read all of S

Join Algorithm Summary

- Nested-Loop Join
  - Versatile
- **Hash Join** (single pass)
  - Fast
  - Needs at least one input to be small
- Sort-Merge Join (single pass)
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small

Hash Tables 101

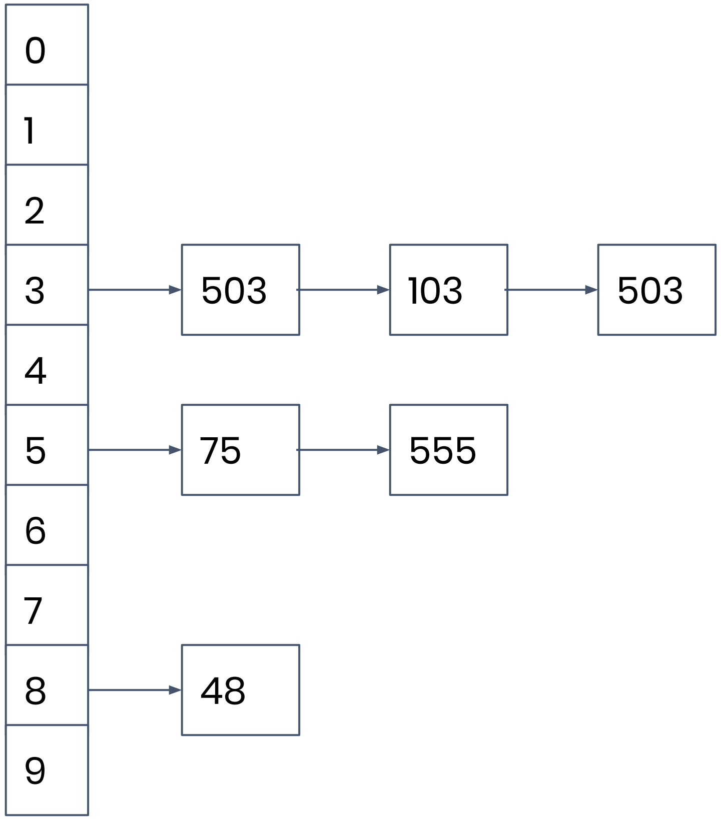
A naive hash function:

h(x) = x mod 10

Operations:

find(103) = ??
insert(488) = ??

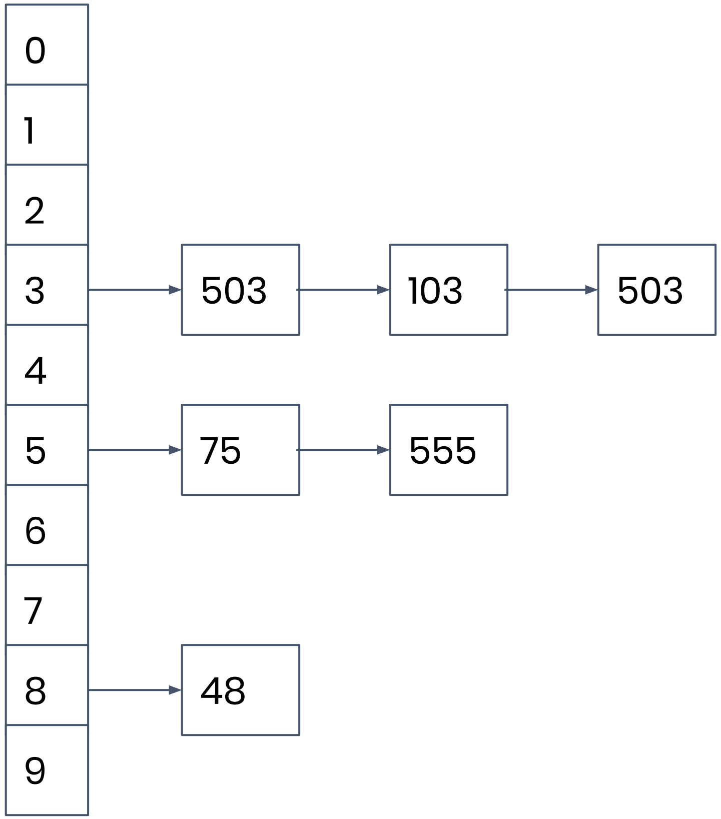
Separate chaining:



Hash Tables 101

- insert(k, v) inserts key k with value v
- Many values for one key
    - Duplicates are ok for our bag semantics
- find(k) returns a *list* of all values associated with the key

Separate chaining:



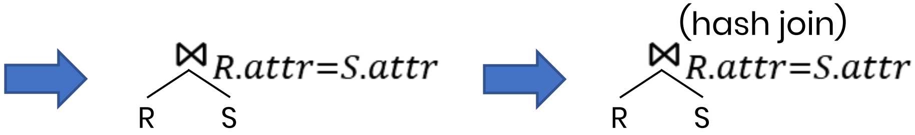
Hash Join

- Make a lookup/hash table from the smaller table
  - Smaller table has to be smaller than total main memory available (**$B(R) < M$ or $B(S) < M$**)
- For each block of the larger table, join using the lookup/hash table

