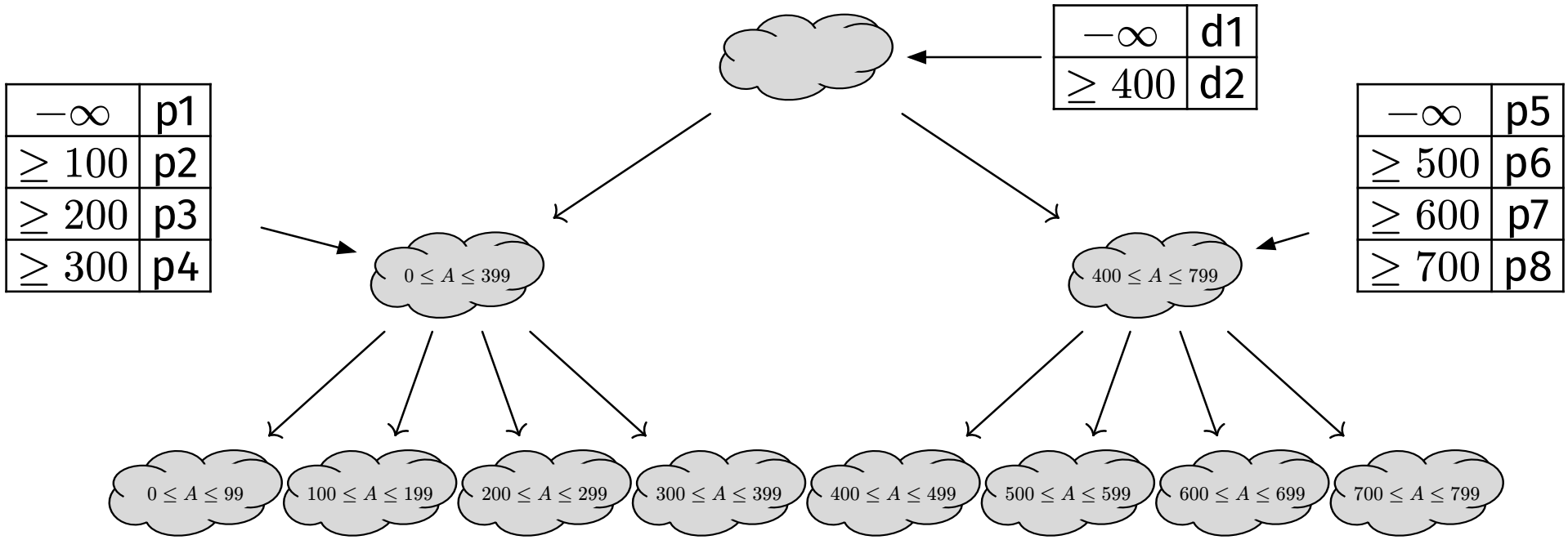
Hash-Based Indexes, Complex Keys

CSE 4/562: Database Systems | Lecture 9

DB. Sys.: T.C.B.: Ch. 8.3-8.4, 14.1-14.2, 14.4


−∞ d1
≥ 400 d2
−∞ p1
≥ 100 p2
≥ 200 p3
≥ 300 p4
−∞ p5
≥ 500 p6
≥ 600 p7
≥ 700 p8
0 ≤ A ≤ 399
400 ≤ A ≤ 799
0 ≤ A ≤ 99
100 ≤ A ≤ 199
200 ≤ A ≤ 299
300 ≤ A ≤ 399
400 ≤ A ≤ 499
500 ≤ A ≤ 599
600 ≤ A ≤ 699
700 ≤ A ≤ 799

Tree Index Scan


```
class TreeIndexScan:
  def init():
    # find leaf page for first record in result range

  def next():
    while current_key < last_key:
      yield current_record
      # step to next record
```

Tree Index Scan

`TreeIndexScan(R, θ)`

- IO: $O(\log(|R|) + |\sigma_\theta(R)|)$
- Memory: $O(1)$
- θ: Any 1 attribute, equality or range[1]

