Join and Aggregation Algorithms

CSE 4/562: Database Systems | Lecture 4

DB. Sys.: T.C.B.: 15.2-15.5

Quiz!

Recap

Summary

	Filter	**Merge**	**Derive**
Column	$\pi_{A,B,\ldots}(R)$ “Project”	$R \times S$ “Cartesian Product”	$\pi_{C=A+B}(R)$ “Project”
Row	$\sigma_{A<3}(R)$ “Select” or “Filter”	$R \cup S$ “Union”	$\Sigma_{A,B=\text{ COUNT()}}(R)$ “Aggregate”

Query Evaluation: Example

Passenger	**Name**	**Train_no**	**Dest**
	Athena	48	Buffalo
	Bragi	239	Albany
	Cerberus	241	Hudson

Train	**Train_no**	**Route**
	48	Lake Shore Ltd.
	239	Emp. Service
	241	Emp. Service

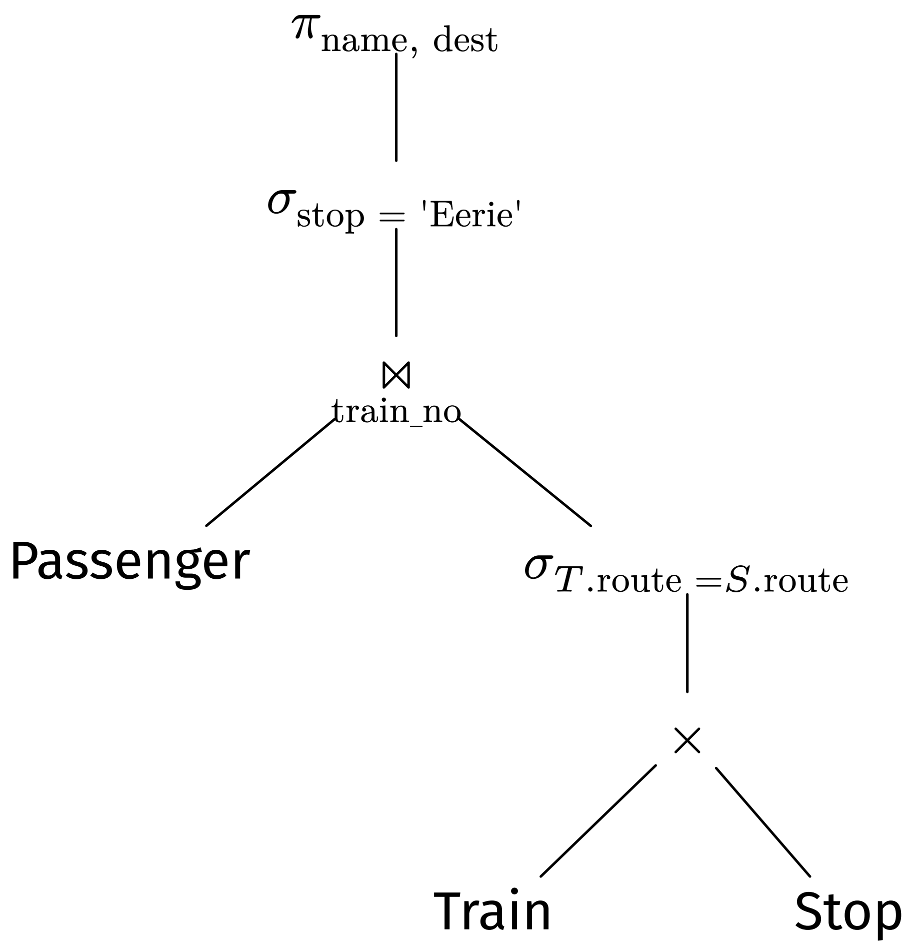
Stop	**Route**	**Stop**
	Emp. Service	NYC
	Emp. Service	Yonkers
	Emp. Service	Hudson
	Emp. Service	Albany
	Lake Shore Ltd.	Chicago
	Lake Shore Ltd.	Erie
	Lake Shore Ltd.	Buffalo

“Find me every passenger on a train that stops in Erie, and their destination”

Query Evaluation: Example

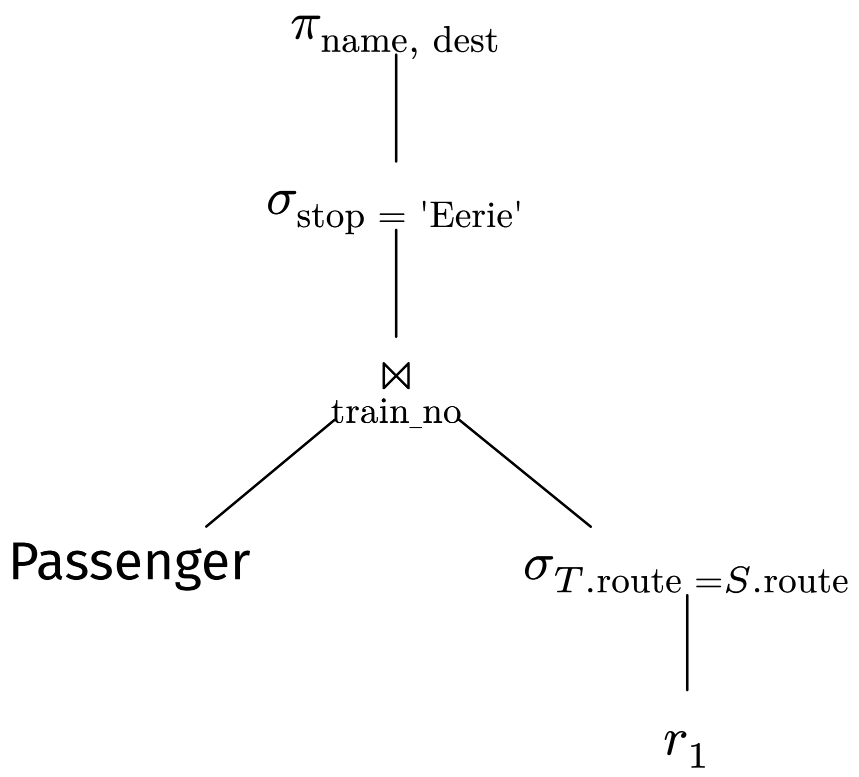
```sql
SELECT name, dest
FROM Passenger P, Train T, Stop S
WHERE stop = 'Erie'
  AND P.train_no = T.train_no
  AND T.route = S.route
```

Query Evaluation: Example

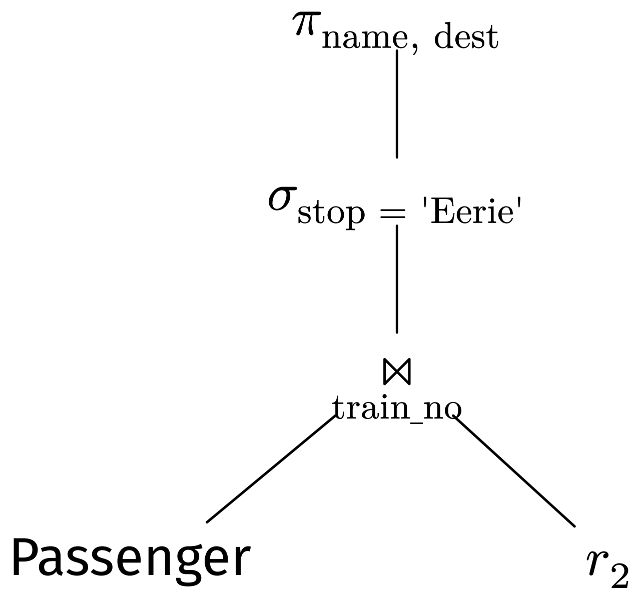


$\pi_{\text{name, dest}}$
$\sigma_{\text{stop = 'Eerie'}}$
$\bowtie_{\text{train\_no}}$
Passenger
$\sigma_{T.\text{route} = S.\text{route}}$
$\times$
Train
Stop

Query Evaluation: Example


$\pi_{\text{name, dest}}$
$\sigma_{\text{stop} = \text{'Eerie'}}$
$\bowtie_{\text{train\_no}}$
Passenger
$\sigma_{T.\text{route} = S.\text{route}}$
r_1

Query Evaluation: Example


$\pi_{\text{name, dest}}$
$\sigma_{\text{stop = 'Eerie'}}$
$\bowtie_{\text{train\_no}}$
Passenger
r_2

Query Evaluation: Example

$$\begin{tikzcd}
\pi_{\text{name, dest}} \arrow[d, dash] \\
\sigma_{\text{stop} = \text{'Eerie'}} \arrow[d, dash] \\
r_3
\end{tikzcd}$$

Query Evaluation: Example

$$\begin{tikzcd}
\pi_{\text{name, dest}} \arrow[d, dash] \\
r_4
\end{tikzcd}$$

IO and Algorithms

```
def filter(condition, In):
    output = []

    for r in In:
        if condition(r):
            output += [ r ]

    return output
```

Big!

The External Memory Model

The model

- I: A limited “Internal” memory that starts off empty.
  - $|I|$ is the Memory Complexity
- E: An infinite “External” memory.

Actions

- $I[42] \leftarrow \mathbf{Compute}(I[0], I[1], ...)$: Adds to Runtime Complexity
- $I[12] \leftarrow \mathbf{Read}(E[974])$: Adds to IO Complexity
  - “Read Cost”: Just reads
- $E[974] \leftarrow \mathbf{Write}(I[12])$: Adds to IO Complexity
  - “Write Cost”: Just writes

The “IO Cost” or “IO Complexity” is Read cost + Write cost

Algorithm Quality

- **Runtime Complexity**: How fast is the algorithm for a given input?
- **Memory Complexity**: How much memory do we need to run the algorithm on a given input?
- **IO Complexity**: How much data gets moved between layers of the hierarchy?

We can measure complexity…
- exactly ($3N$ pages of IO)
- asymptotically ($O(N)$ memory)

Evaluation Strategies

Operator-at-a-Time

Input: Complete relations

Output: Complete relation

Tradeoffs:

- Easy to implement
- Very low spatial locality (relations that don’t fit in memory need to be on disk)

The Pull Model (Volcano, Iterators)

Input: Iterators over relations

Output: Iterator over relation

Tradeoffs:

- Much better spatial locality

Iterators

```
class Iterator:
    def next(self) -> Option[Record]:
        pass
```

Operator iterators compose and the final result is read off the final iterator:

```
def print_result(query_iterator):
    while (row := query_iterator.next()) is not None
        print(row)
```

Evaluation Models

	Operator-at-a-time		Pull Evaluation	
Operator	**Memory**	**IO**	**Memory**	**IO**
Table	n/a	n/a	$O(1)$	$O(n)$
Project (π)	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Filter (σ)	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Union ($\cup$)	$O(1)$	$O(n)$	???	???
Product ($\times$)	$O(1)$ or $O(n)$	$O(n^2)$ or $O(n)$	???	???
Join ($\bowtie$)	$O(1)$ or $O(n)$	$O(n^2)$ or $O(n)$	???	???
Aggregate (Σ)	???	???	???	???

Product and Join Algorithms

Union Iterator

```
class UnionIterator:
    def __new__(self, in1, in2):
        self.in1 = in1
        self.in2 = in2

    def next(self) -> Option[Record]
        ???
```

Union Iterator

```
class UnionIterator:
    def __new__(self, in1, in2):
        self.in1 = in1
        self.in2 = in2

    def next(self) -> Option[Record]
        if (row := self.in1.next()) is not None:
            return row
        else:
            return self.in2.next()
```

Union Iterator

```
class UnionIterator:
    def __new__(self, in1, in2):
        self.in1 = in1
        self.in2 = in2

    def next(self) -> Option[Record]
        if (row := self.in1.next()) is not None:
            return row
        else:
            return self.in2.next()
```

What is the...

- Memory complexity?
- IO Complexity?

