Indexes

Database Systems: The Complete Book
Ch. 13.1-13.3, 14.1-14.2

Hash-Based Indexes

What’s a Hash Function?

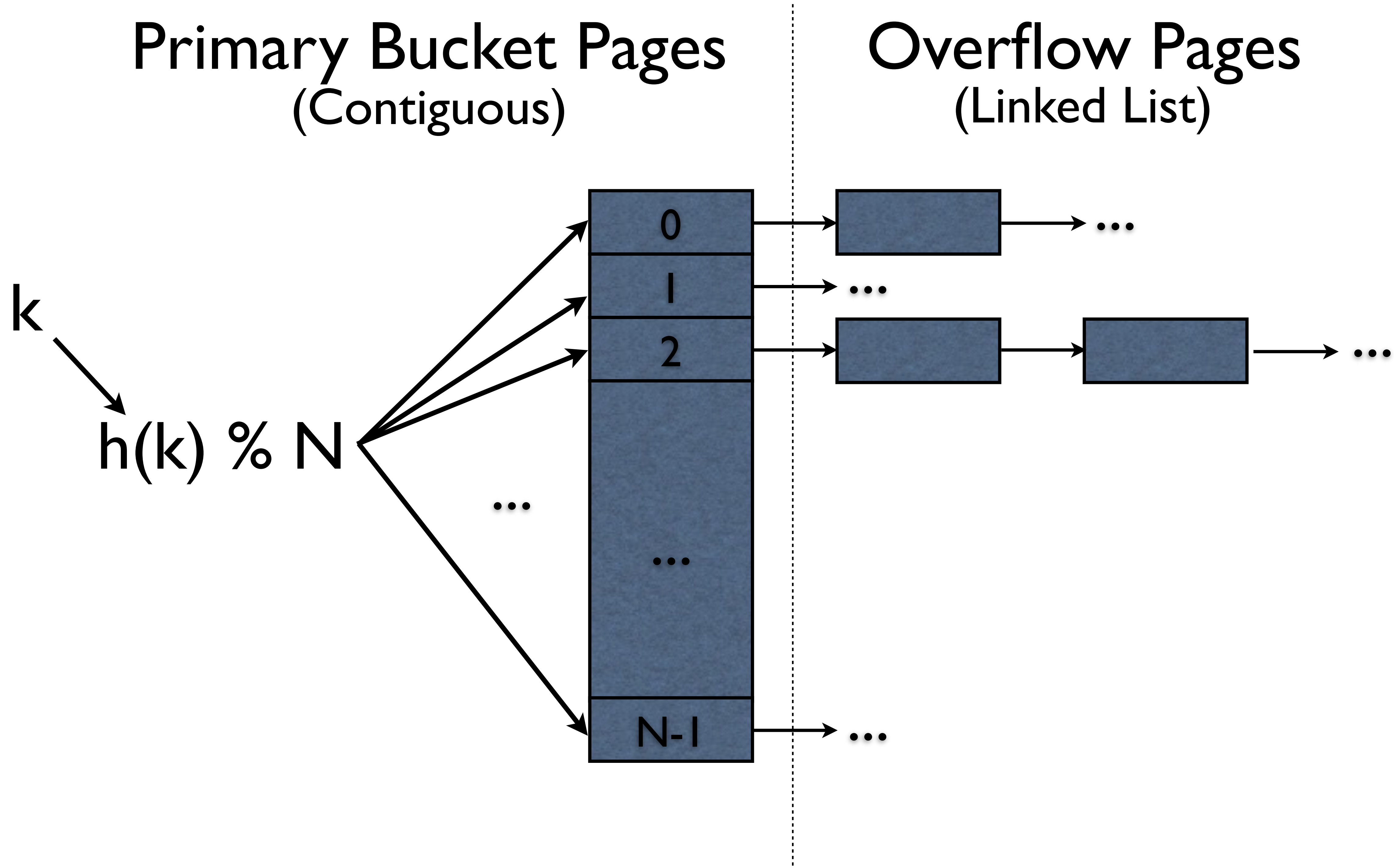
Hash Functions

- A hash function is a function that maps a large data value to a small fixed-size value
  - Typically is deterministic & pseudorandom
- Used in Checksums, Hash Tables, Partitioning, Bloom Filters, Caching, Cryptography, Password Storage, …
- Examples: MD5, SHA1, SHA2
  - MD5() part of OpenSSL (on most OSX / Linux / Unix)
- Can map h(k) to range [0,N) with h(k) % N (modulus)

Hash-based Indexes

- As with trees: request a key k and get record(s) or record id(s) with k.
- Hash-based indexes support equality lookups
  - … in constant time (vs log(n) for tree)
  - … but don't support range lookups
- Static vs Dynamic Hashing
  - Tradeoffs similar to ISAM vs B+Tree

Static Hashing



Static Hashing

- Buckets contain data entries.
- Hash fn maps the search key field of records to one of a finite number of buckets (% N)
- N chosen when the index is created
  - Too small N: Long overflow chains
  - Too big N: Wasted space/Poor IO