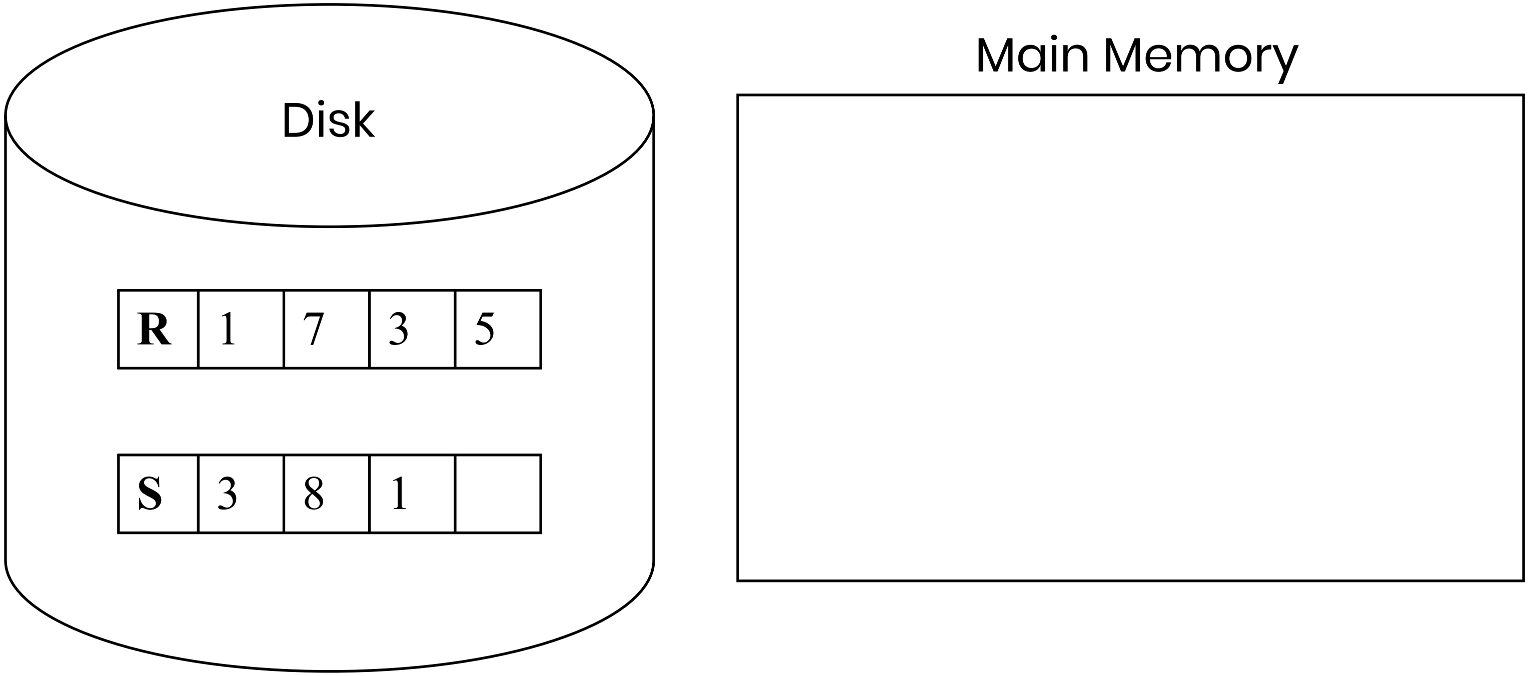
Hash Join

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



M = 10 blocks, hash(x) = x mod 5



R	1	7	3	5

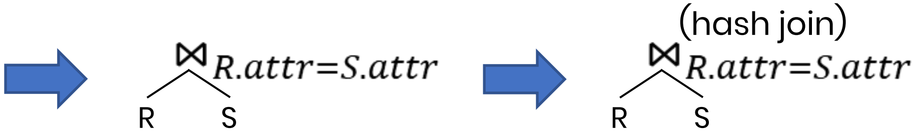
S	3	8	1	

Assume block size = 2 tuples

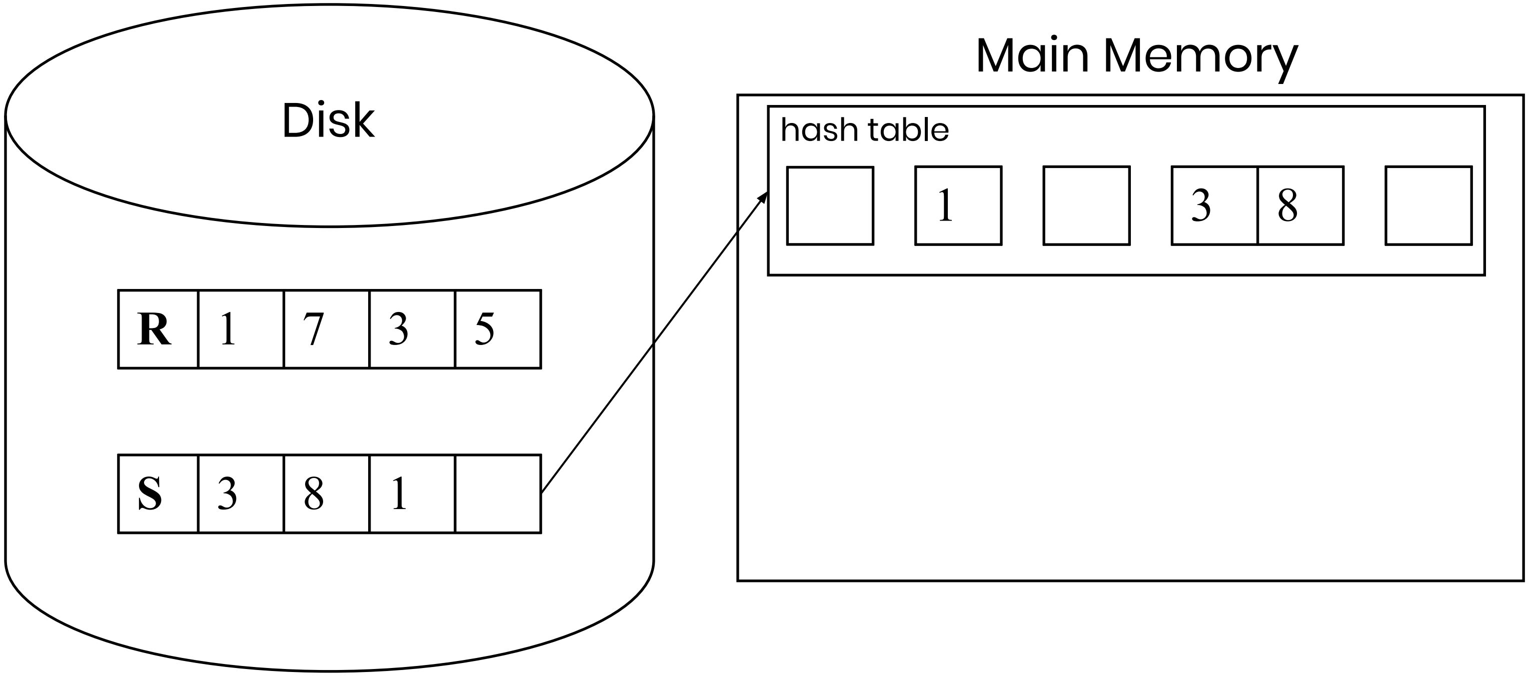
Hash Join

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



M = 10 blocks, hash(x) = x mod 5

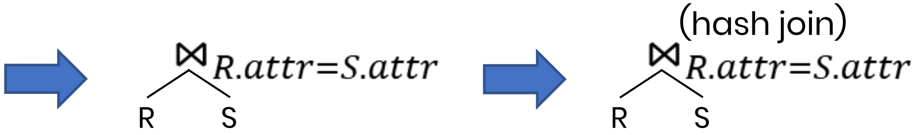


Assume block size = 2 tuples

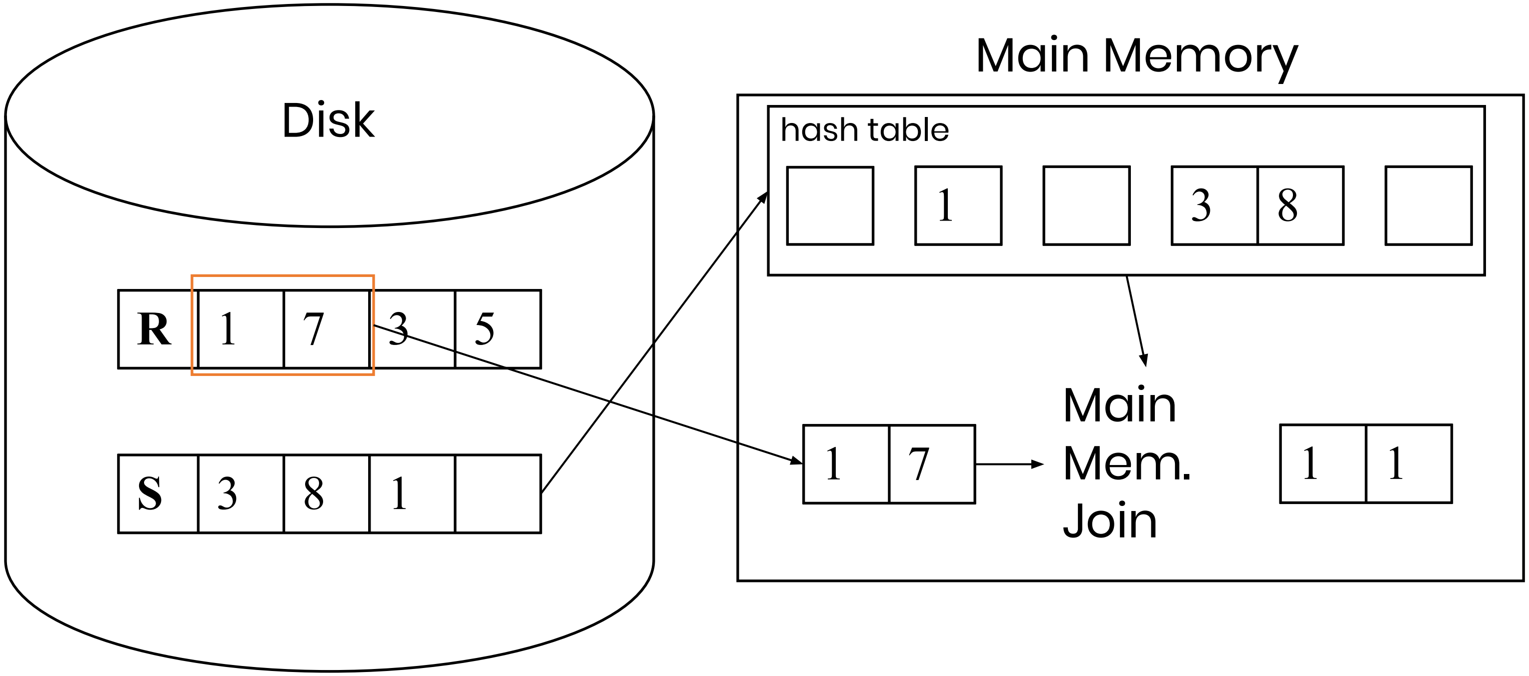
Hash Join

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