Union Iterator

```
def product(In1, In2):
    for r in In1:
        yield r
    for s in In2:
        yield s
```

Product Iterator

```
class ProductIterator:
    def __new__(self, in1, in2):
        self.in1 = in1
        self.in2 = in2

    def next(self) -> Option[Record]
        ???
```

Product Iterator

```
class ProductIterator:
    def __new__(self, in1, in2):
        self.in1 = in1
        self.in2 = in2

        self.in2_temp = [row for row in in2]
        self.i = 0
        self.in1_row = in1.next()

    def next(self) -> Option[Record]
        if self.i >= len(self.in2_temp):
            self.i = 0
            self.in1_row = self.in1.next()
        row = self.in1_row + self.in2_temp[i]
        i += 1
        return row
```

Product Iterator

```
class ProductIterator:
    def __new__(self, in1, in2):
        self.in1 = in1
        self.in2 = in2

        self.in2_temp = write_to_file(in2)
        self.in2 = TableIterator(self.in2_temp)
        self.in1_row = self.in1.next()

    def next(self) -> Option[Record]
        if (in2_row := self.in2.next()) is None:
            self.in2 = TableIterator(self.in2_temp)
            self.in1_row = self.in1.next()
        row = self.in1_row + self.in2.next()
        return row
```

Product Iterator

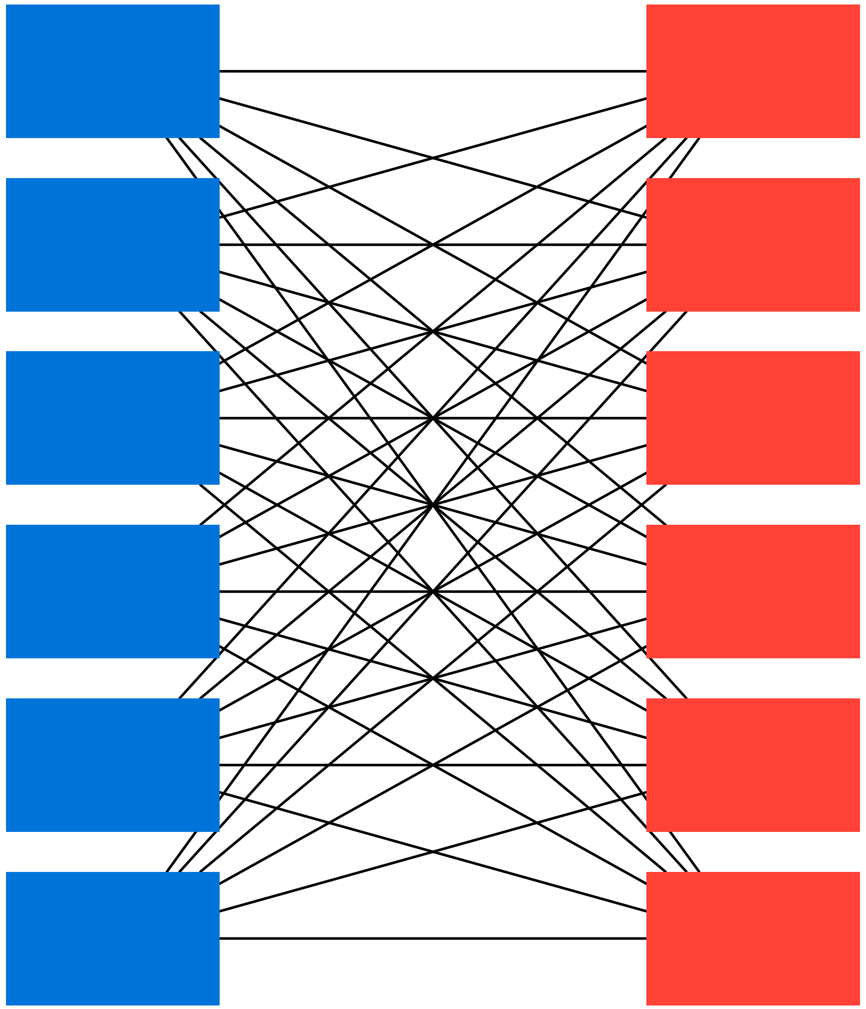
```
def product(In1, In2):
    in2_temp = [s for s in In2]
    for r in In1:
        for s in in2_temp:
            yield r + s
```

or

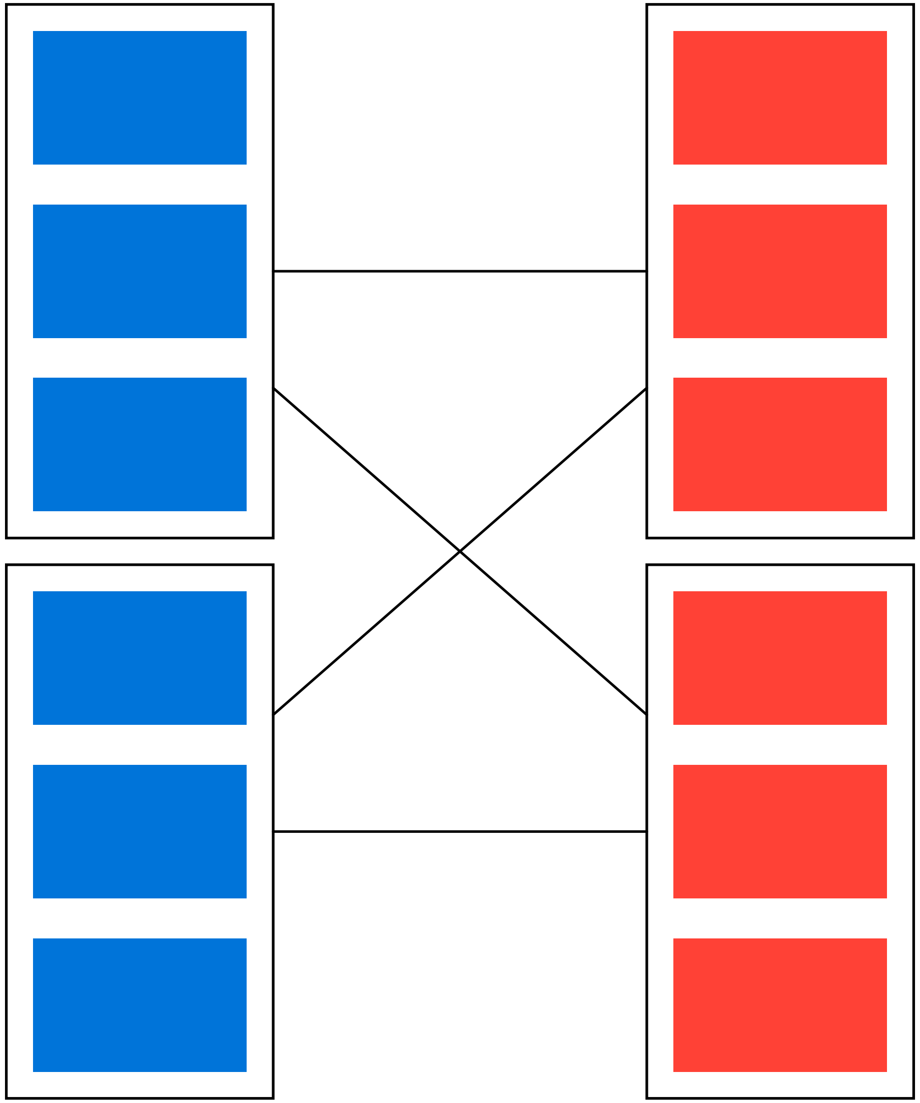
```
def product(In1, In2):
    in2_temp = write_to_file(in2)
    for r in In1:
        for s in TableIterator(in2_temp):
            yield r + s
```

What are the IO & Memory Complexities

Cartesian Product



Cartesian Product



Block-Nested Loop

```
def product(In1, In2):
    in2_temp = write_to_file(In2)
    for r_batch in In1.by_batch:
        for s in TableIterator(in2_temp):
            for r in r_batch:
                yield r + s
```

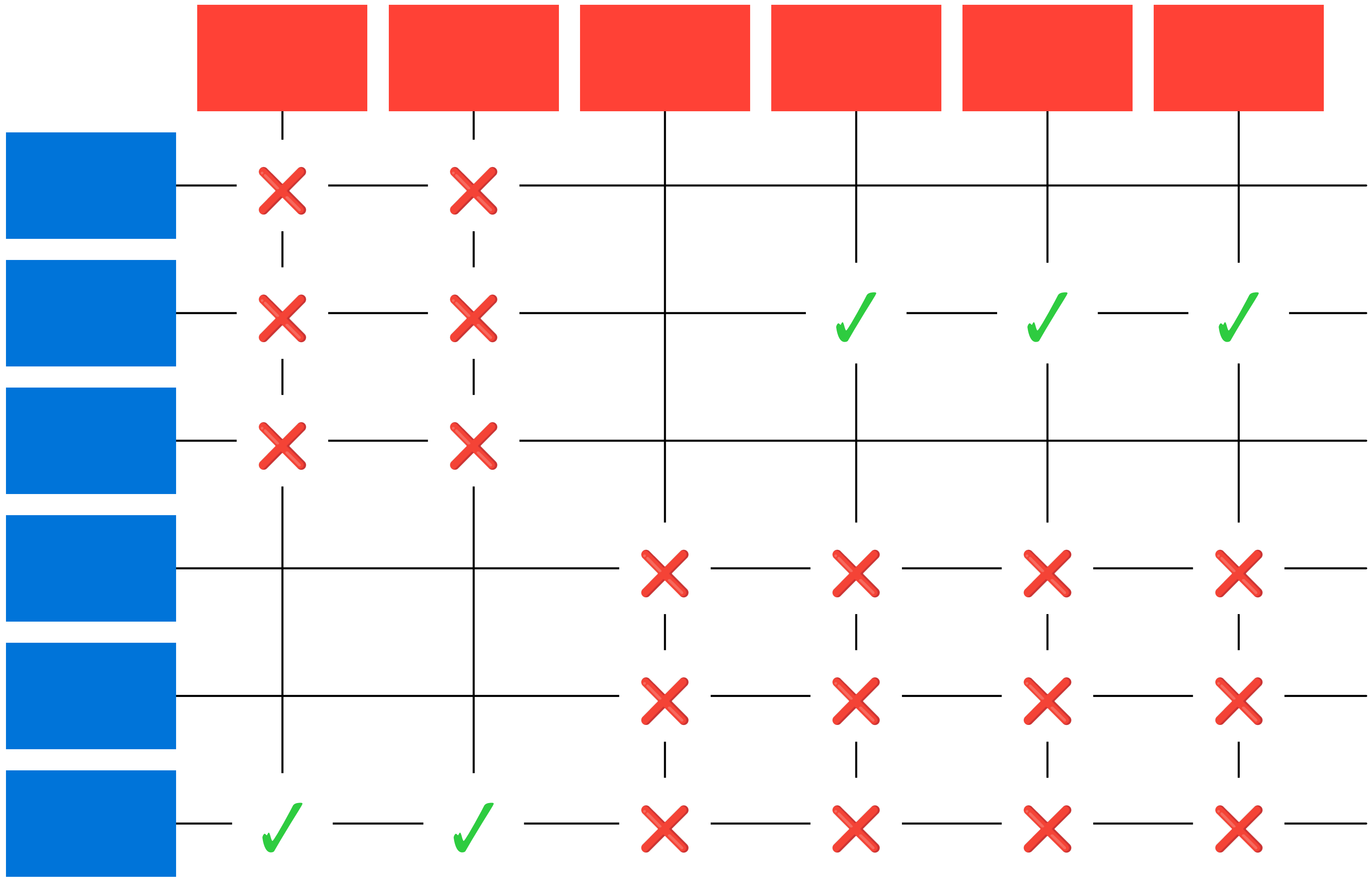
Block-Nested Loop

```
def product(In1, In2):
    in2_temp = write_to_file(In2)
    for r_batch in In1.by_batch:
        for s in TableIterator(in2_temp):
            for r in r_batch:
                yield r + s
```