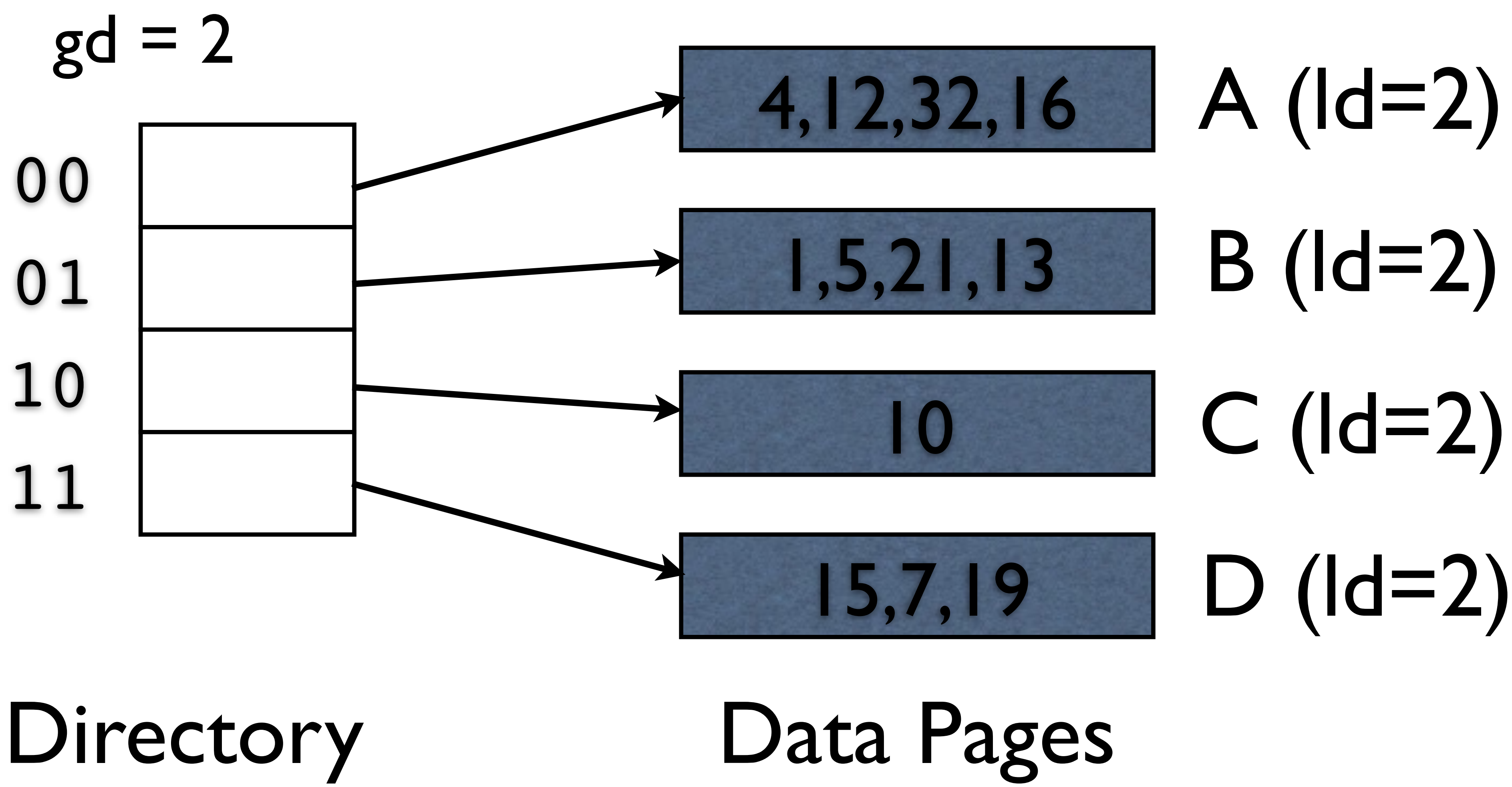
What's to stop us from "just resizing the hashmap?"

Dynamic Solutions: Extendible and Linear Hashing

Extendible Hashing

- **Situation:** A bucket becomes full
  - Solution: Double the number of buckets!
  - Expensive! (N reads, 2N writes)
- **Idea:** Add one level of indirection
  - A directory of pointers to (noncontiguous) bucket pages.
  - Doubling just the directory is much cheaper.
  - Can we double only the directory?