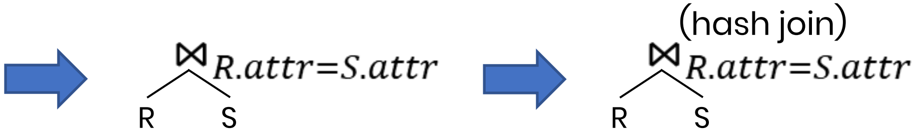
M = 10 blocks, hash(x) = x mod 5



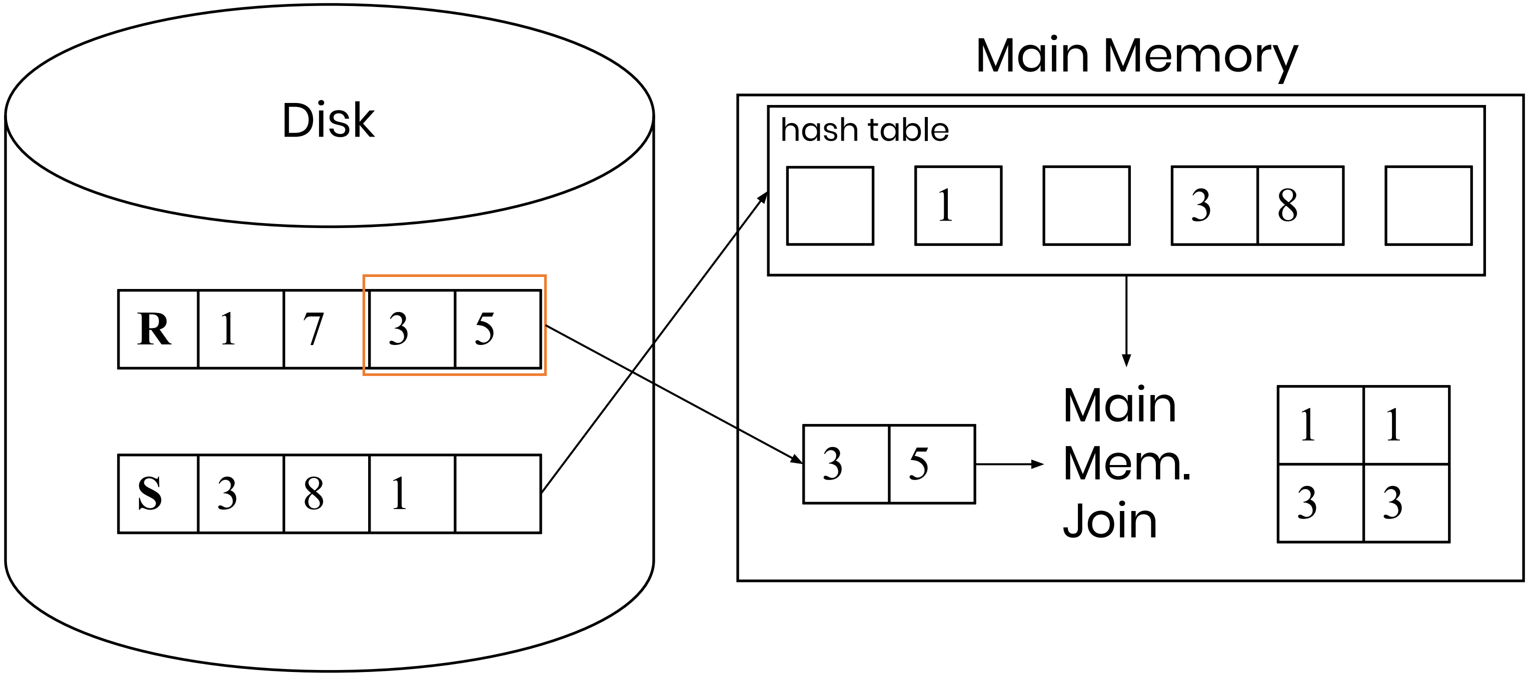
Assume block size = 2 tuples

Hash Join

Example equijoin
equijoin



M = 10 blocks, hash(x) = x mod 5



Assume block size = 2 tuples

Hash Join

Hash join

Cost = B(R)+B(S)

Assuming B(R) < M
Read all of R into a hash table…

…and join with all of S

Hash Join

Hash join

Cost = B(R)+B(S)

Isn’t this the same as block-nested-loop join where B(R)=N?

Cost = B(R)+B(R)/N*B(S)

Hash Join

Hash join

Cost = B(R)+B(S)

Isn’t this the same as block-nested-loop join where B(R)=N?

Cost = B(R)+B(R)/N*B(S)

Yes! It’s the optimal “one-pass” join!

Join Algorithm Summary

- Nested-Loop Join
  - Versatile
- Hash Join (single pass)
  - Fast
  - Needs at least one input to be small
- **Sort-Merge Join** (single pass)
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small

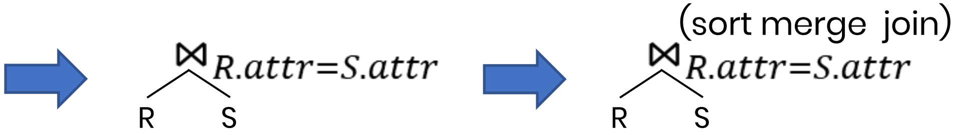
Sort-Merge Join

- Sort both tables into lists in memory
  - Since the sorted lists must contain all tuples, both tables together must fit in memory $(\mathbf{B(R)+B(S) < M})$
- Merge the lists in memory to join
  - Preserves order!