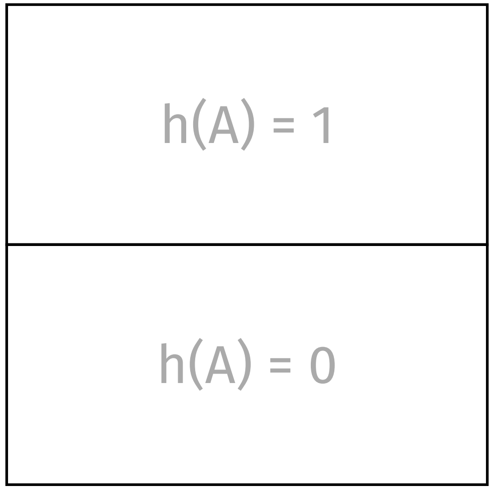
What are the IO & Memory Complexities

Joins

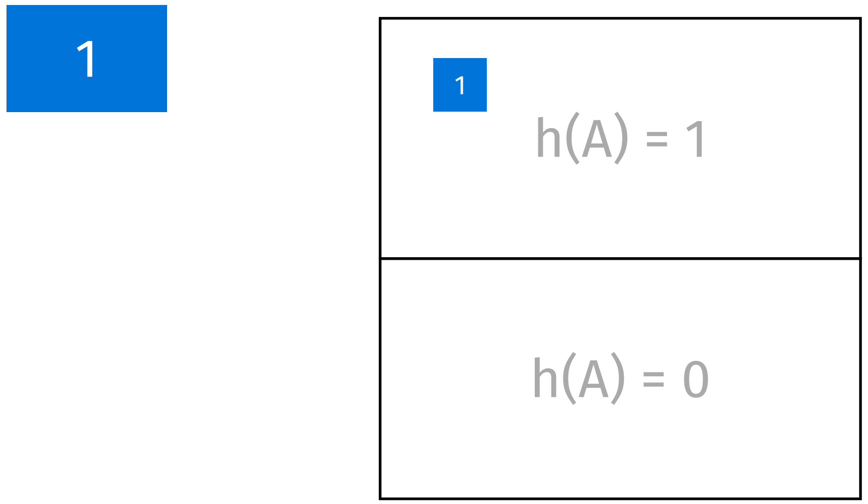
Join



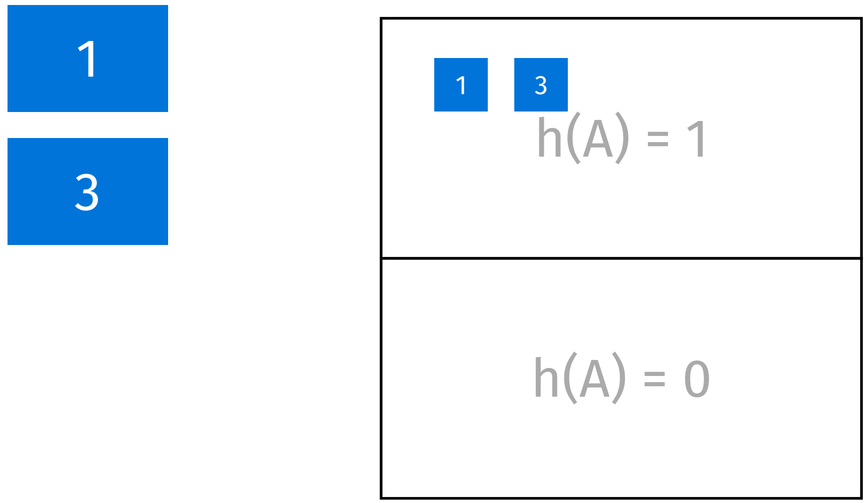
1-Pass Hash Join



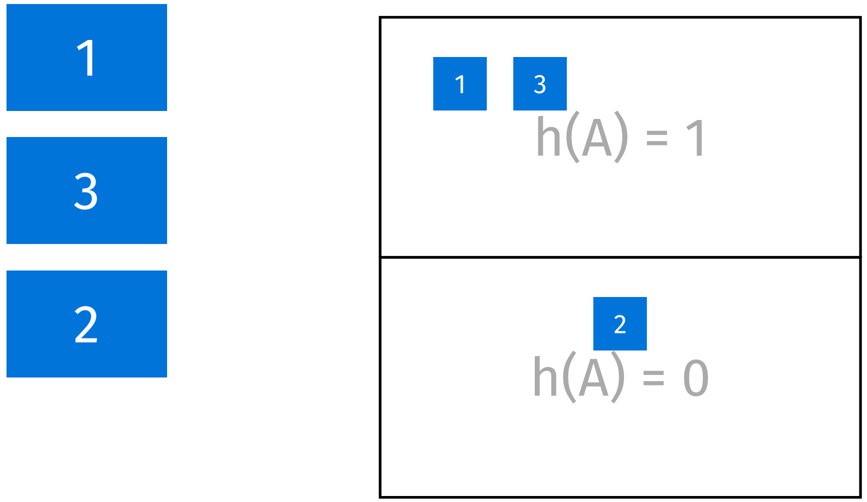
1-Pass Hash Join



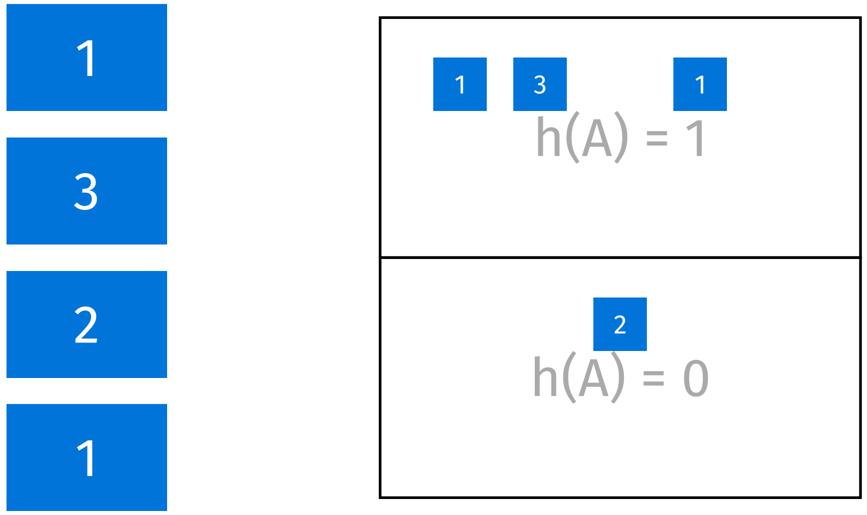
1-Pass Hash Join



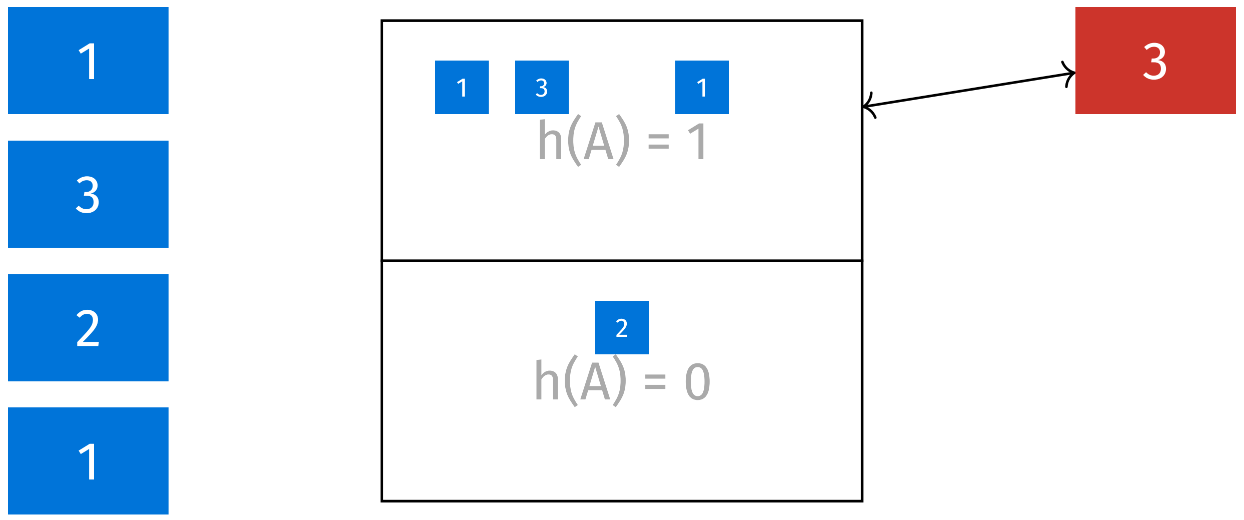
1-Pass Hash Join