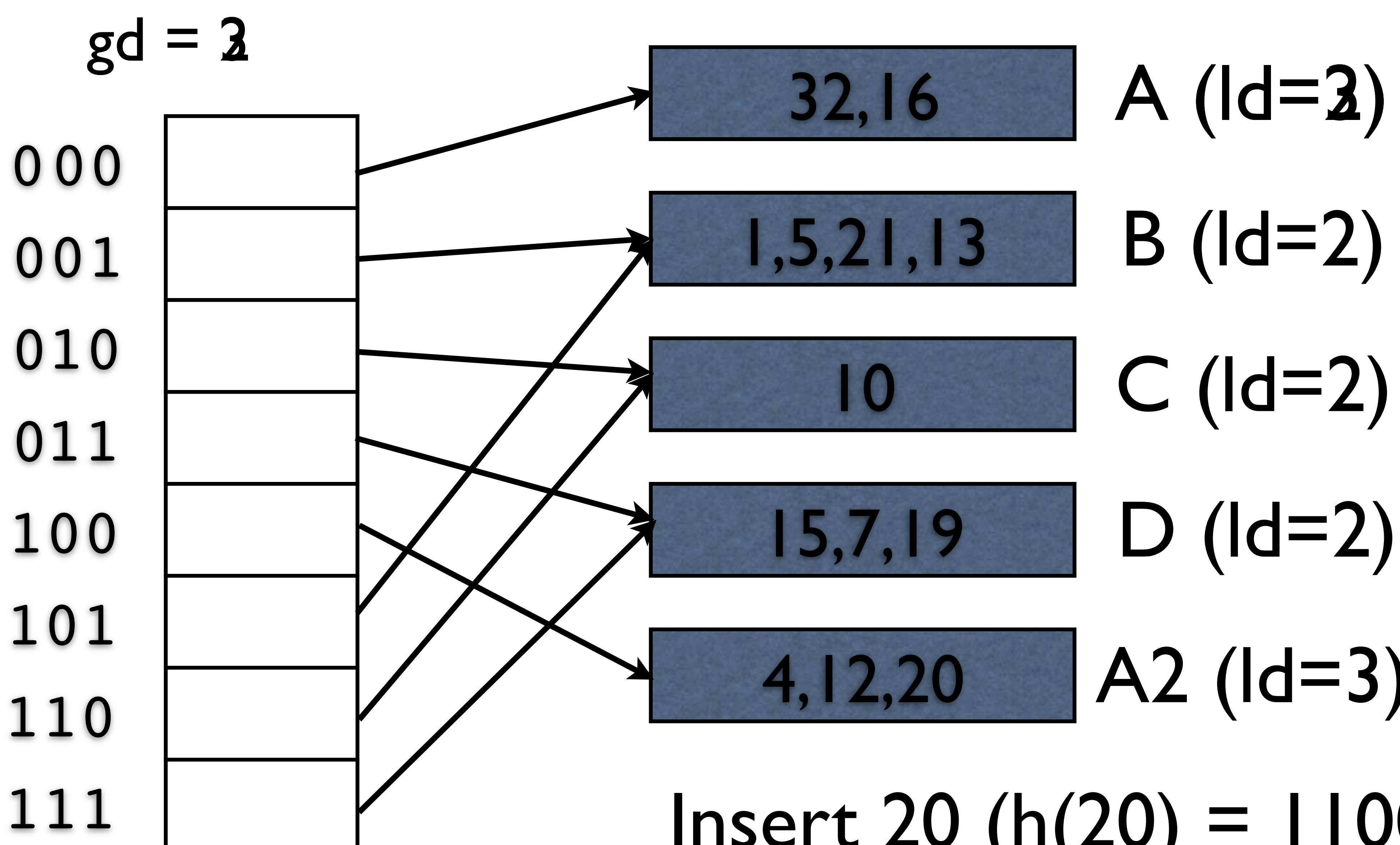
Extendible Hashing



The directory and data pages have an associated “depth” (global/local)

To look up a value use the last **gd** bits of the key’s hash value as an index into the dir