$\sigma_\theta(R)$

- IO: $O(|R|)$
- Memory: $O(1)$
- θ: Any condition at all

[1] We generalize this slightly later

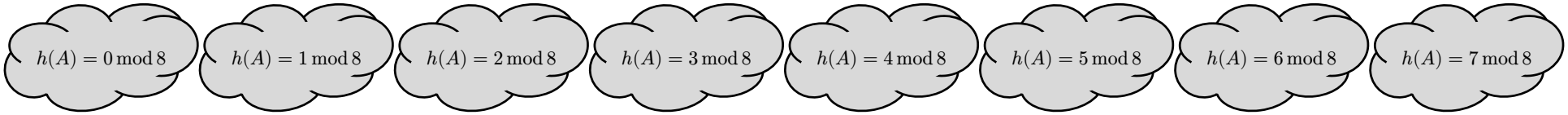
Hash Indexes

A hash function $h(k)$ is a function that is:

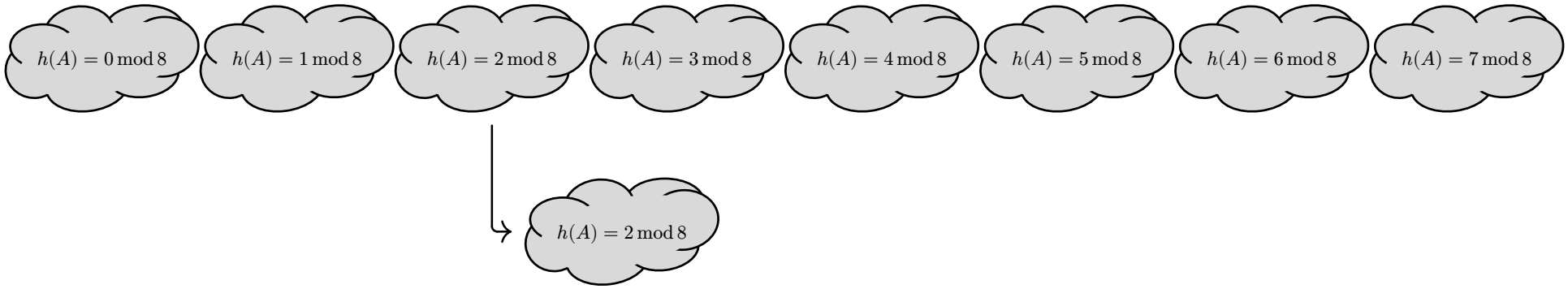
- **deterministic**: The same k always produces the same output.
- **(pseudo-)random**: Different ks produce uncorrelated outputs, even if the inputs are correlated.

$h(k) \bmod N$ gives you a random number in $[0, N)$

Hash Indexes



Hash Indexes: Chaining



Hash Indexes: Chaining

With...

- $|R|$ records
- P records per page
- B buckets

In expectation, we will have $\frac{|R|}{B}$ records per bucket.

In expectation each bucket will be $\frac{|R|}{B \cdot P}$ pages.

- Each lookup/insert is an expected $\left\lceil \frac{|R|}{B \cdot P} \right\rceil$ IOs.

Ideally, $\left\lceil \frac{|R|}{B \cdot P} \right\rceil = 1$ (i.e., $\frac{|R|}{B \cdot P} < 1$)

Problem: But $|R|$ changes over time!

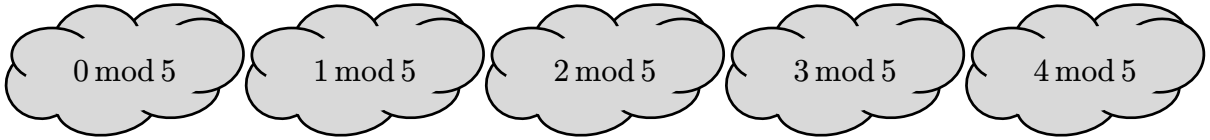
Hash Indexes

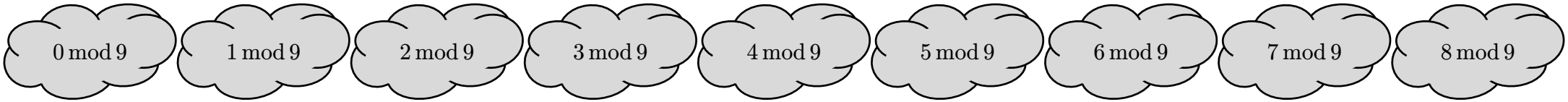
Ideally, $\left\lceil \frac{|R|}{B \cdot P} \right\rceil = 1$ (i.e., $\frac{|R|}{B \cdot P} < 1$)

Problem: But $|R|$ changes over time!