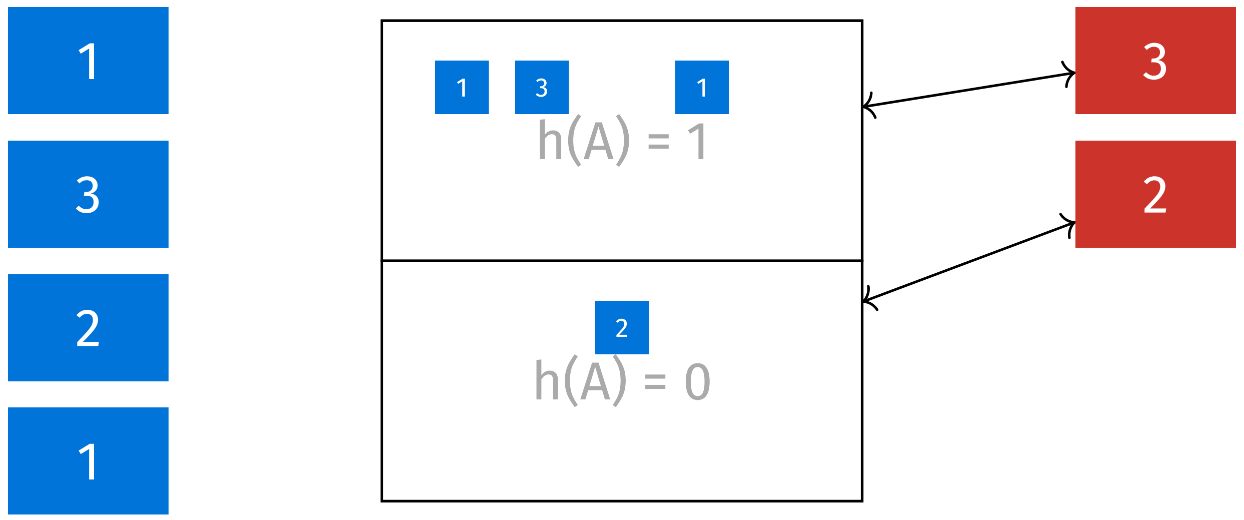
1-Pass Hash Join



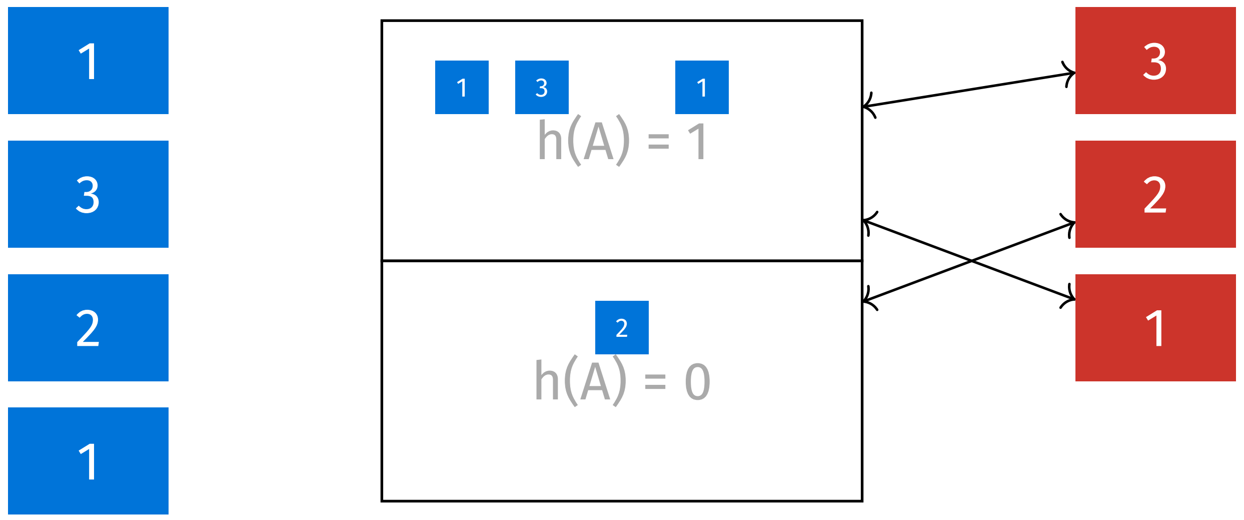
1-Pass Hash Join



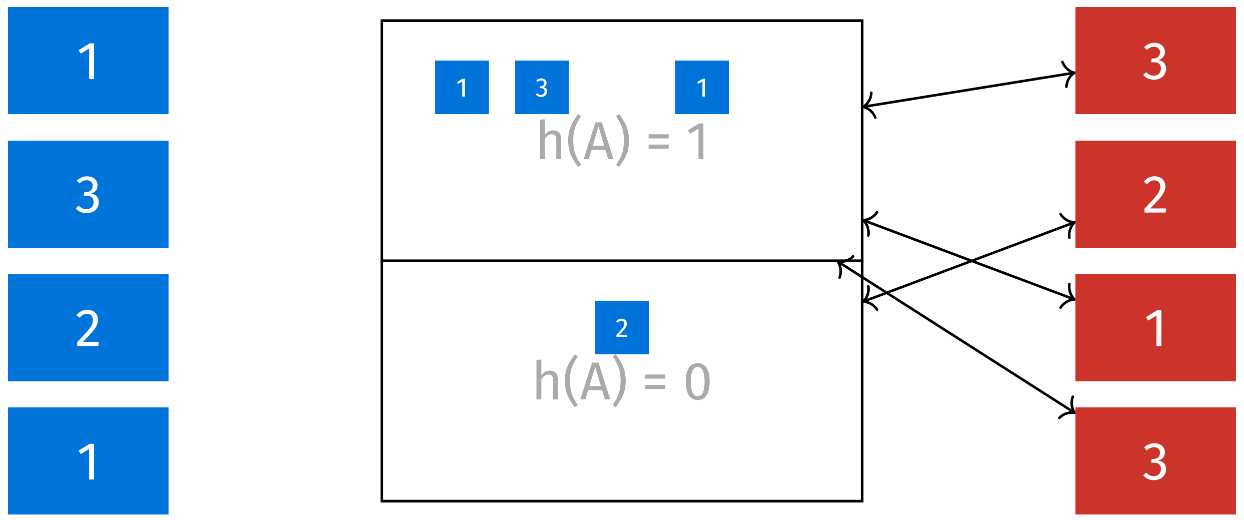
1-Pass Hash Join



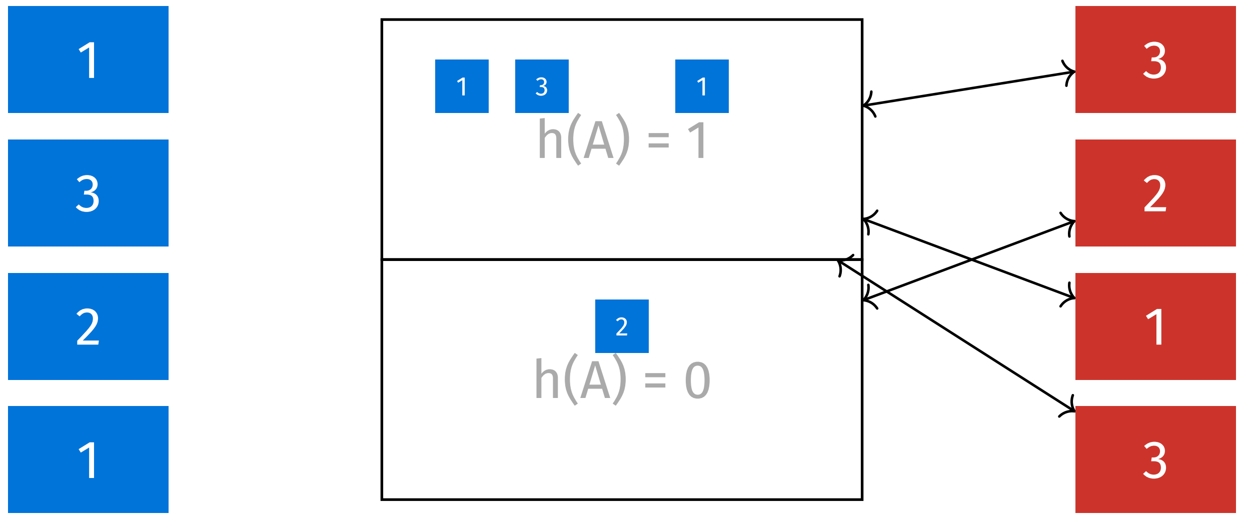
1-Pass Hash Join



1-Pass Hash Join

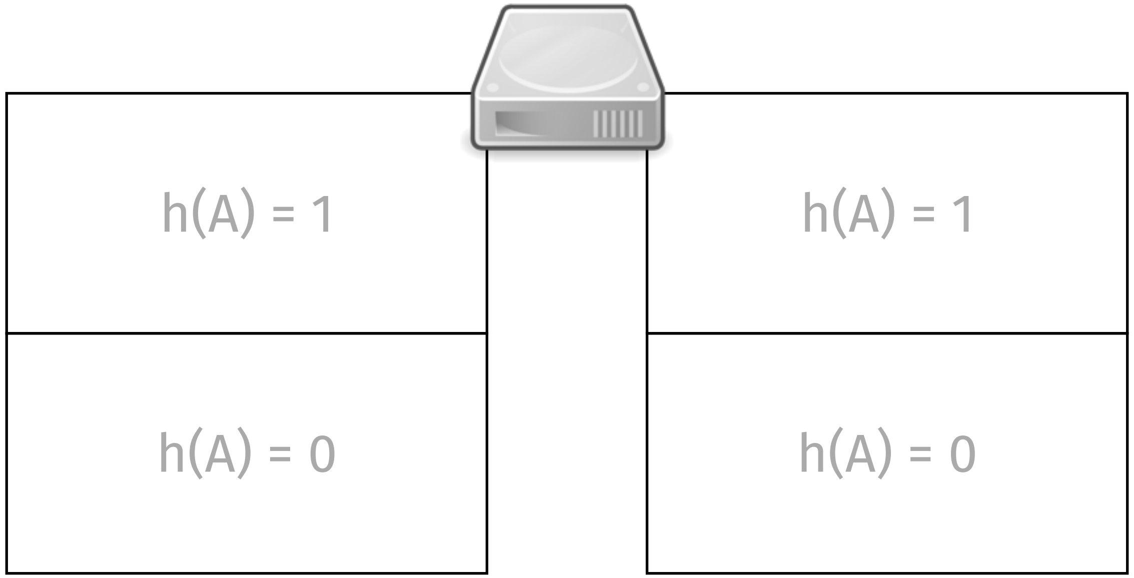


1-Pass Hash Join

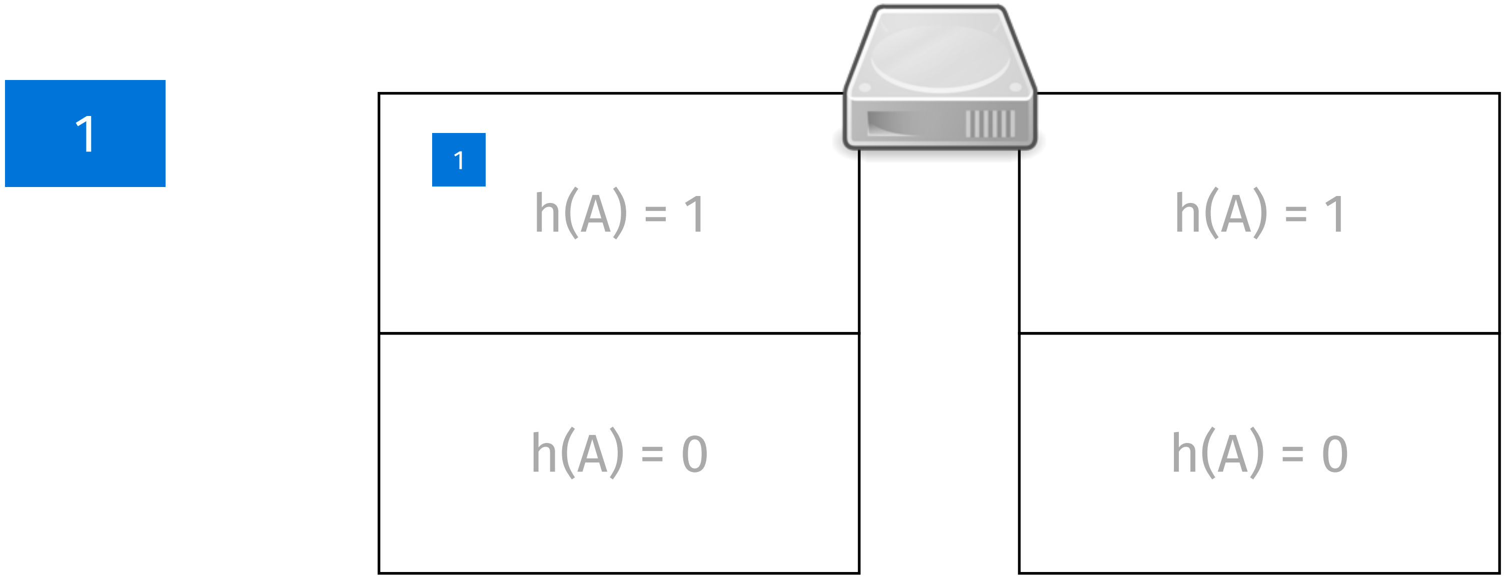


What are the IO & Memory Complexities?