Extendible Hashing

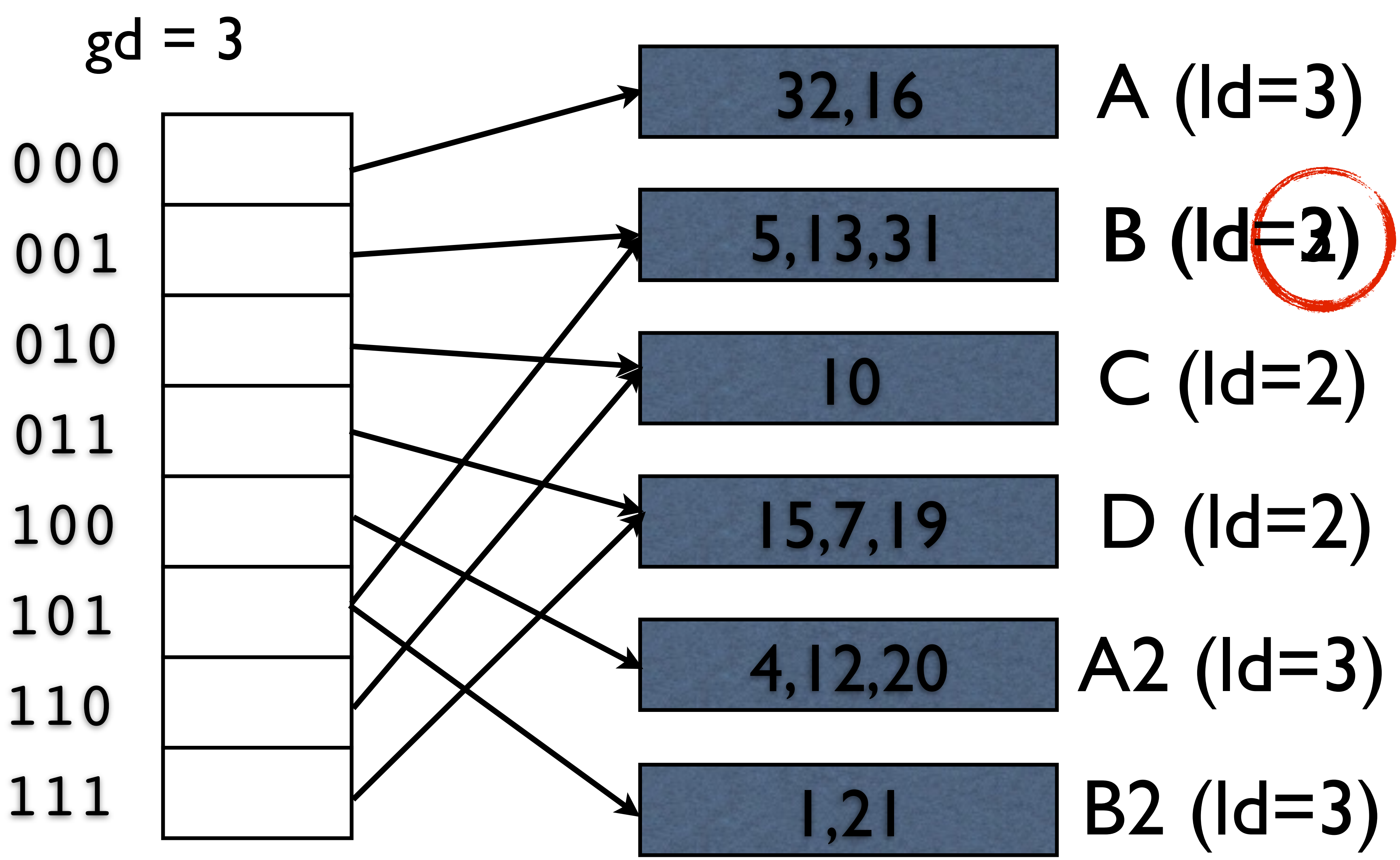


Dir entries not being split point to the same bucket

Insert 20 (h(20) = 1100)

(Need to Split Bucket A)

Extendible Hashing



Don't need to double dir
when splitting bucket w/ ld < gd

Insert 31 (h(31) = 1001)
(Need to Split Bucket B)

Extendible Hashing

- Global depth of directory
  - **Upper bound** on # of bits required to determine the bucket of an entry.
- Local depth of a bucket
  - **Exact** # of bits required to determine if an entry belongs in this bucket.
- Why use least significant bits (vs MSB)?

Extendible Hashing

- If the entire directory fits in memory, any equality search can be answered in one disk access. (otherwise two)
  - Is this true even if the directory spans multiple pages?
  - 100 MB file, 100 B/rec = 1m records over 4k pages.
    - Minimum of 25k directory entries.
    - Hash table still likely to be < 1M

Extendible Hashing

- Hashing Issues:
  - Need a uniform distribution of hash values.
    - Even a true random function will not provide this
  - What could happen if multiple keys have the same hash value? (A hash ‘collision’)
- Deletions
  - Deleting the last entry in a bucket allows it to be merged with its ‘split image’.
  - Can potentially halve directory if this happens.

Breaking Up Conditions

Boolean formulas can create complex conditions

```
(Officer.Ship = '1701A'
   AND Officer.Rank > 2)
        OR Officer.Rank > 3
```

Breaking Up Conditions

Boolean formulas can create complex conditions

```
(Officer.Ship = '1701A'
   AND Officer.Rank > 2)
      OR Officer.Rank > 3
```

First convert all conditions to <u>Conjunctive Normal Form</u> (CNF)

```
   (Officer.Ship = '1701A'
         OR Officer.Rank > 3)
AND(Officer.Rank > 2
         OR Officer.Rank > 3)
```

Breaking Up Conditions

Boolean formulas can create complex conditions

```
(Officer.Ship = '1701A'
   AND Officer.Rank > 2)
        OR Officer.Rank > 3
```

First convert all conditions to <u>Conjunctive Normal Form</u> (CNF)

```
(Officer.Ship = '1701A'
        OR Officer.Rank > 3)
            AND Officer.Rank > 2
```

Simplification may be possible

Indexing

- Indexes are typically built over one (key) field k
- Index stores mappings from key k to :
    - k → The full tuple with key value k — Clustered
    - k → Record ID for Tuple with key value k — Unclustered
    - k → List of Record/RecordIDs with key value k — Unclustered
- The choice of data to store is orthogonal to the choice of how to map key to value.

Multi-Attribute Indexes

We can create an ordering on <A,B>:
<A1, B1> is less than <A2, B2>
whenever

- A1 is less than A2
- A1 = A2 and B1 is less than B2

Can we use this sort order to find all <A,B> where…

All A < 3?

All A = 3 and B = 2?

All A = 3 and B < 2?

All A < 3 and B = 2?

Access Paths and Join Algorithms

Database Systems: The Complete Book
Ch. 15.4-15.6

Example

```
SELECT COUNT(*)
FROM Students S,
     CourseRegs R
WHERE S.Name = ‘Alice’
  AND S.Id = R.StudentId
  AND R.Grade > 90
  AND R.Grade < 100
```

What is the Equivalent Relational Algebra Expression?

Example

```
SELECT COUNT(*)
FROM Students S,
     CourseRegs R
WHERE S.Name = ‘Alice’
  AND S.Id = R.StudentId
  AND R.Grade > 90
  AND R.Grade < 100
```