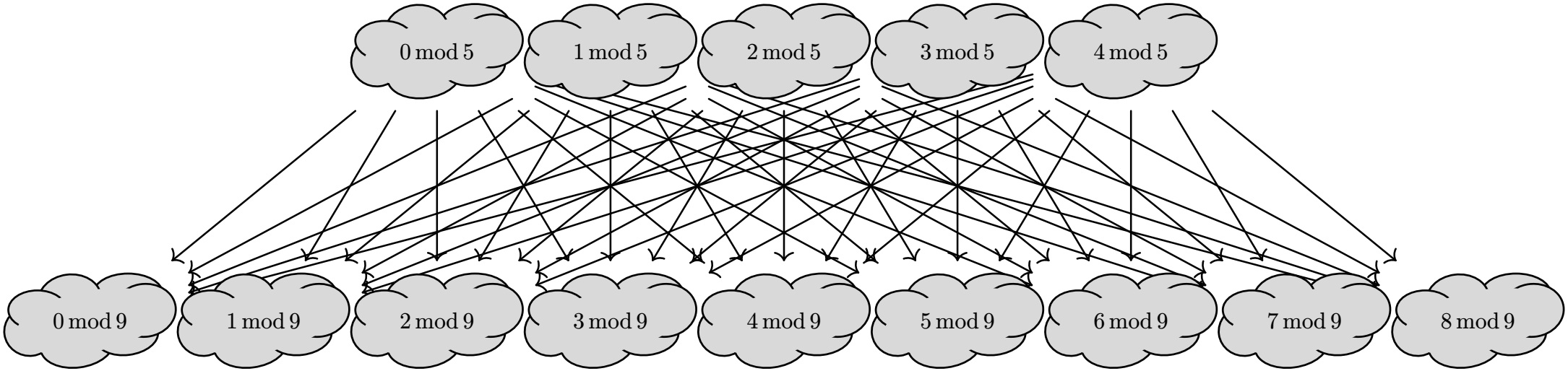
- We can't control P
- Our only option is to change B