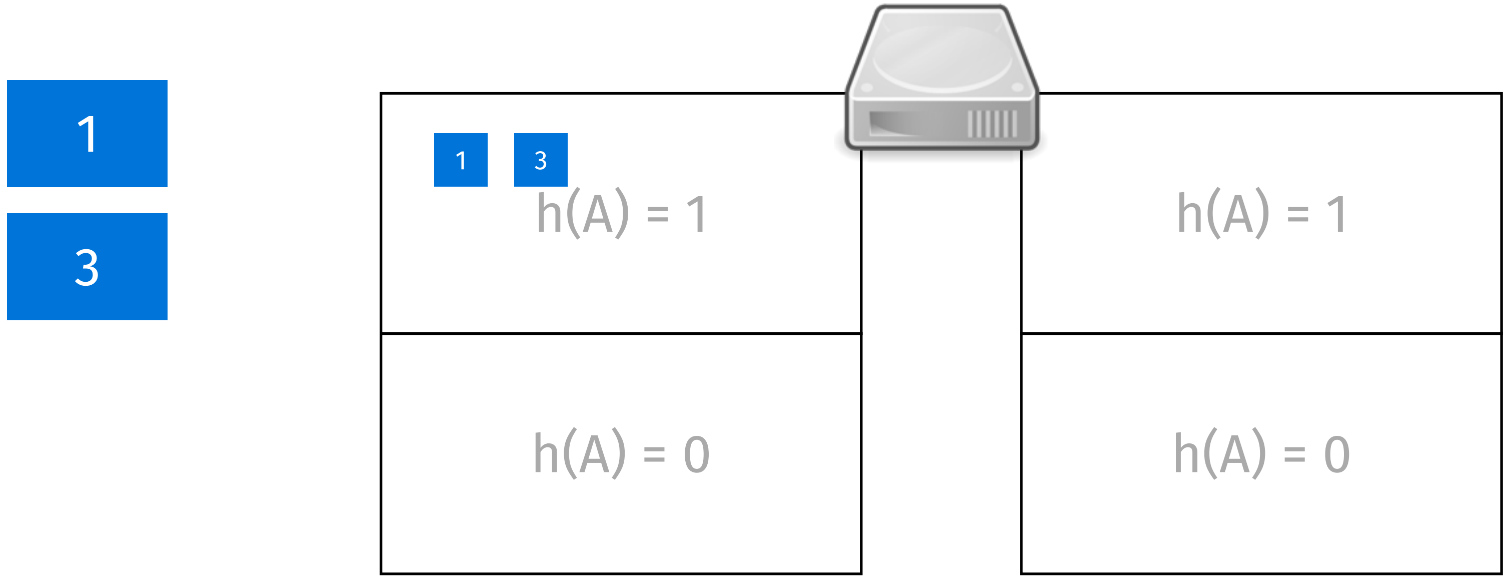
2-Pass (Grace) Hash Join



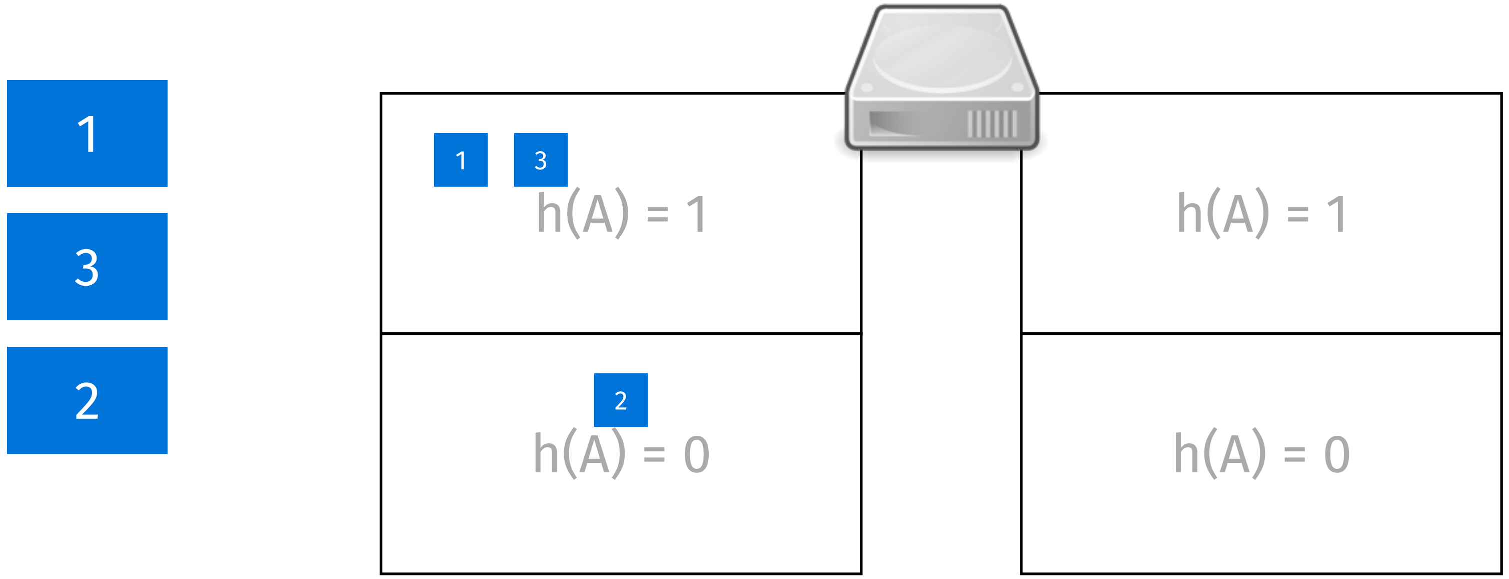
2-Pass (Grace) Hash Join



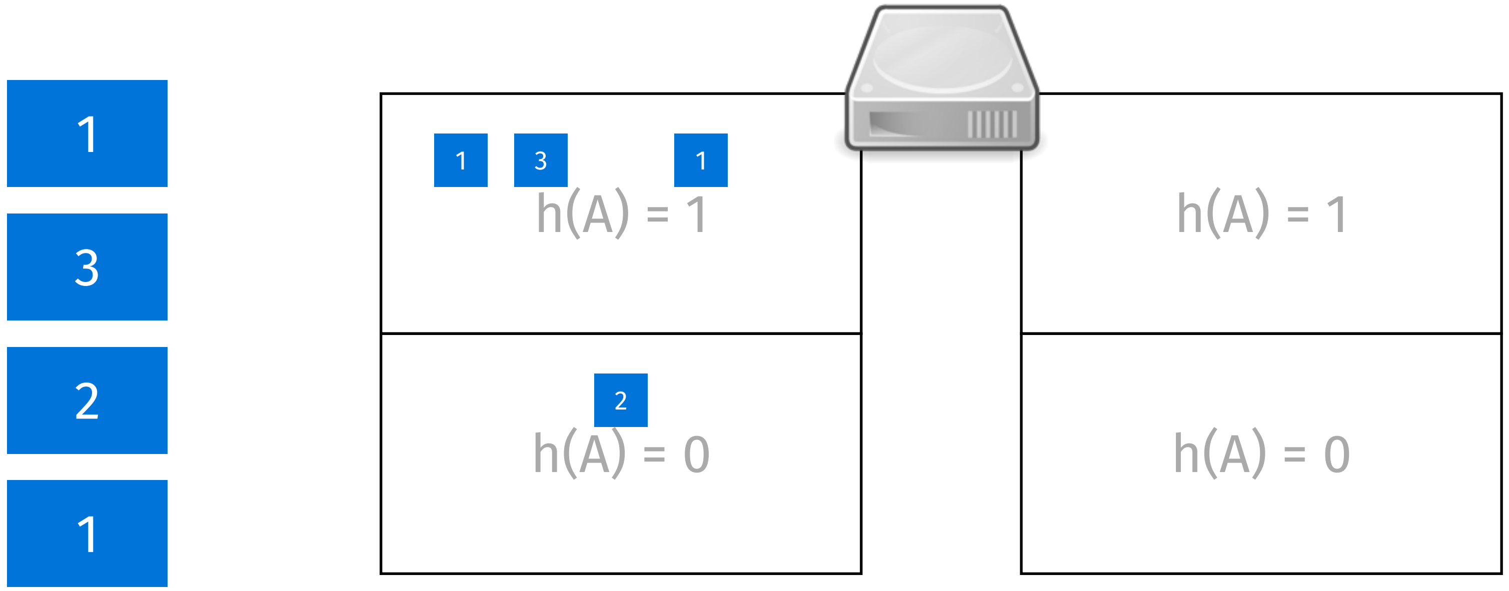
2-Pass (Grace) Hash Join



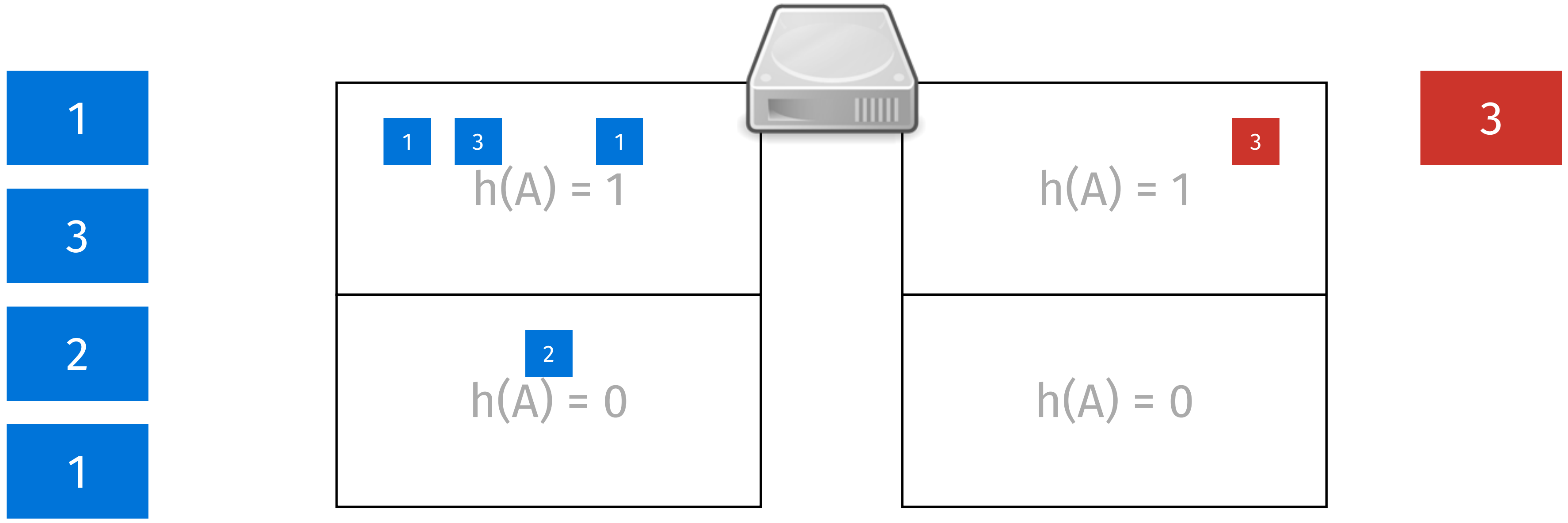
2-Pass (Grace) Hash Join



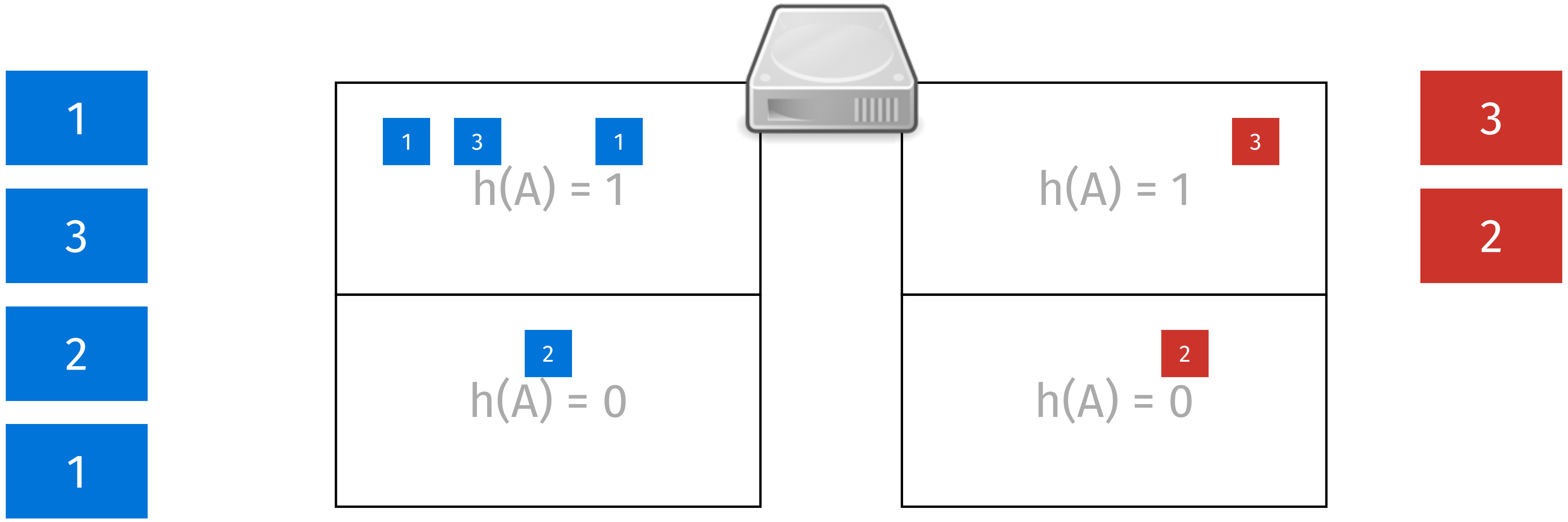
2-Pass (Grace) Hash Join