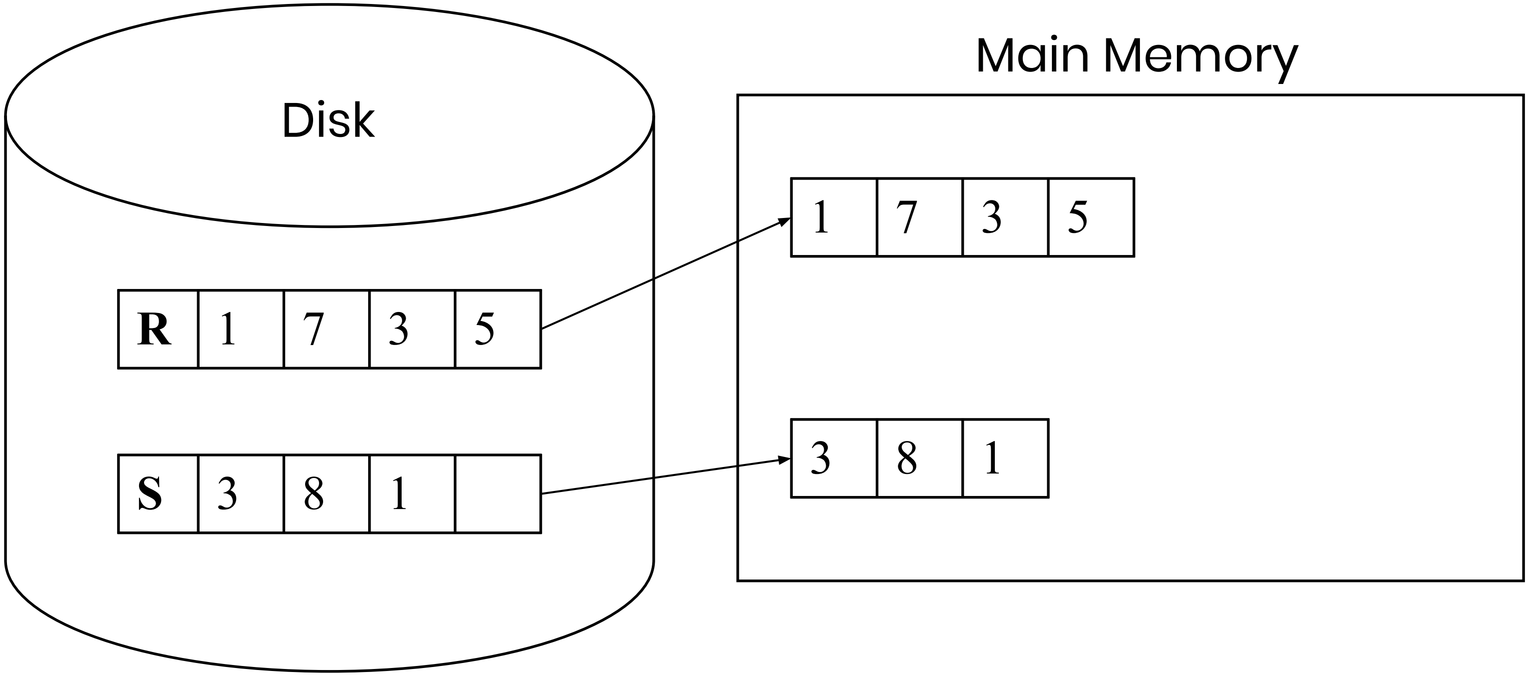
Sort-Merge Join

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



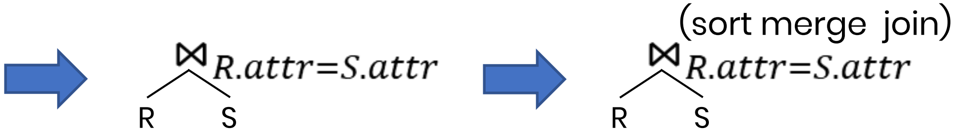
M = 10 blocks



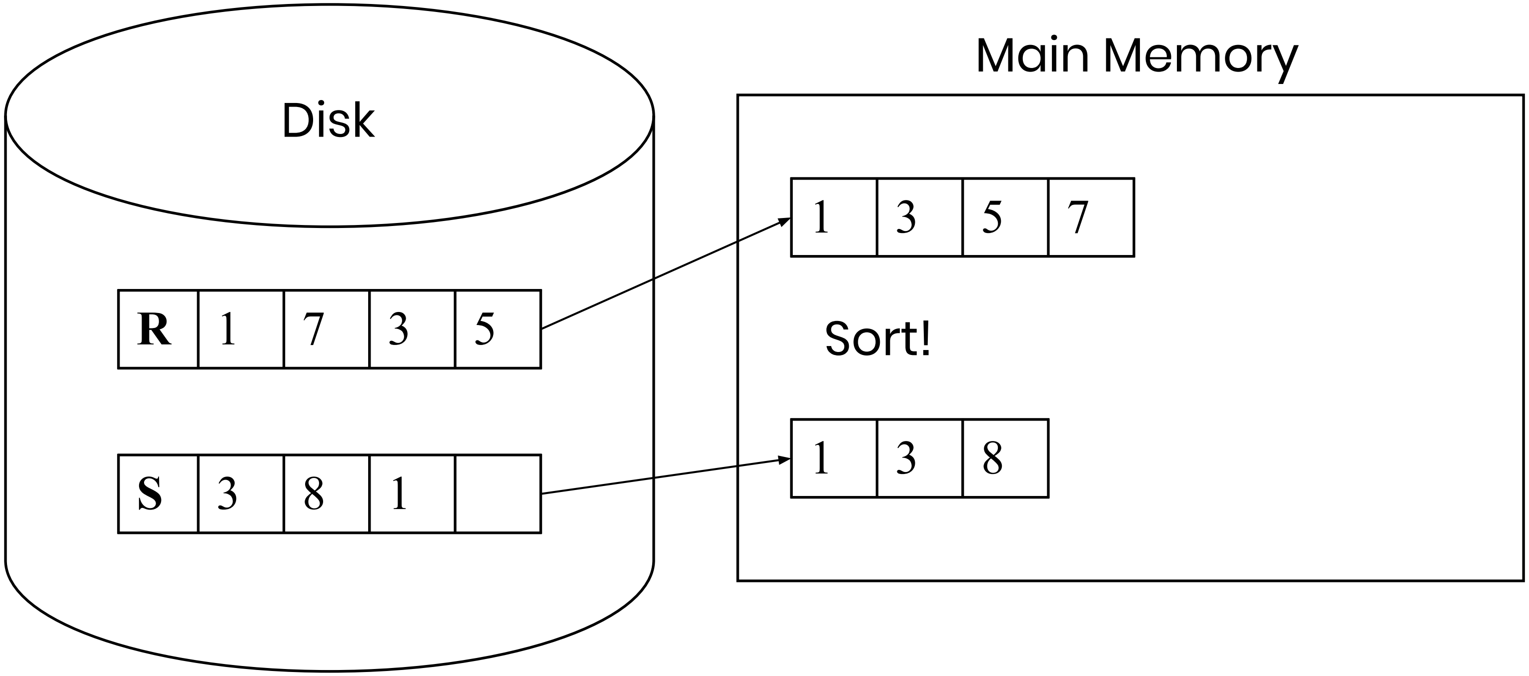
Sort-Merge Join

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



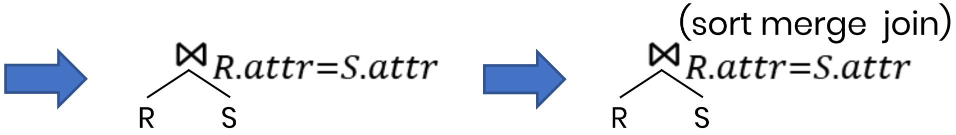
M = 10 blocks



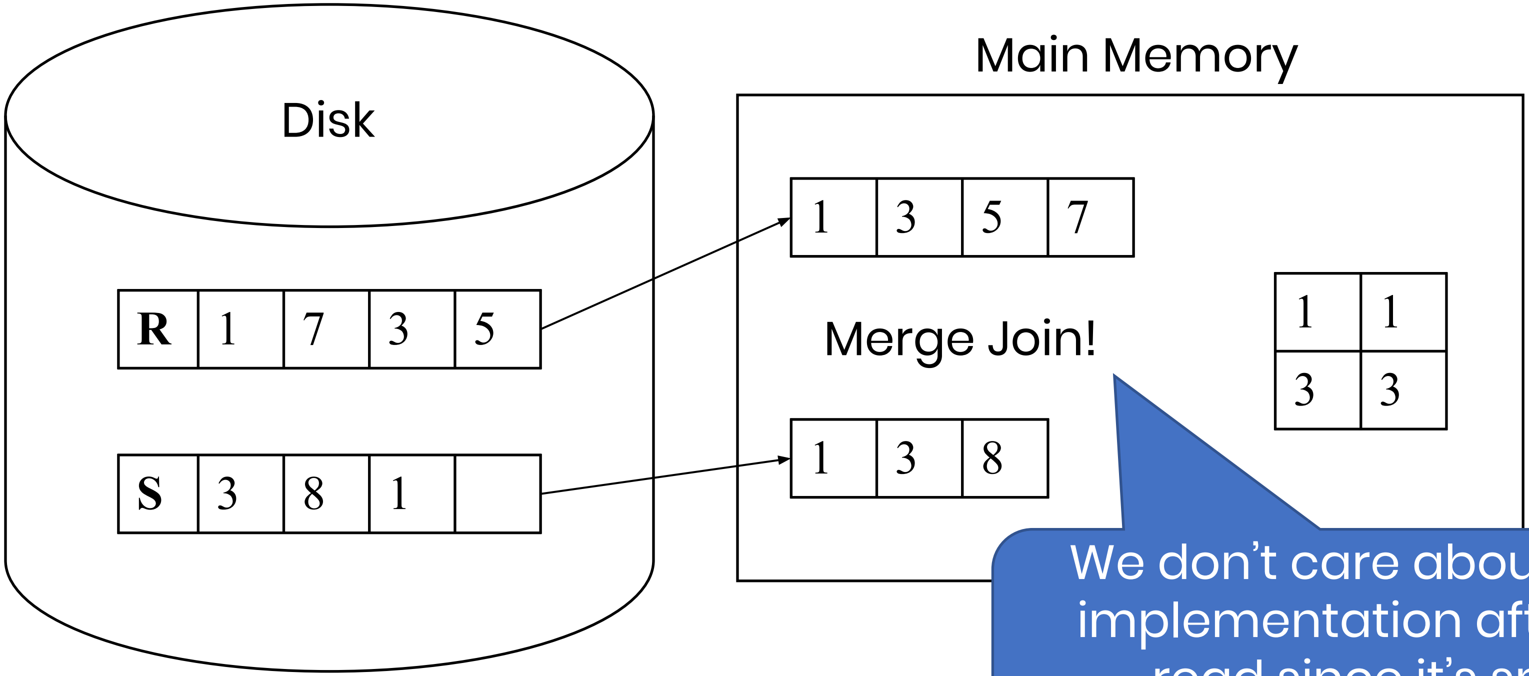
Sort-Merge Join

Example equijoin

```
SELECT *
  FROM R, S
 WHERE R.attr = S.attr
```



M = 10 blocks

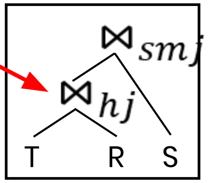


We don't care about exact implementation after disk read since it's small compared to IO

Cardinality Estimation

- Another building block when estimating the overall cost of a plan
- If we have an RA tree, we need to estimate the output cardinality of the "lower" operations since it's the input to "upper" operations

how many tuples here??