Rehashing

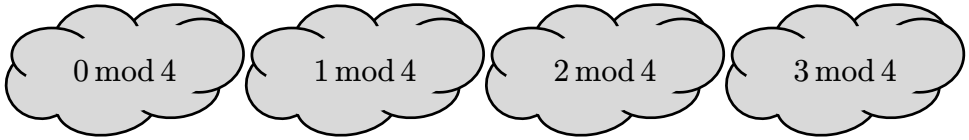

0 mod 5
1 mod 5
2 mod 5
3 mod 5
4 mod 5

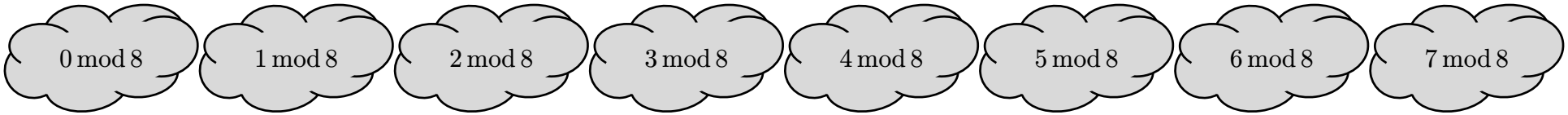

0 mod 9
1 mod 9
2 mod 9
3 mod 9
4 mod 9
5 mod 9
6 mod 9
7 mod 9
8 mod 9

Rehashing

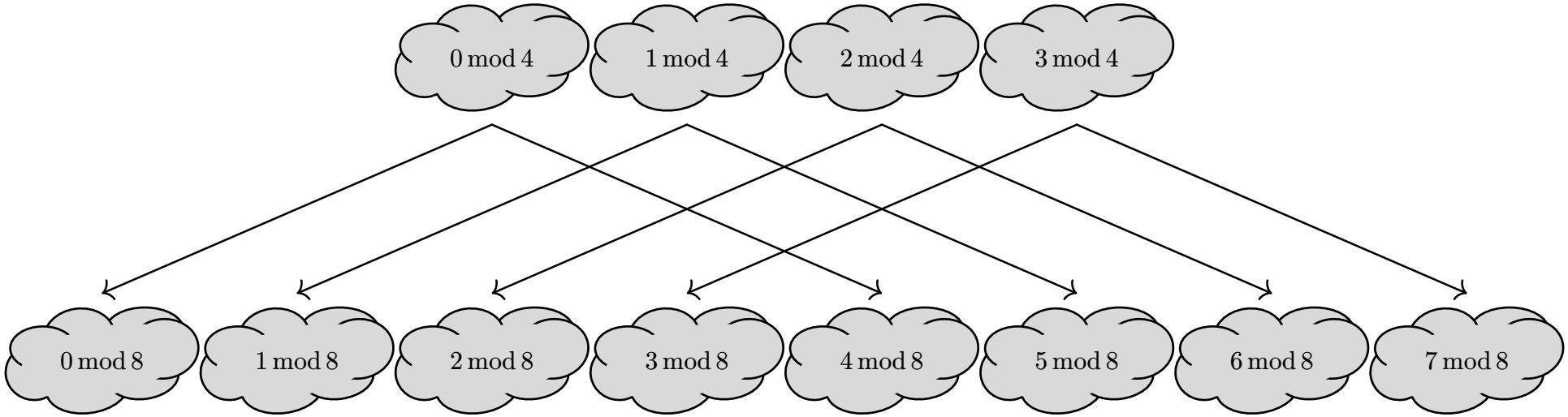

0 mod 5
1 mod 5
2 mod 5
3 mod 5
4 mod 5
0 mod 9
1 mod 9
2 mod 9
3 mod 9
4 mod 9
5 mod 9
6 mod 9
7 mod 9
8 mod 9