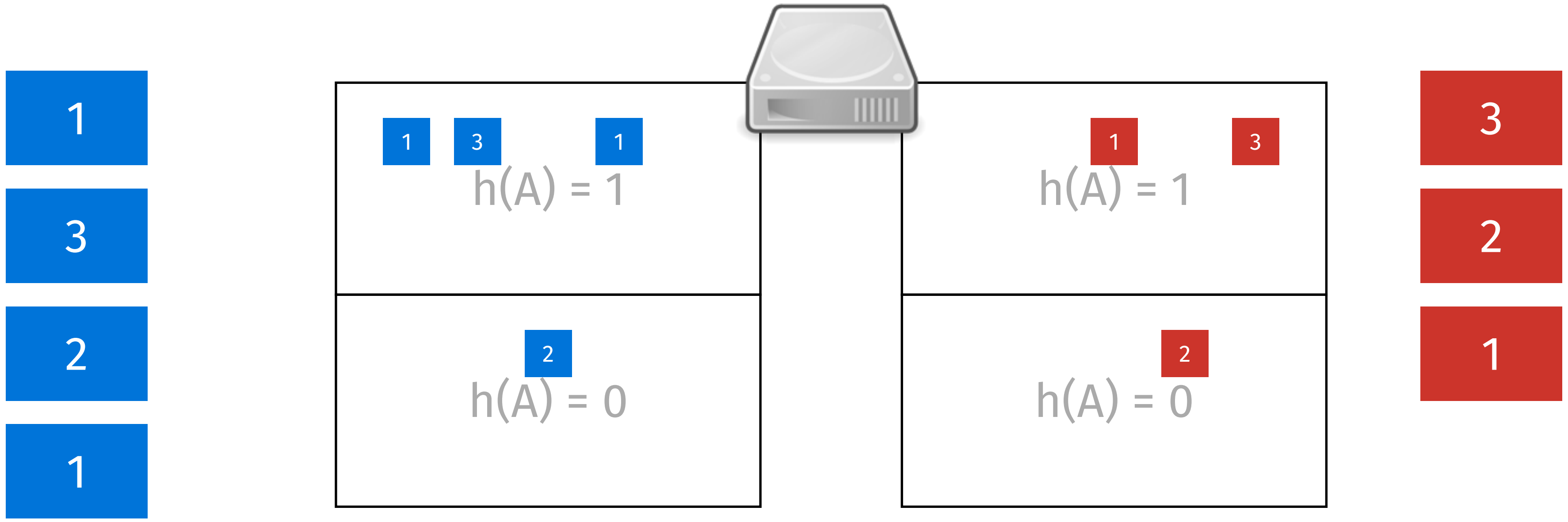
2-Pass (Grace) Hash Join



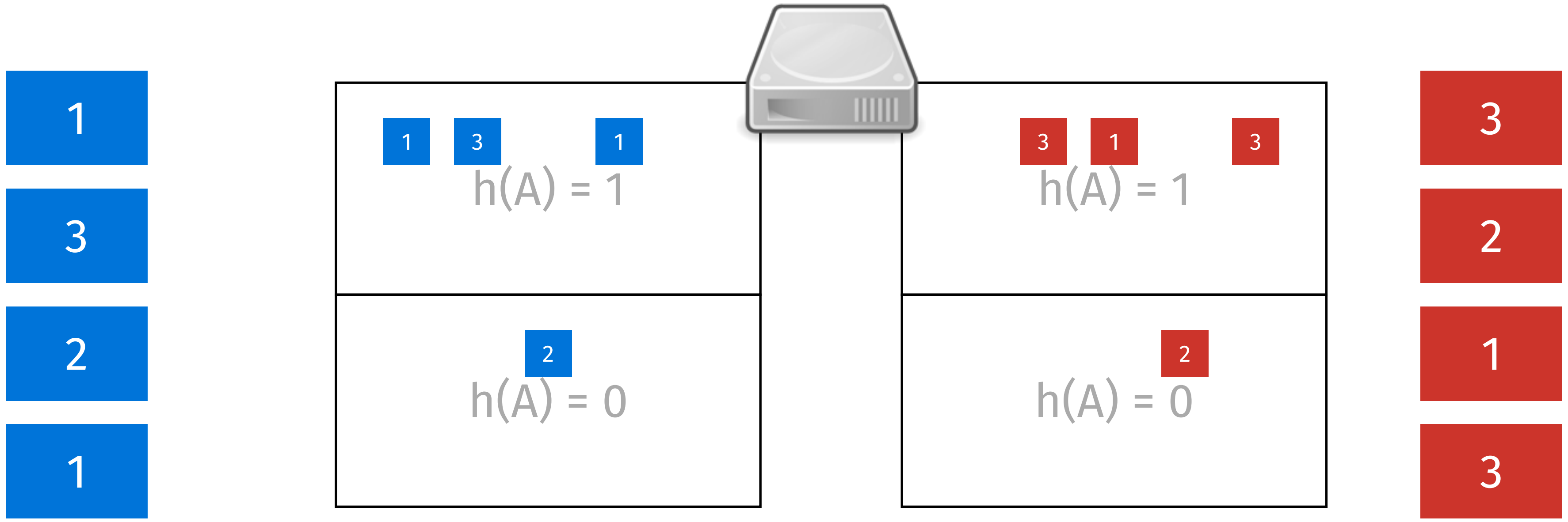
2-Pass (Grace) Hash Join



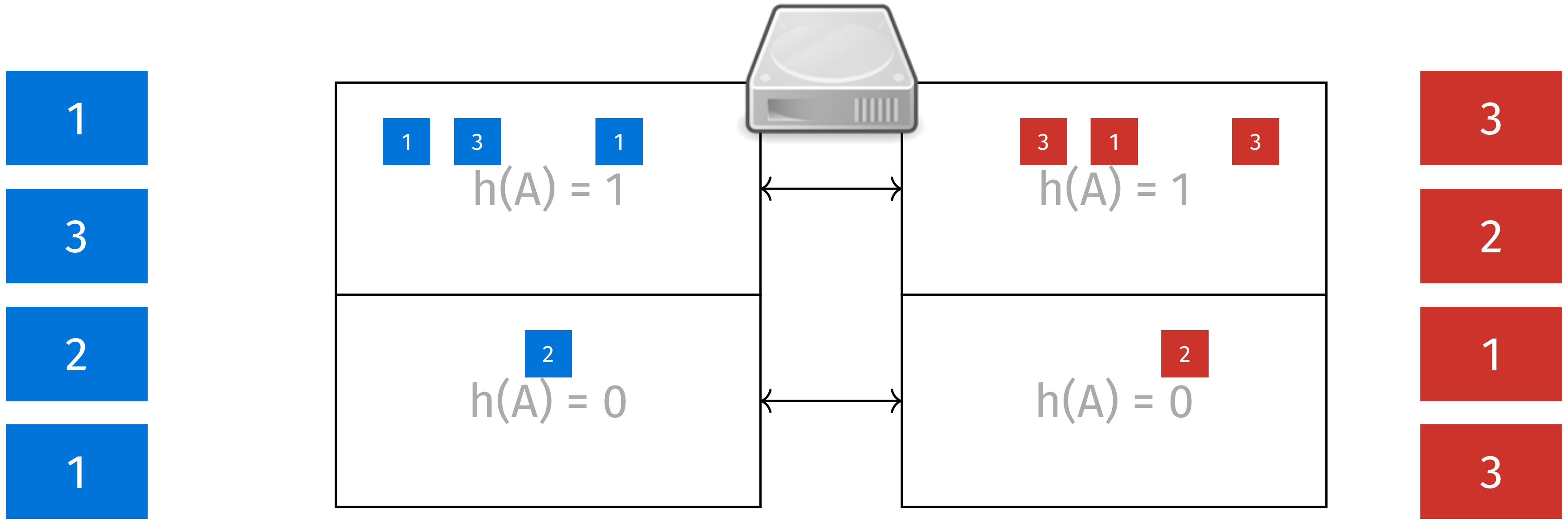
2-Pass (Grace) Hash Join



2-Pass (Grace) Hash Join



2-Pass (Grace) Hash Join

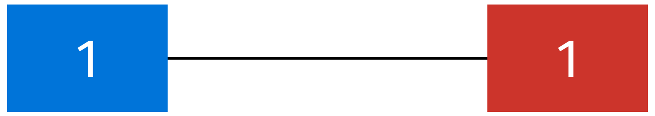


What are the IO & Memory Complexities?