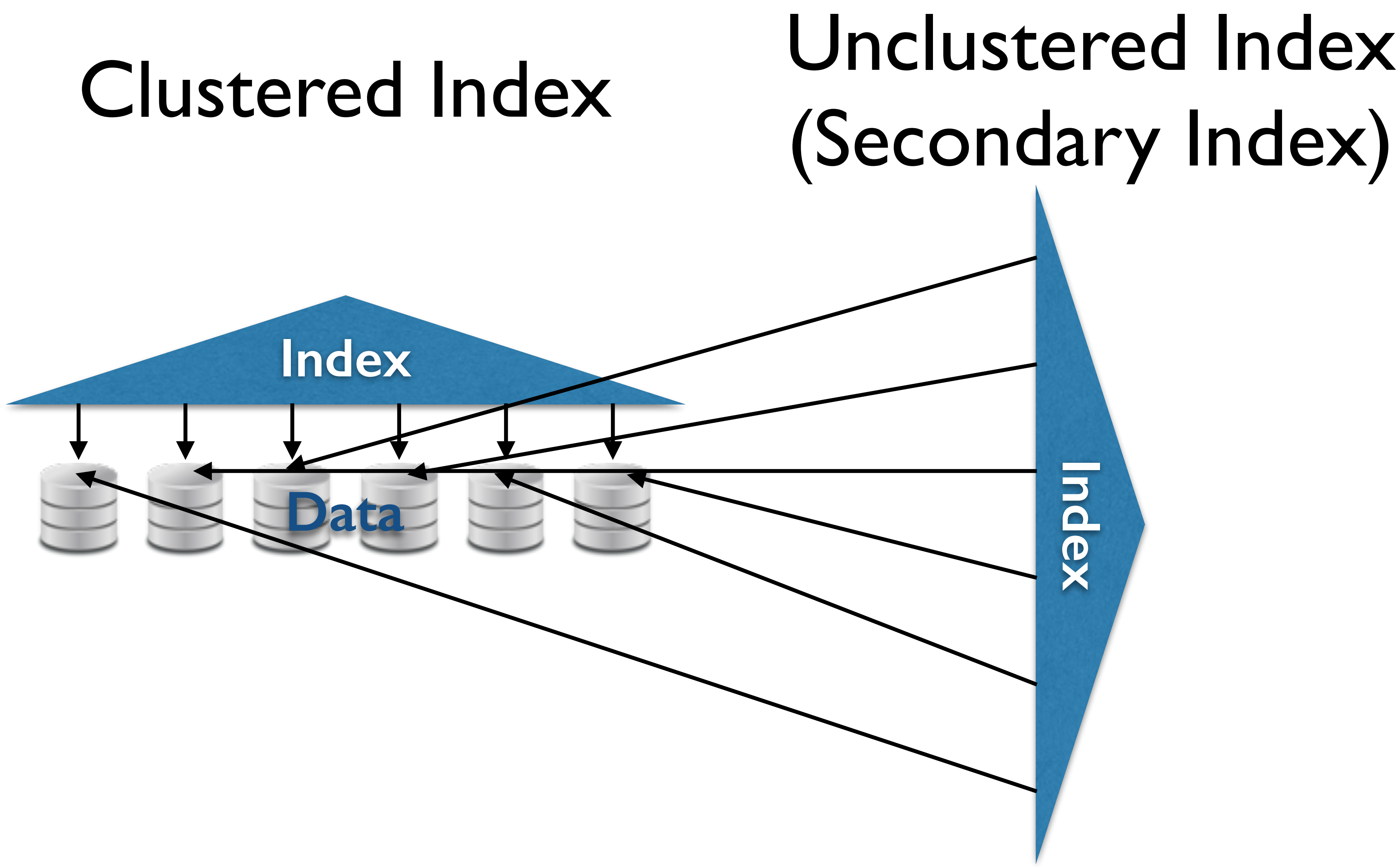
How Do We Optimize This Expression?

Example

What Indexes Might be Helpful?

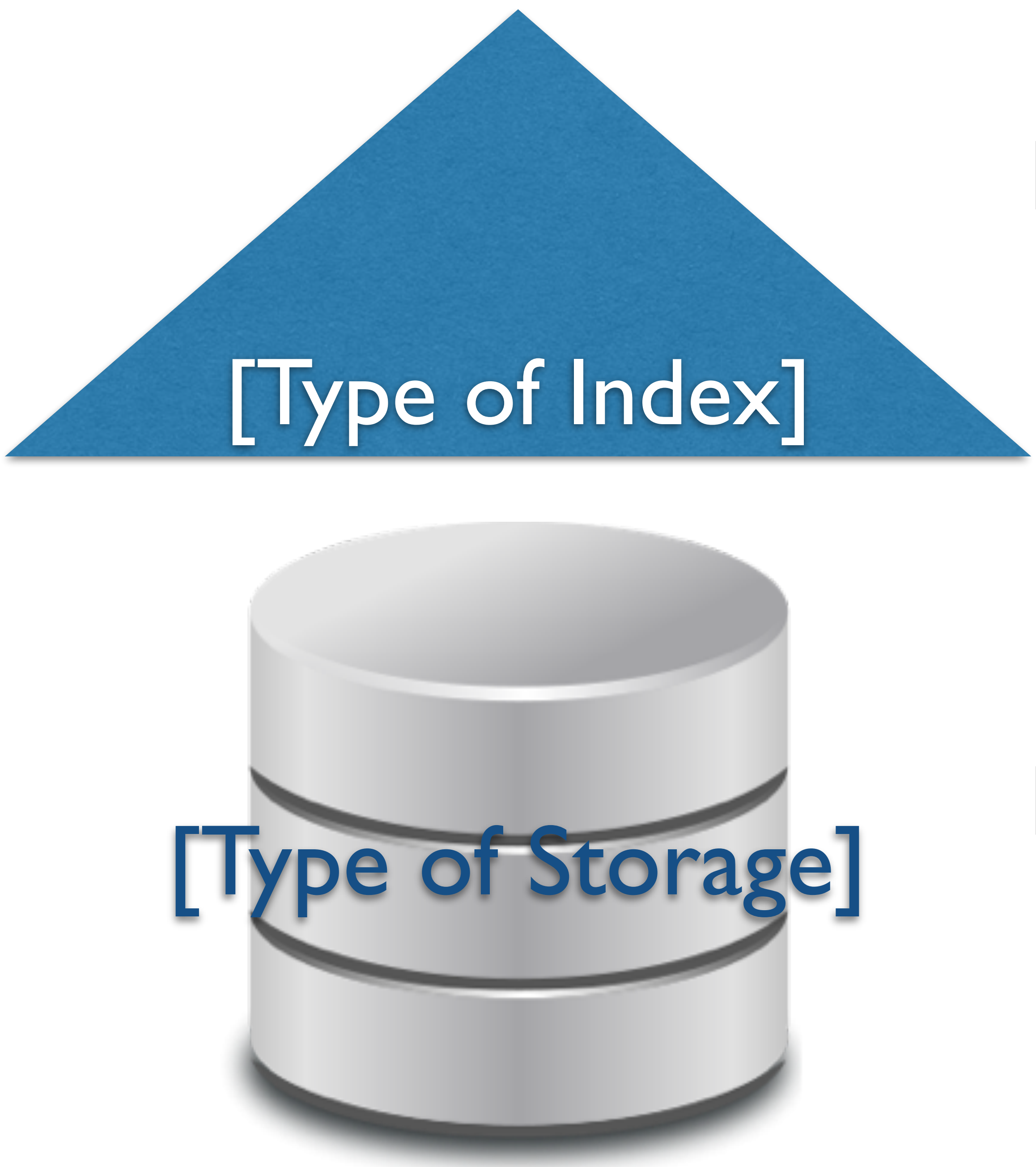
When?