Rehashing

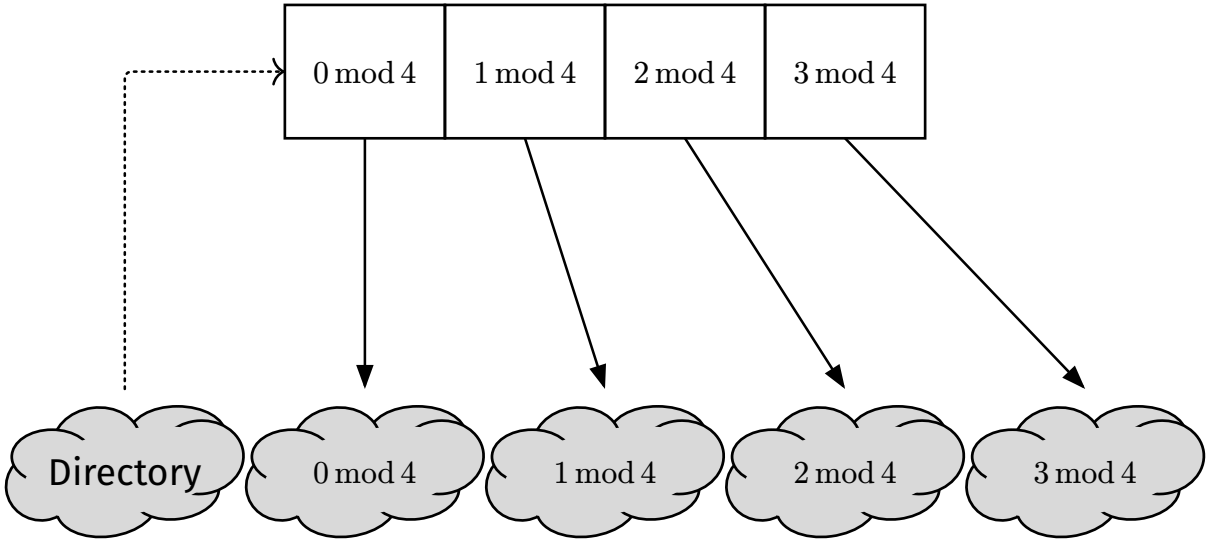

0 mod 4
1 mod 4
2 mod 4
3 mod 4


0 mod 8
1 mod 8
2 mod 8
3 mod 8
4 mod 8
5 mod 8
6 mod 8
7 mod 8

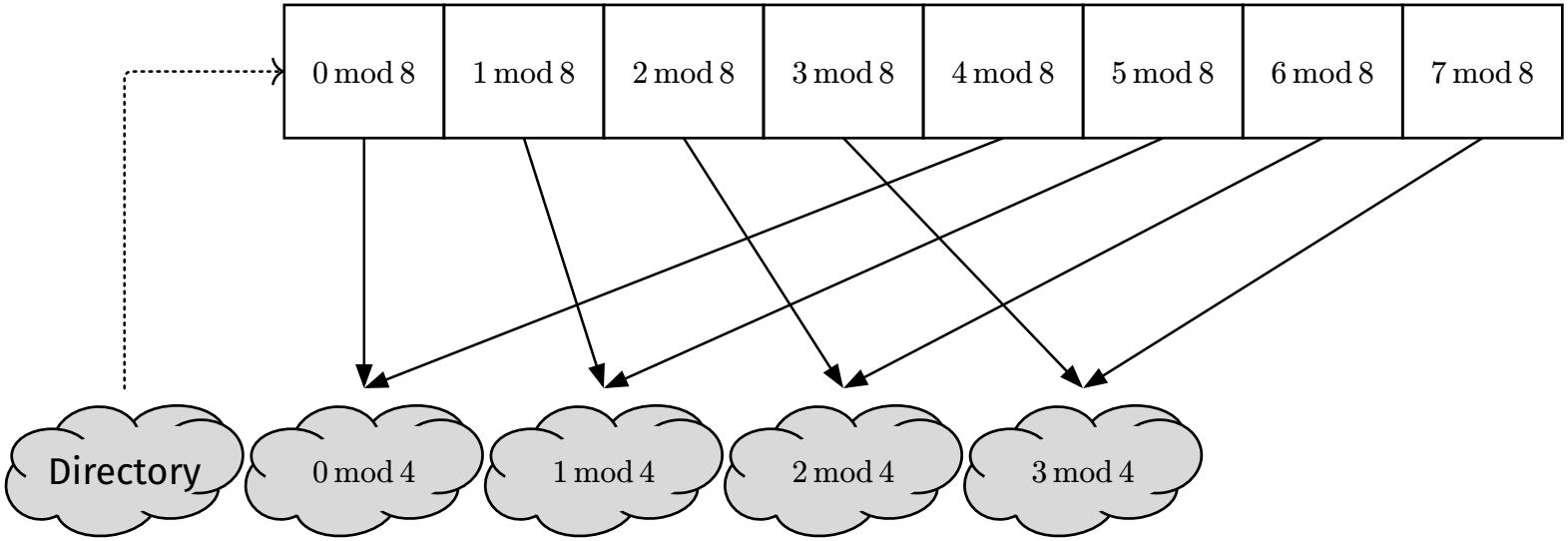
Rehashing


0 mod 4
1 mod 4
2 mod 4
3 mod 4
0 mod 8
1 mod 8
2 mod 8
3 mod 8
4 mod 8
5 mod 8
6 mod 8
7 mod 8