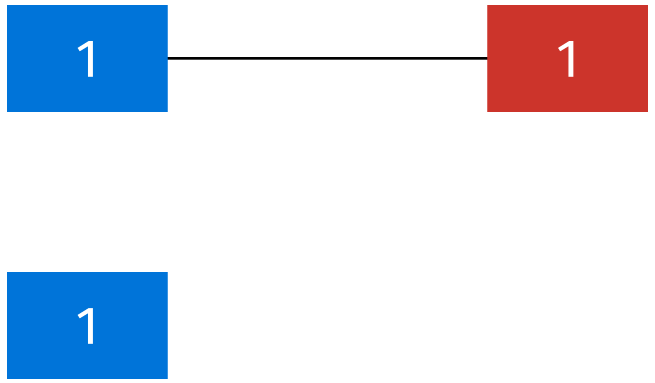
Sort-Merge Join

Sort-Merge Join

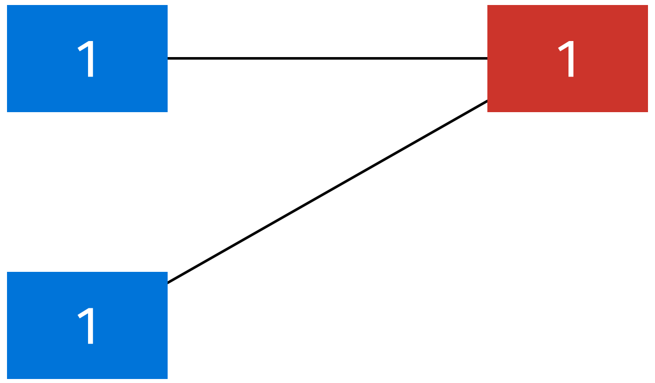
Sort-Merge Join



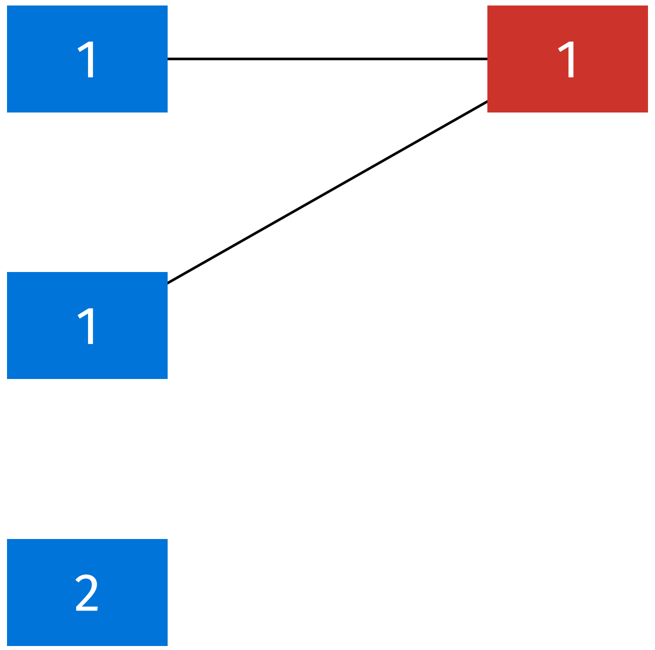
Sort-Merge Join



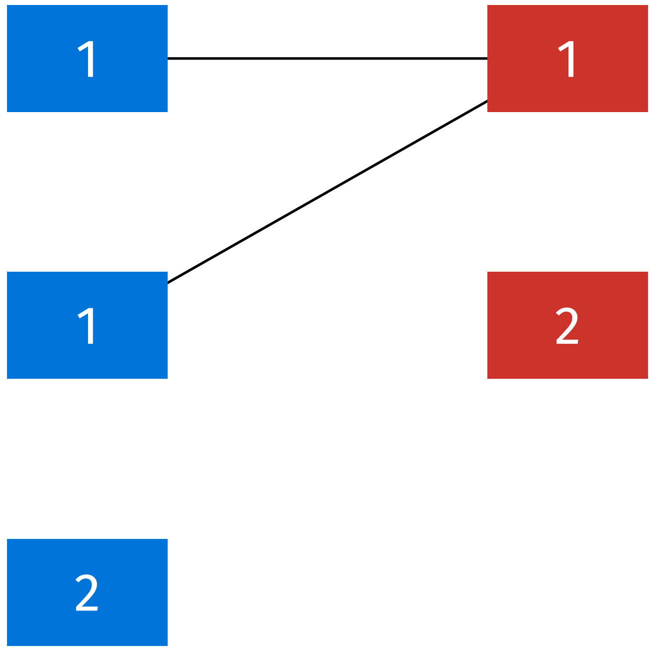
Sort-Merge Join



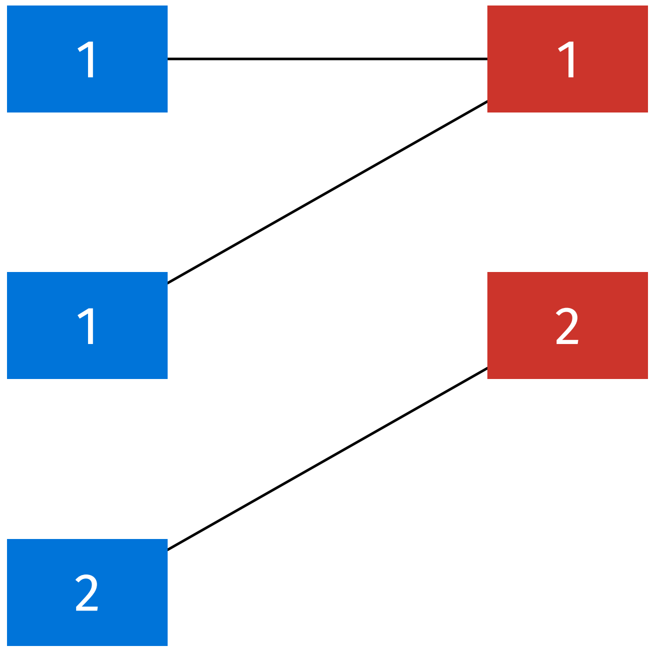
Sort-Merge Join



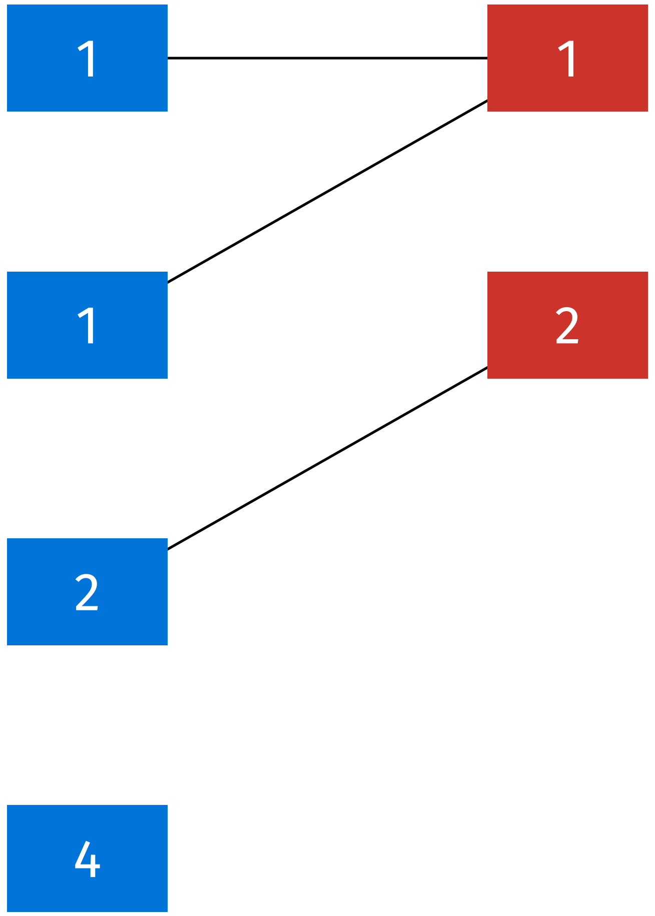
Sort-Merge Join



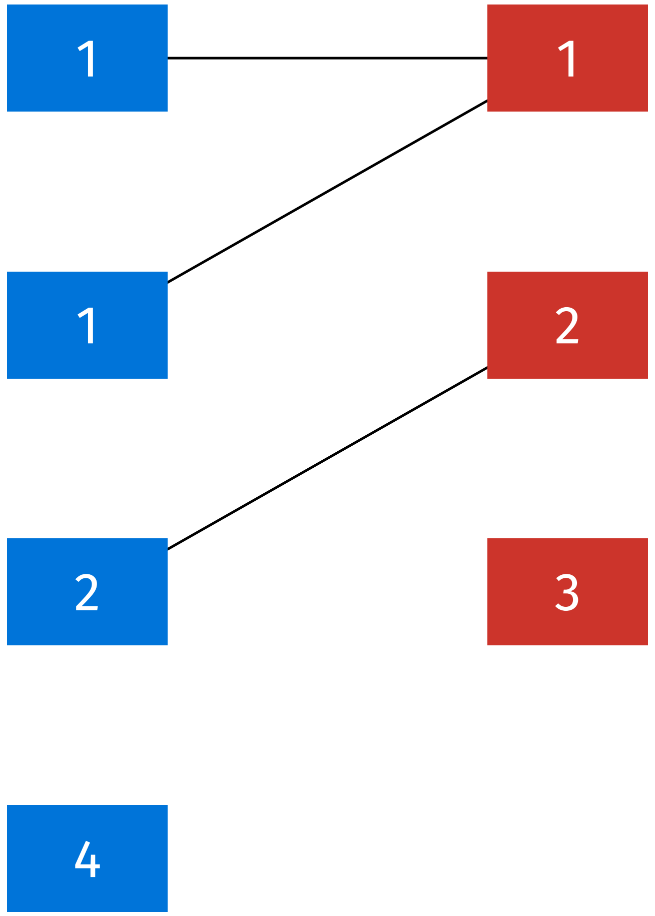
Sort-Merge Join



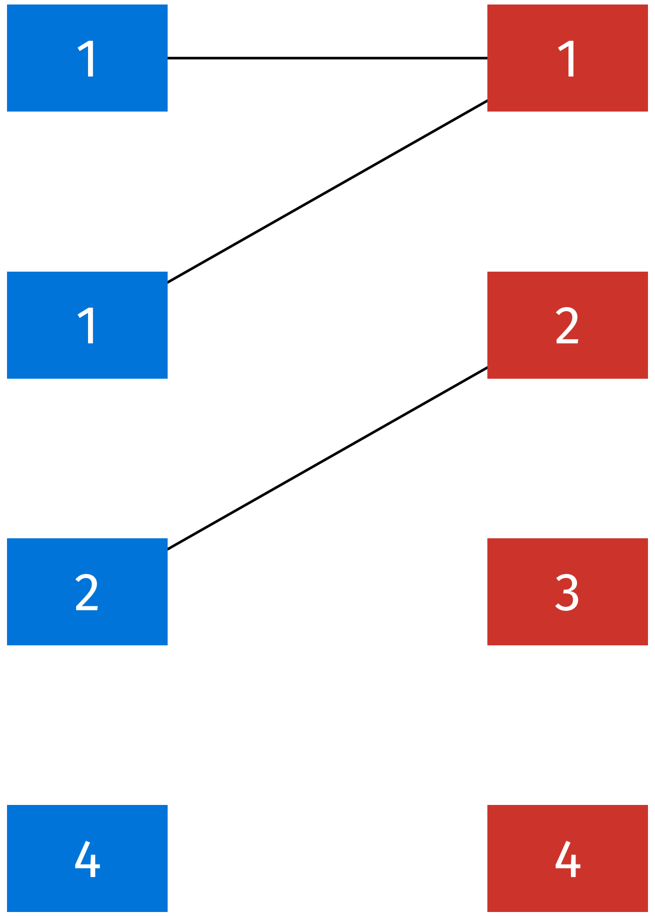
Sort-Merge Join



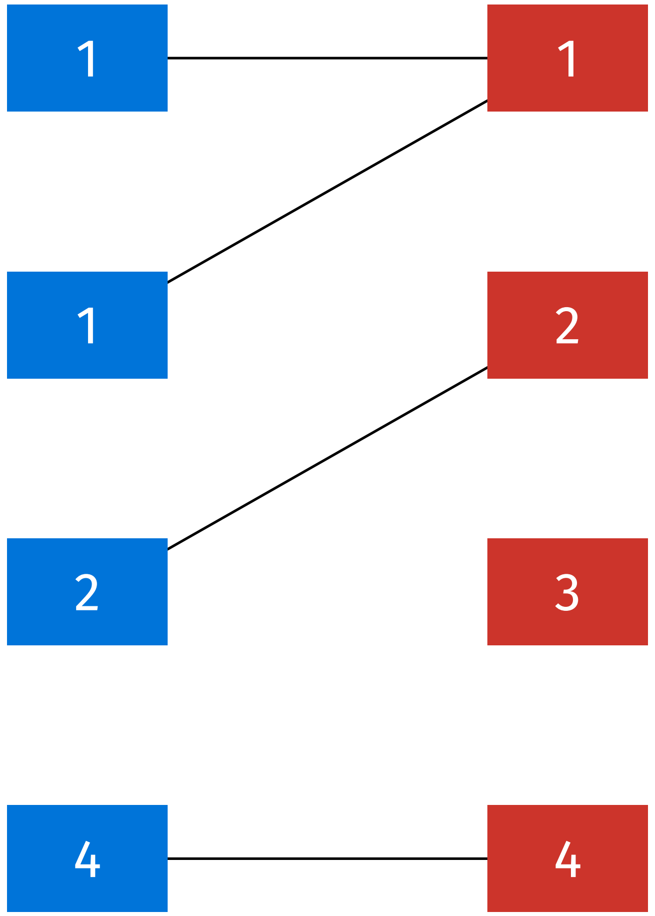
Sort-Merge Join



Sort-Merge Join



Sort-Merge Join



Sort-Merge Join

```
def sort_merge_join(key, In1, In2):
    In1 = In1.sorted();    In2 = In2.sorted()
    row_1 = In1.next();    row_2 = In2.next()

    while row_1 is not None and row_2 is not None:
        if row_1[key] == row_2[key]:
            yield row_1 + row_2
        elif row_1[key] < row_2[key]:
            row_1 = In1.next()
        else:
            row_2 = In2.next()
```

What are the Memory and IO complexities?