Indexes



Clustered Index
Unclustered Index
(Secondary Index)
Index
Data
Index

Indexes



How the Data is Organized

ISAM
B+Tree
Other Tree-Based
Hash Table
Other Hash-Based
Other…

How the Data is Laid Out

<u>In the Index</u>
Clustered

<u>Outside of the Index</u>
Sorted
Heap

Multiple Indexes

Can we have multiple indexes over one table?

How does this affect our design considerations?

Access Paths

How do I read from the data

<u>Originally</u>

σ_c
|
R

"File Scan + Select"

<u>Now</u>

How do we pick?

IndexScan(R, c, Index#)

New Index Scan Operator

Joins

- Two General Classes of Joins
  - Equality (Equi-) Joins: `R.B = S.B`
  - Inequality (Inequi-) Joins: `R.B < S.B`

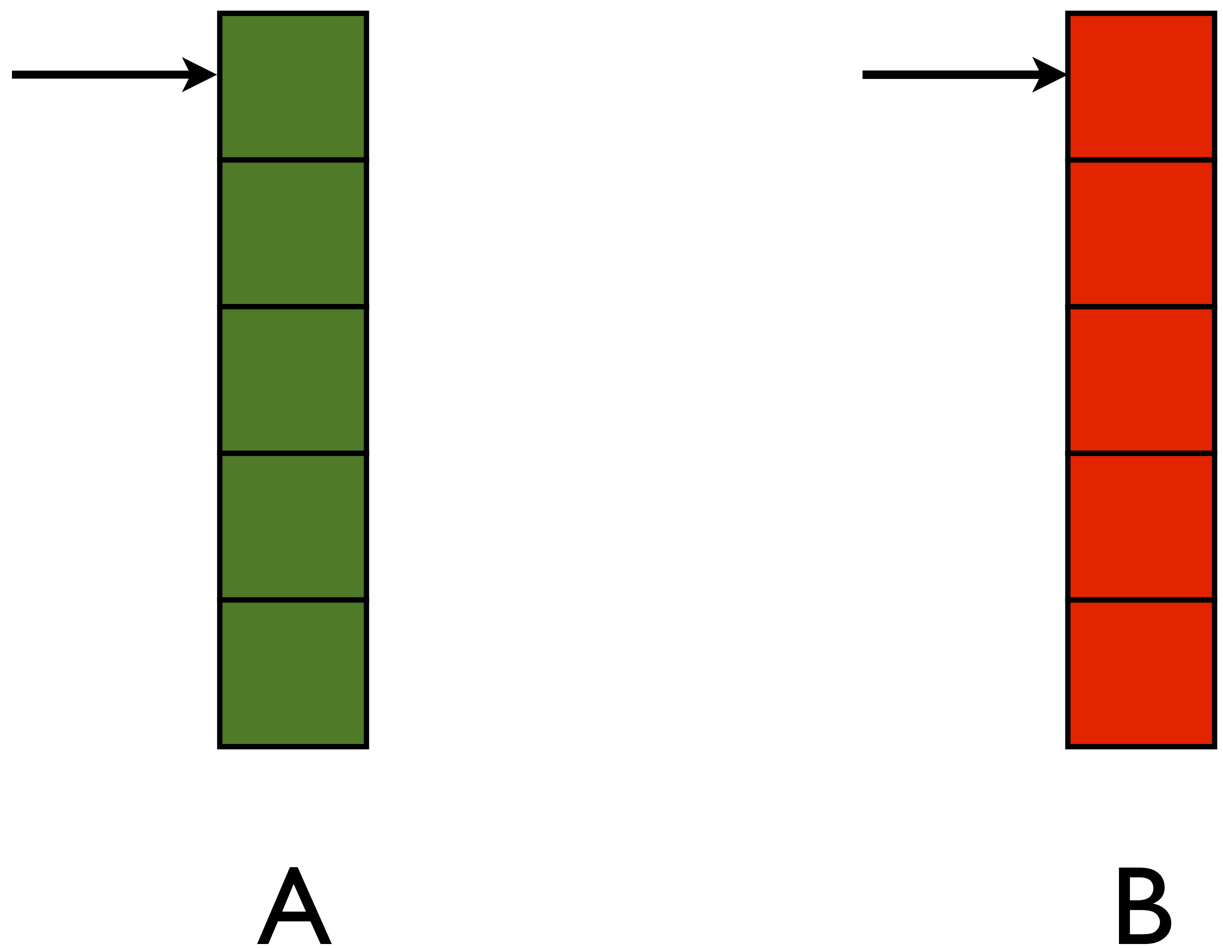
- How do the outputs of these joins differ?