Least Cost Plan

Cardinality Estimation

- Estimate the number of tuples in the output of each RA operator
  - err, without actually computing the output

- Let’s go grocery shop!
  - Safeway(id, name, category, price)
  - QFC(id, name, category, price)

- Let’s use store stats to estimate the cardinality of some queries

Cardinality Estimation

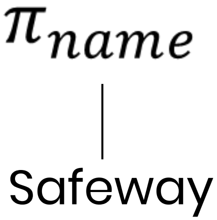
Underline = primary key

- Safeway(id, name, category, price)
  - T = 1000 *# of tuples*
  - V(name) = 900 *# of distinct values*
  - V(category) = 10
  - V(price) = 200
  - Range(price) = [1,50) *range of values*
- QFC(id, name, category, price)
  - T = 2000
  - V(name) = 1900
  - V(category) = 12
  - V(price) = 500

Cardinality Estimation: SELECT

Safeway(id, name, category, price) T = 1000

```
SELECT name
  FROM Safeway
```



π_{name}

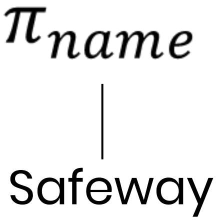
Safeway

How many tuples do we expect this query to output?

Cardinality Estimation: SELECT

Safeway(<u>id</u>, name, category, price)   T = 1000

```
SELECT name
  FROM Safeway
```



How many tuples do we expect this query to output?

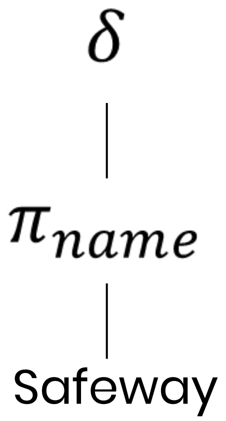
ANSWER: 1000   (no change)

Cardinality Estimation: DISTINCT

Safeway(id, name, category, price)   T = 1000

V(name) = 900

```
SELECT DISTINCT name
   FROM Safeway
```



δ

π_{name}

Safeway

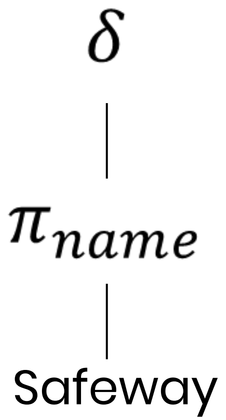
How many tuples do we expect this query to output?

Cardinality Estimation: DISTINCT

Safeway(id, name, category, price) T = 1000

V(name) = 900

```
SELECT DISTINCT name
  FROM Safeway
```



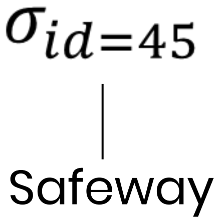
How many tuples do we expect this query to output?

ANSWER: 900 (set to distinct values)

Cardinality Estimation: WHERE Value

Safeway(id, name, category, price)   T = 1000

```
SELECT *
  FROM Safeway
 WHERE id = 45
```



$\sigma_{id=45}$

Safeway

How many tuples do we expect this query to output?

ASSUME: that ‘45’ exists in the distinct values of id

Answer is 0 otherwise…

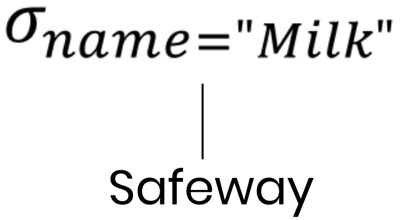
ANSWER: 1

Cardinality Estimation: WHERE Value

Safeway(id, name, category, price)   T = 1000

V(name) = 900

```
SELECT *
FROM Safeway
WHERE name = 'Milk'
```



$\sigma_{name="Milk"}$

Safeway

ASSUME: distinct values uniformly distributed

Without assumptions, estimation is impossible…

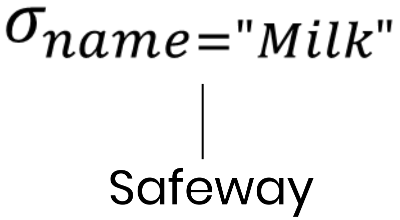
ANSWER: 1000 / 900 ≈ 1.11 tuples

Cardinality Estimation: WHERE Value

Safeway(id, name, category, price) T = 1000

V(name) = 900

```
SELECT *
FROM Safeway
WHERE name = 'Milk'
```



$\sigma_{name="Milk"}$

Safeway

Select Value: $\frac{T(op)}{V(op, attr)}$

ASSUME: distinct values uniformly distributed

Without assumptions, estimation is impossible...