Join Patterns

For $R \bowtie S$

- **Product** (aka Nested Loop): $|R| + |R| \cdot |S|$ extra IO, $O(1)$ mem, General
- **Block Nested Loop**: General, $|R| + \frac{|R| \cdot |S|}{|B|}$ extra IO, $O(|B|)$ mem, General
- **1-Pass Hash Join**: No extra IO, $O(|R|)$ mem Equijoin Only
- **2-Pass (Grace) Hash Join**: $2 \cdot (|R| + |S|)$ extra IO, $O(1)$ mem, Equijoin Only
- **Sort/Merge Join**: No extra IO, $O(1)$ mem, Equijoin Only, Data must be sorted

Aggregation

Aggregate Functions

- `Init()`: The "default" value of the aggregate
- `Accumulate(agg, value)`: Incorporate a new value
- `Finalize(agg) -> value`: Post-process the aggregate

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum			

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity
Count			

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity
Count	`0`	`a + 1`	identity

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity
Count	`0`	`a + 1`	identity
Max			

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity
Count	`0`	`a + 1`	identity
Max	$-\infty$	`max(a, v)`	identity

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity
Count	`0`	`a + 1`	identity
Max	$-\infty$	`max(a, v)`	identity
Average			

Aggregate Functions

Aggregate	`Init()`	`Accumulate(a, v)`	`Finalize(a)`
Sum	`0`	`a + v`	identity
Count	`0`	`a + 1`	identity
Max	`-∞`	`max(a, v)`	identity
Average	`{s: 0, c: v}`	`{s: a.s+v, c: a.c+1}`	`a.s/a.c`

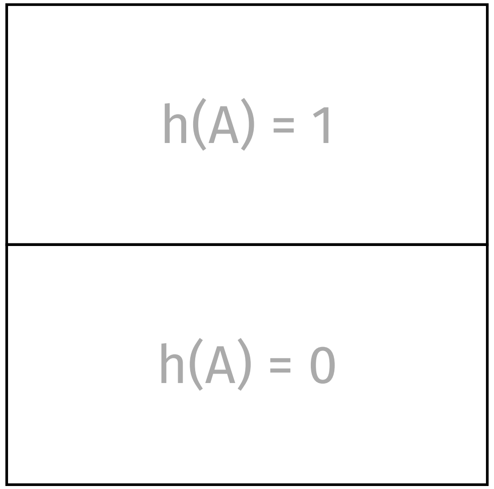
Aggregation

```
def aggregate(group_by, In):
    groups = defaultdict(Init)

    for row in In:
        groups[row[group_by]] = Accumulate(groups[row[group_by]],
                                           row)

    for (key, value) in groups
        yield (key, Finalize(value))
```

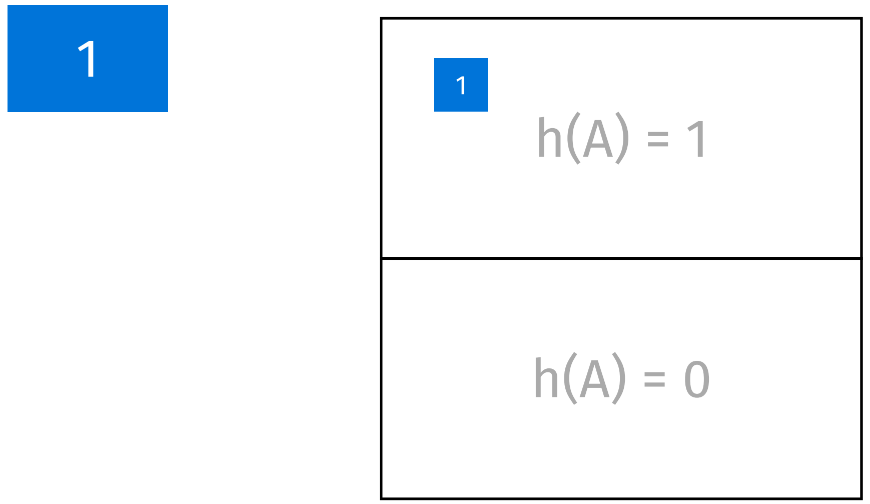
What are the Memory and IO complexities?

What if there’s not enough memory?

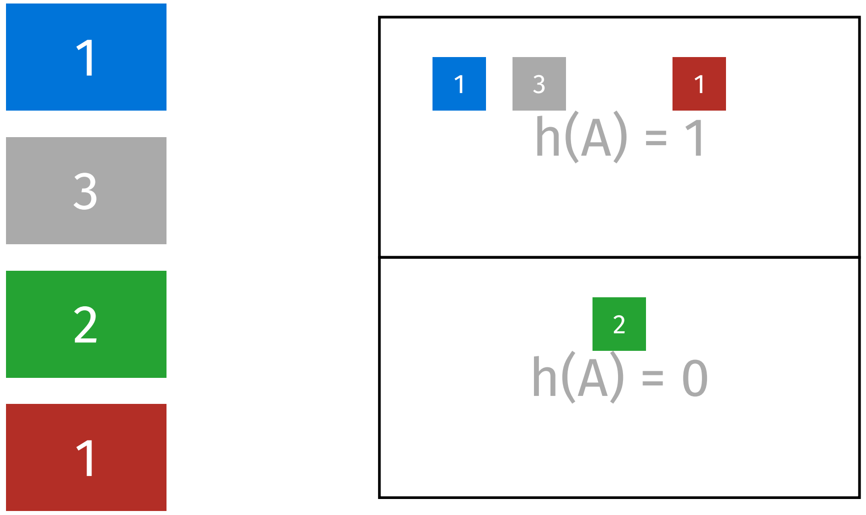
Aggregation



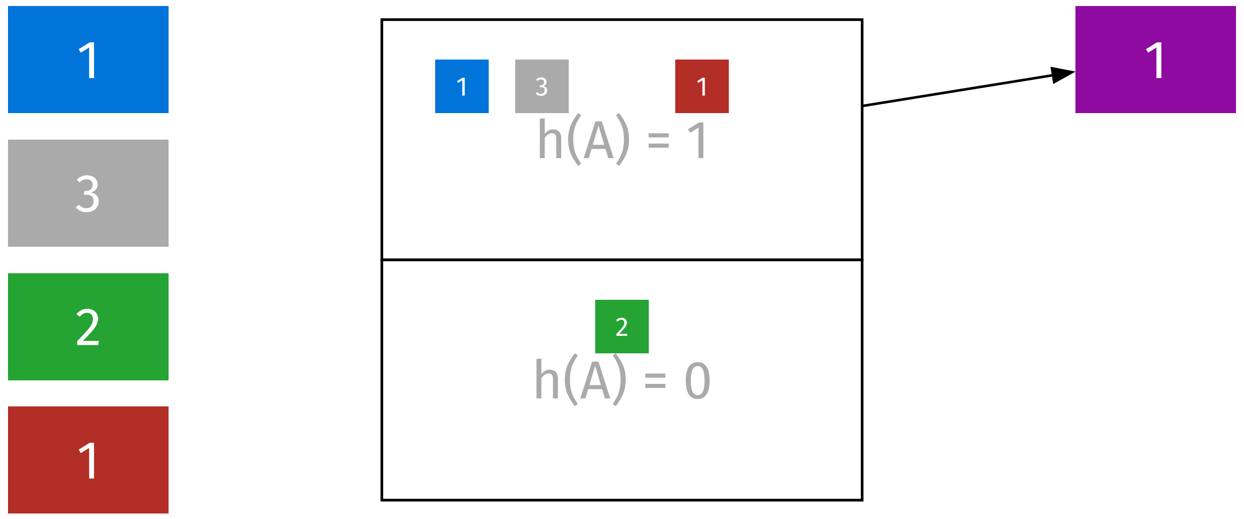
Aggregation



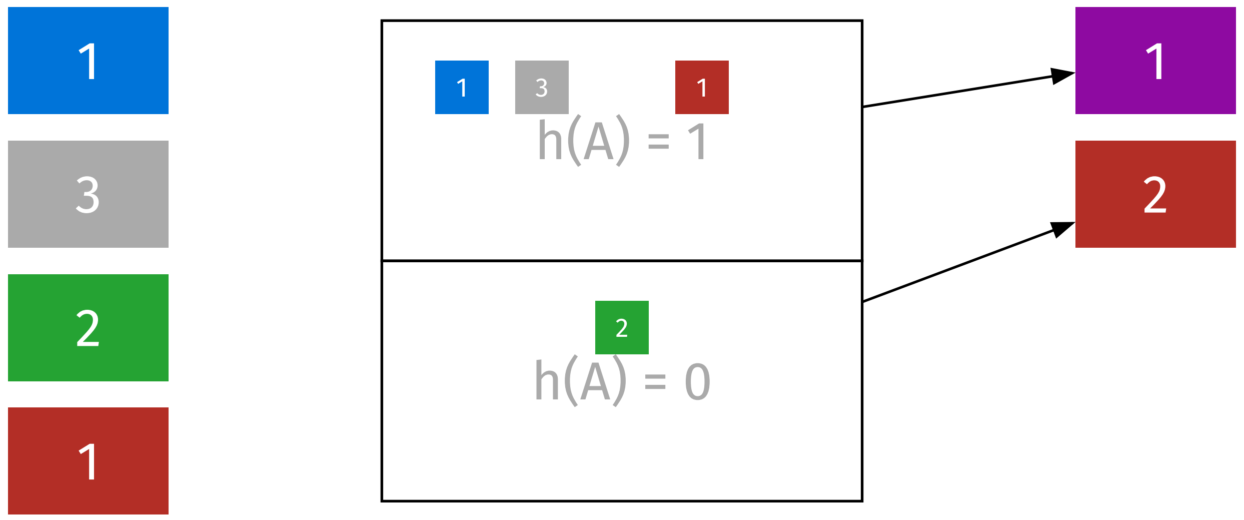
Aggregation



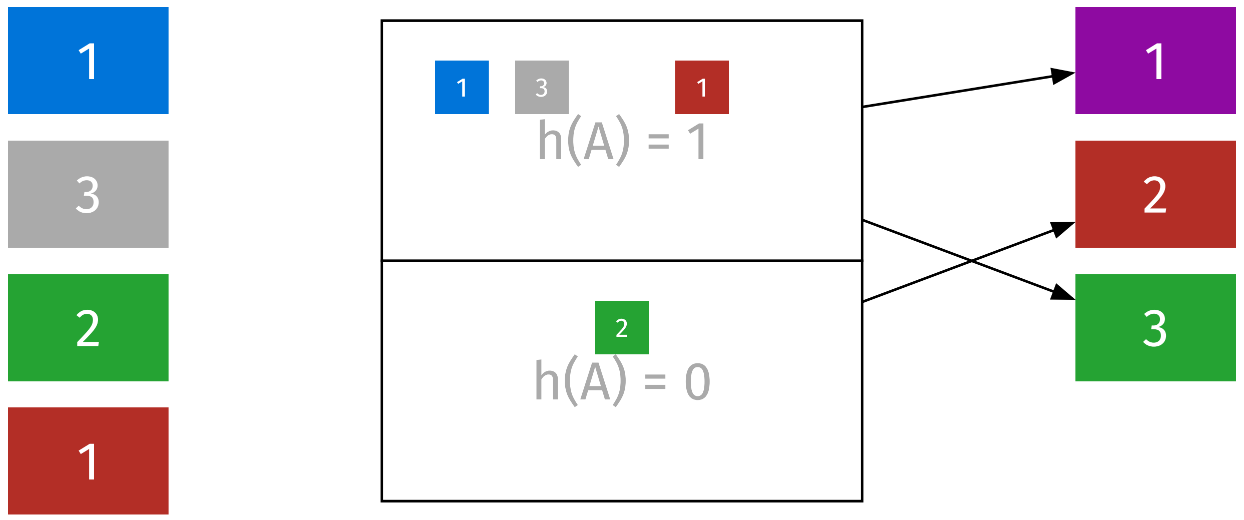
Aggregation



Aggregation



Aggregation



Aggregation

```
def partitioned_aggregate(group_by, In):
    partitions = [ TempFile() for i in range(PARTITIONS) ]

    for row in In:
        partitions[ row[group_by].hash() % PARTITIONS ].write(row)

    for partition_file in partitions:
        groups = defaultdict(Init)
        for row in partition_file:
            groups[row[group_by]] = Accumulate(groups[row[group_by]], row)
        for (key, value) in groups
            yield (key, Finalize(value))
```

What are the Memory and IO complexities?

Aggregation

```
def partitioned_aggregate(group_by, In):
    In = In.sorted()
    group = None;       agg = None
    while row := In.next() is not None:
        if row[group_by] != group:
            if group is not None:
                yield group, agg
            group = row[group_by];  agg = Init()
        agg = Accumulate(agg, row)

    if group is not None:
        yield group, Finalize(agg)
```

What are the Memory and IO complexities?

Updates

Deliverables

- AI Quiz and Checkpoint 0 Past Due!
  - Reach out to me if you haven't finished it yet!
- Checkpoint 1 due Monday!