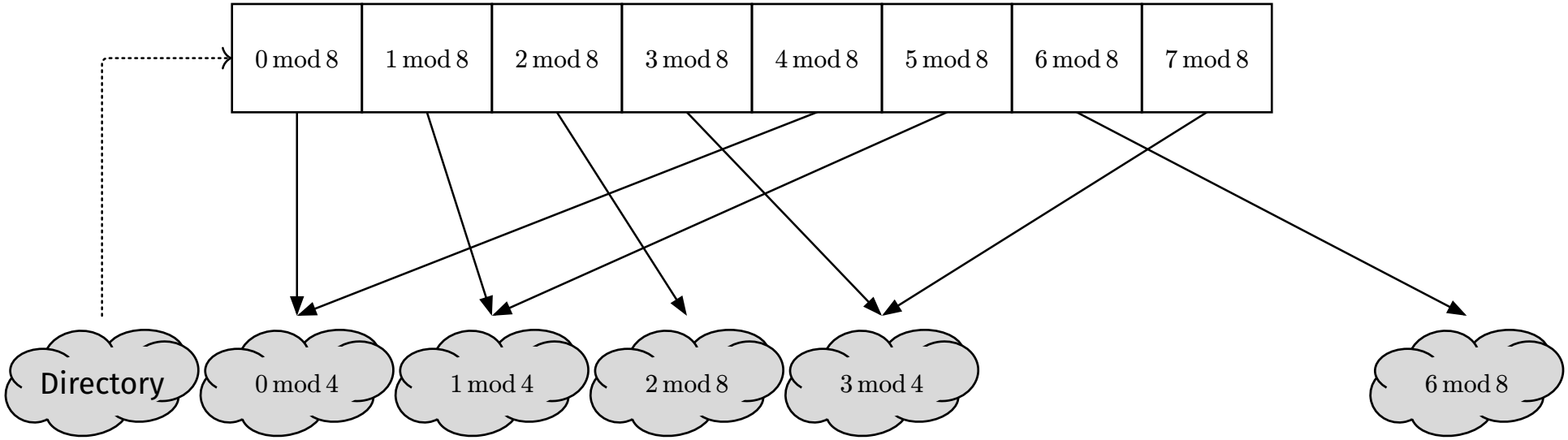
Dynamic Hashing



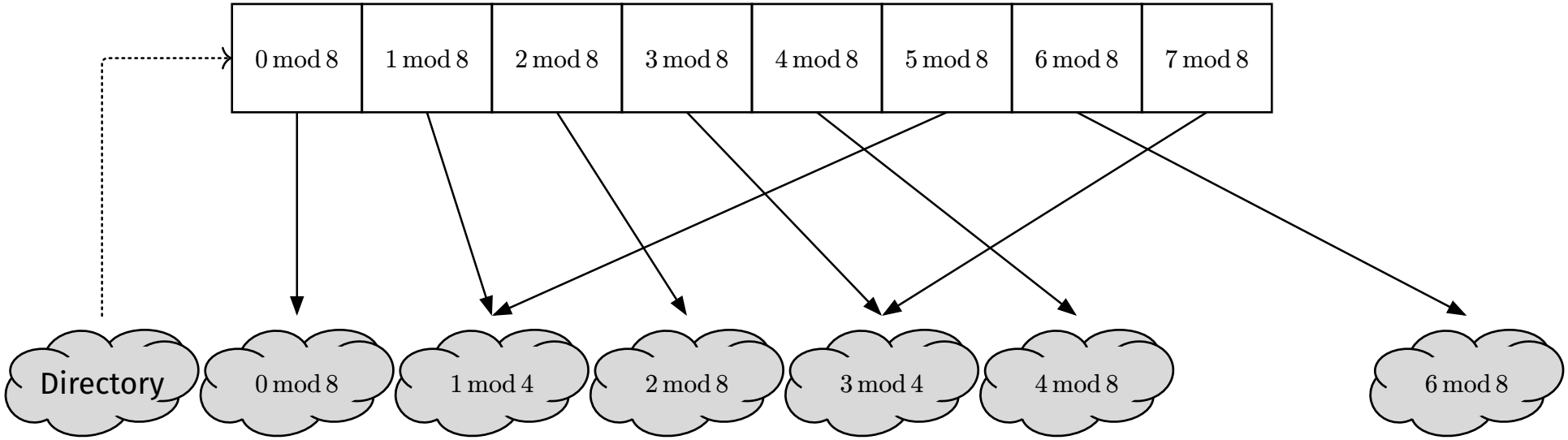
Dynamic Hashing



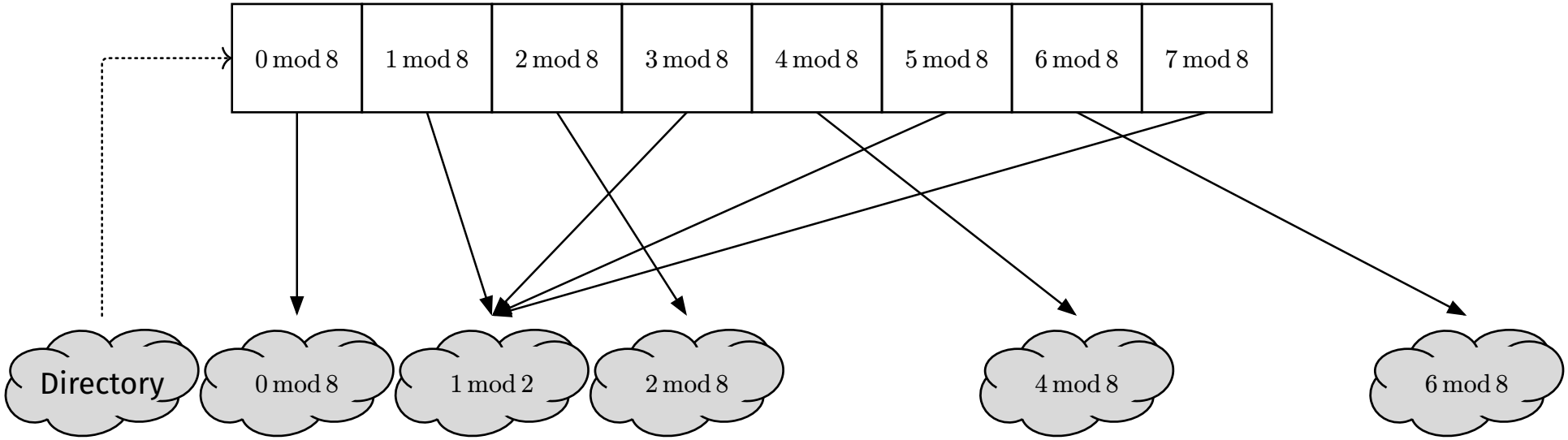
Dynamic Hashing



Dynamic Hashing



Dynamic Hashing



Split Page i from $\bmod N$ to $\bmod 2N$

1. Double the size of the directory if it is not already at least $2N$ entries
2. Rehash everything that was $i \bmod N$ to $i \bmod 2N$ and $i + N \bmod 2N$
3. Update directory pointers i and $i + N$

Merge pages i and $i + N$ from $\bmod 2N$ to $\bmod N$

1. Merge everything that was $i \bmod 2N$ and $i + N \bmod 2N$ into $i \bmod N$.
2. Update directory pointer $i + N$ (and $i + N + 2N$, ... if necessary)
3. Halve the size of the directory if possible

The total IO cost of doubling is still approximately $O(N)$, but the cost is more spread out.

Supporting multiple keys

Lexical sort

To answer $A = 1 \wedge B = 2 \wedge C = 3$:

Sort data with (A, B, C) as a key, **lexically** sorted.

- All records with $A = 1$ are contiguous
- All records with $A = 1 \wedge B = 2$ are contiguous
- All records with $A = 1 \wedge B = 2 \wedge C = 3$ are contiguous
- All records with $A = 1 \wedge B = 2 \wedge C \geq 3$ are contiguous

A lexically sorted tree index supports:

- Point lookups of the key
- Range lookups of all records matching an exact **prefix** of the key
- As above, but where the last attribute in the prefix is a range query

How about multiple sort orders?