ANSWER: 1000 / $\boxed{900 \approx}$ 1.11 tuples

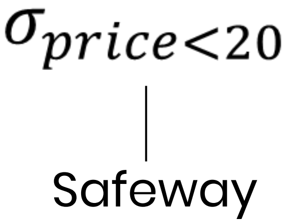
The **selectivity factor**

Cardinality Estimation: WHERE Range

Safeway(id, name, category, price) T = 1000

V(price) = 200 Range(price) = [1,50)

```
SELECT *
FROM Safeway
WHERE price < 20
```



ASSUME: distinct values uniformly distributed & continuous

Without assumptions, estimation is impossible…

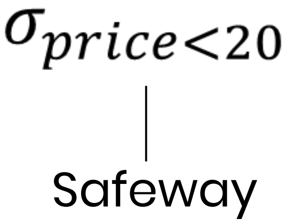
ANSWER: 1000 * (20 – 1) / (50 – 1) ≈ 387.8 tuples

Cardinality Estimation: WHERE Range

Safeway(id, name, category, price) T = 1000

V(price) = 200 Range(price) = [1,50)

```
SELECT *
FROM Safeway
WHERE price < 20
```



Select Range: $T(op) * \frac{(Val-Min)}{(Max-Min)}$

ASSUME: distinct values uniformly distributed & continuous

Without assumptions, estimation is impossible…

ANSWER: 1000 * $\boxed{(20 - 1) / (50 - 1)}$ ≈ 387.8 tuples

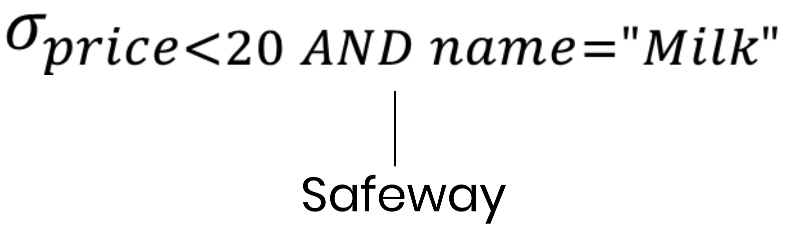
The **selectivity factor**

Cardinality Estimation: WHERE and

Safeway(<u>id</u>, name, category, price)   T = 1000

V(name) = 900   V(price) = 200   Range(price) = [1,50)

```
SELECT *
FROM Safeway
WHERE price < 20
   AND name = 'Milk'
```



$\sigma_{price<20\ AND\ name="Milk"}$

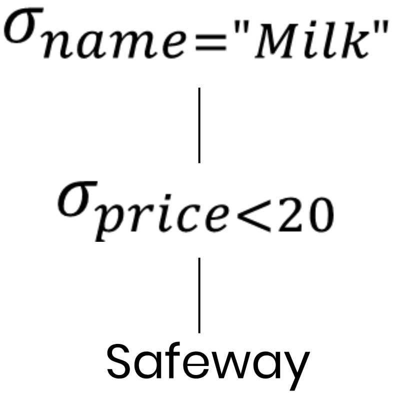
Safeway

Cardinality Estimation: WHERE and

Safeway(id, name, category, price)   T = 1000

V(name) = 900   V(price) = 200   Range(price) = [1,50)

```
SELECT *
FROM Safeway
WHERE price < 20
  AND name = 'Milk'
```



Hard to say

e.g. no milk costs < 20

e.g. milk & price independent

e.g. all milk costs < 20

If conditions disjoint, **0** tuples result

If conditions independent, **multiply** estimates

If conditions fully overlap, take **minimum** of estimates

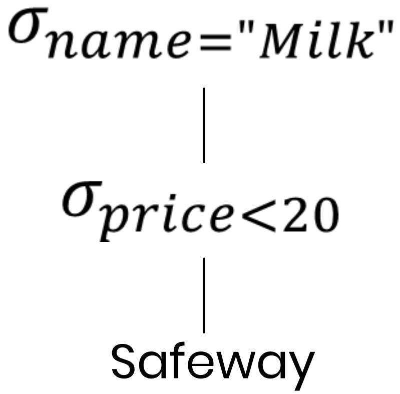
ASSUME independent unless you know for sure

Cardinality Estimation: WHERE and

Safeway(<u>id</u>, name, category, price) T = 1000

V(name) = 900 V(price) = 200 Range(price) = [1,50)

```
SELECT *
FROM Safeway
WHERE price < 20
   AND name = 'Milk'
```



ANSWER:

assuming independence

$\approx 1000 * [(20 - 1) / (50 - 1)] * 1/900 \approx$ 0.431 tuples

Cardinality Estimation: WHERE and

Safeway(id, name, category, price)   T = 1000

V(name) = 900   V(price) = 200   Range(price) = [1,50)

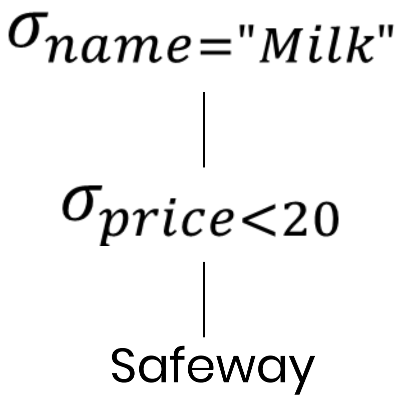
AND / INTERSECT

Assume independence: $T(op) * cond1 * cond2$

unless full overlap: $T(op) * \min\{cond1, cond2\}$

unless disjoint: 0

```
WHERE price < 20
   AND name = 'Milk'
```



ANSWER:

assuming independence

$\approx 1000 * \boxed{[(20 - 1) / (50 - 1)] * 1/900} \approx$ 0.431 tuples

The **selectivity factor**

Cardinality Estimation: JOIN

- Read 16.4.4 in the book for cardinality estimation of JOINs
- We'll use this later!

Takeaways

- Nested-Loop Joins
  - Block-at-a-time □ B(R)+B(R)*B(S)
  - Nested-block-loop □ B(R)+B(R)/N*B(S)
- Hash Join and Sort-Merge Join □ B(R)+B(S)
- Cardinality estimation helps give us inputs for more complex RA trees.