Inequi-joins are $O(N^2)$ (as bad as NLJ)
We will focus on Equi-joins

Implementing: Joins

Solution 1 (Nested-Loop)

For Each (a in A) { For Each (b in B) { emit (a, b); }}



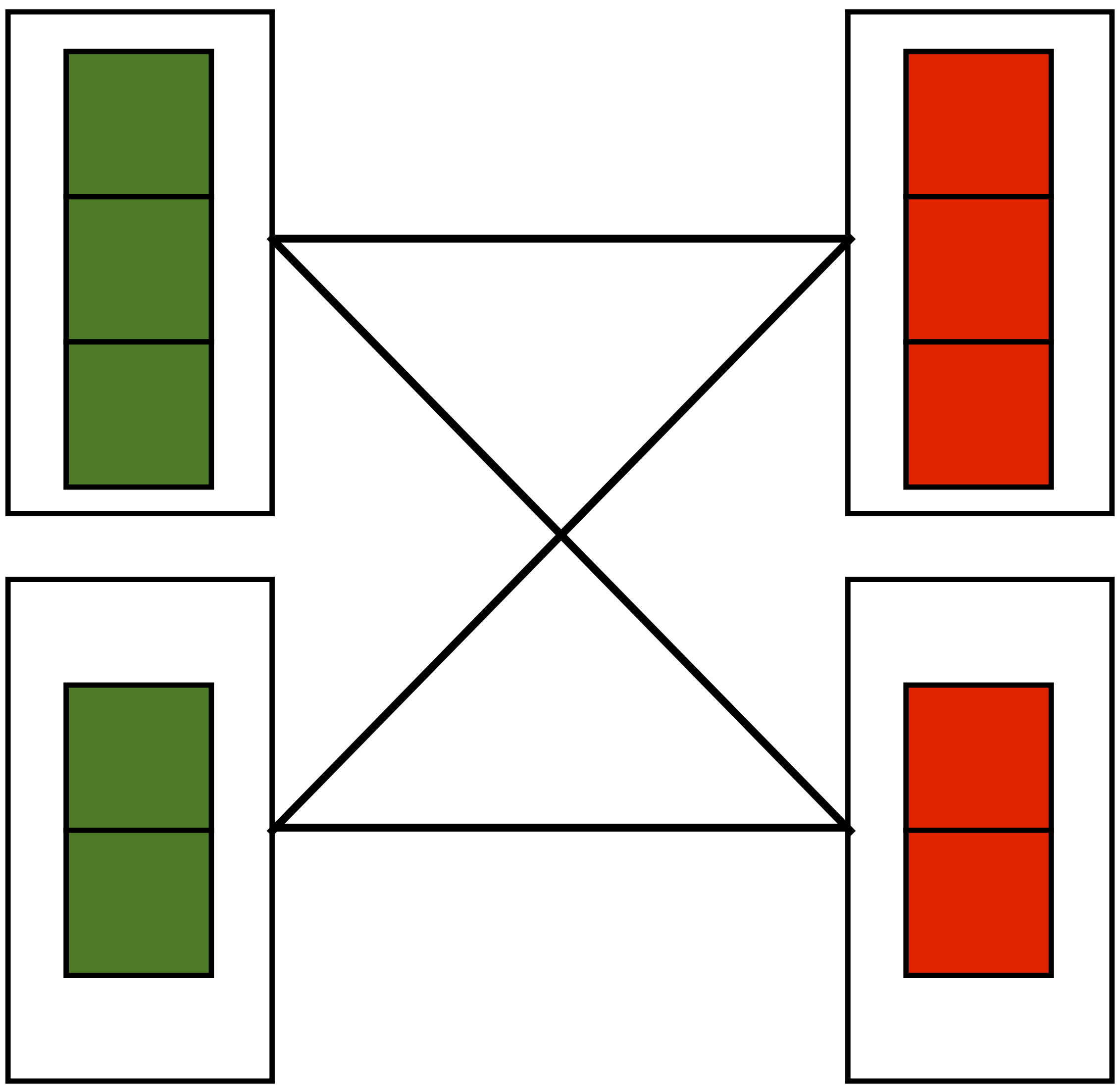
Implementing: Joins

Solution 2 (Block-Nested-Loop)

Implementing: Joins

Solution 2 (Block-Nested-Loop)