Multiple Sort Orders

Can we build a single B+ Tree that supports both of these?

```
SELECT * FROM R WHERE R.A > 3
```

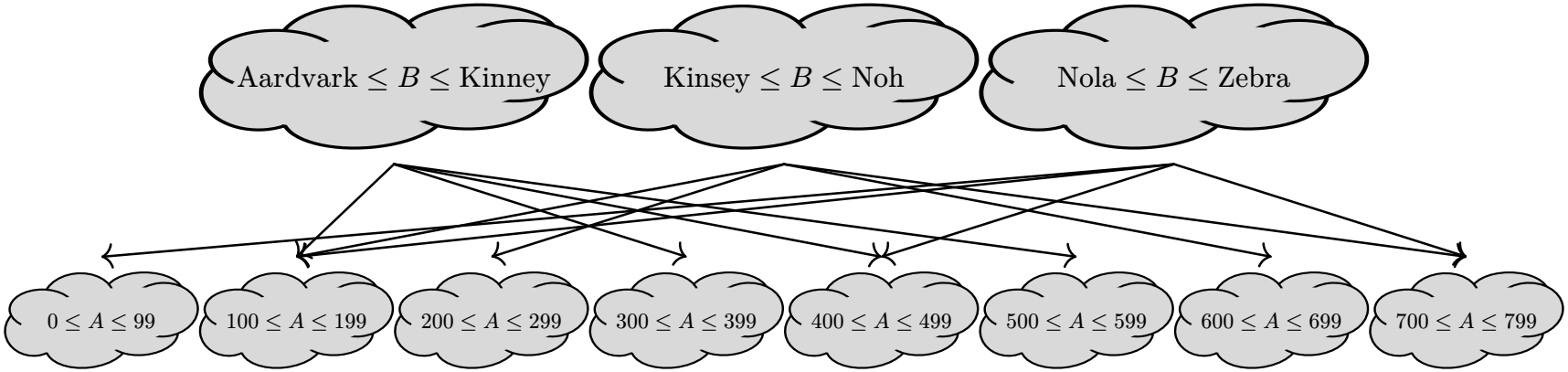
and

```
SELECT * FROM R WHERE R.B = 2
```

Unclustered Index

Idea: Leaves point to pages (and optionally record IDs) holding matching records.

Unclustered Index



Clustering

- **Clustered Index**: Data layout is fully correlated with index attribute
- **Unclustered Index**: Data layout is not fully correlated with index attribute
  - Often called a “Secondary” index

Density

- **Dense Index**: Index stores an entry for each record.
  - (key, page_id, slot_id)
- **Sparse Index**: Index stores references to ranges of records.
  - (lower_bound, upper_bound, page_id)

How about … $\underset{\theta}{\bowtie} S$?

More interesting joins

$$R \underset{R.B > S.B}{\bowtie} S$$

$$R \underset{S.\text{low} < R.B < S.\text{high}}{\bowtie} S$$

1 Pass Hash Join

Goal: Compute $R \underset{R.B=S.B}{\bowtie} S$

1. Build a hash table over S
2. For each row $r \in R$ use the hash table to find rows $s \in S$ where $r.B = s.B$

1 Pass [Index] Join

Goal: Compute $R \underset{R.B=S.B}{\bowtie} S$

1. Build a ~~hash table~~ *index* over S
2. For each row $r \in R$ use the ~~hash table~~ *index* to find rows $s \in S$ where $r.B = s.B$

1 Pass [Index] Join

Goal: Compute $R \underset{R.B>S.B}{\bowtie} S$

1. Build an *index* over S
2. For each row $r \in R$ use the *index* to find rows $s \in S$ where $r.B > s.B$

Index Nested Loop Join

Goal: Compute $R \underset{R.B>S.B}{\bowtie} S$

1. ~~Build an *index* over~~ S (use an index that already exists)
2. For each row $r \in R$ use the *index* to find rows $s \in S$ where $r.B > s.B$

Index Nested Loop Join

Goal: Compute $R \underset{\theta}{\bowtie} S$

```
for r in R:
  for s in IndexScan(S, θ[r]):
    yield r + s
```

Reminders

- Checkpoint 2 posted; Autolab up
- Checkpoint 1 solutions posted
- Quiz 1, 2 results posted