1) Partition into Blocks   2) NLJ on each pair of blocks

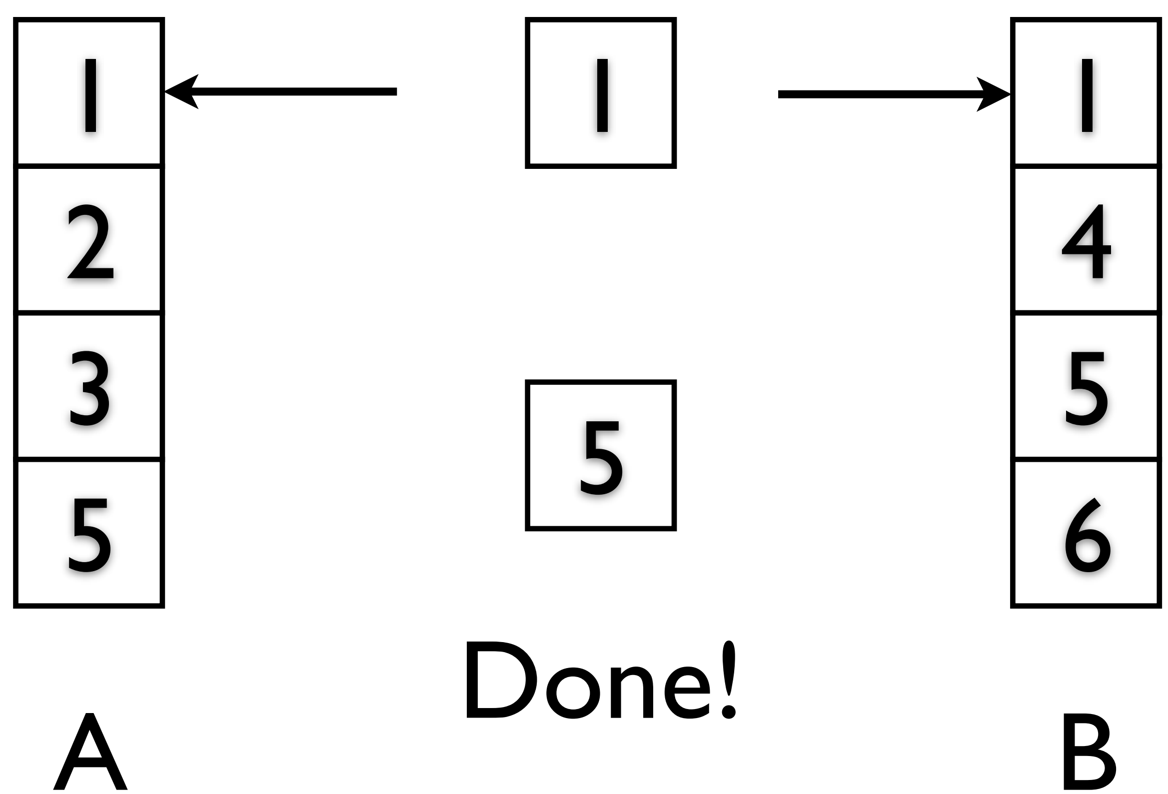


Implementing: Joins

Solution 3 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both



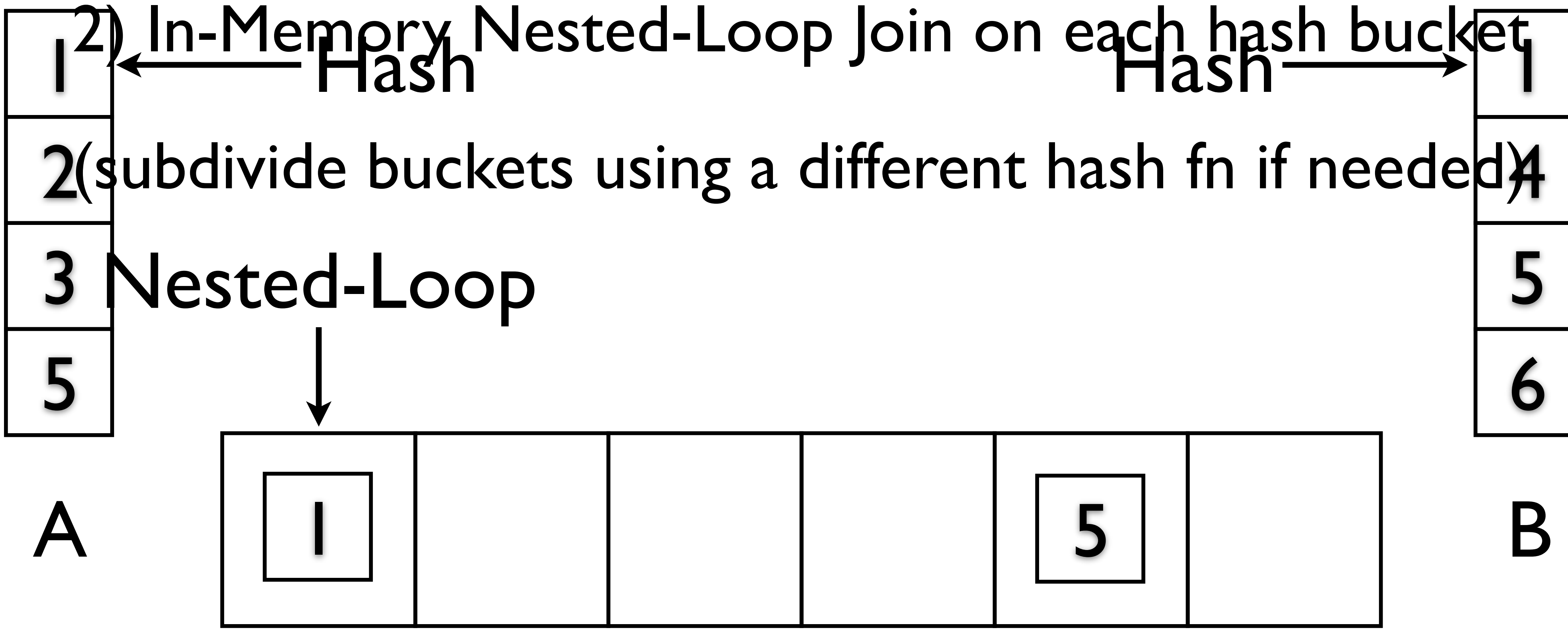
Implementing: Joins

Solution 4 (External Hash)

1) Build a hash table on both relations

2) In-Memory Nested-Loop Join on each hash bucket

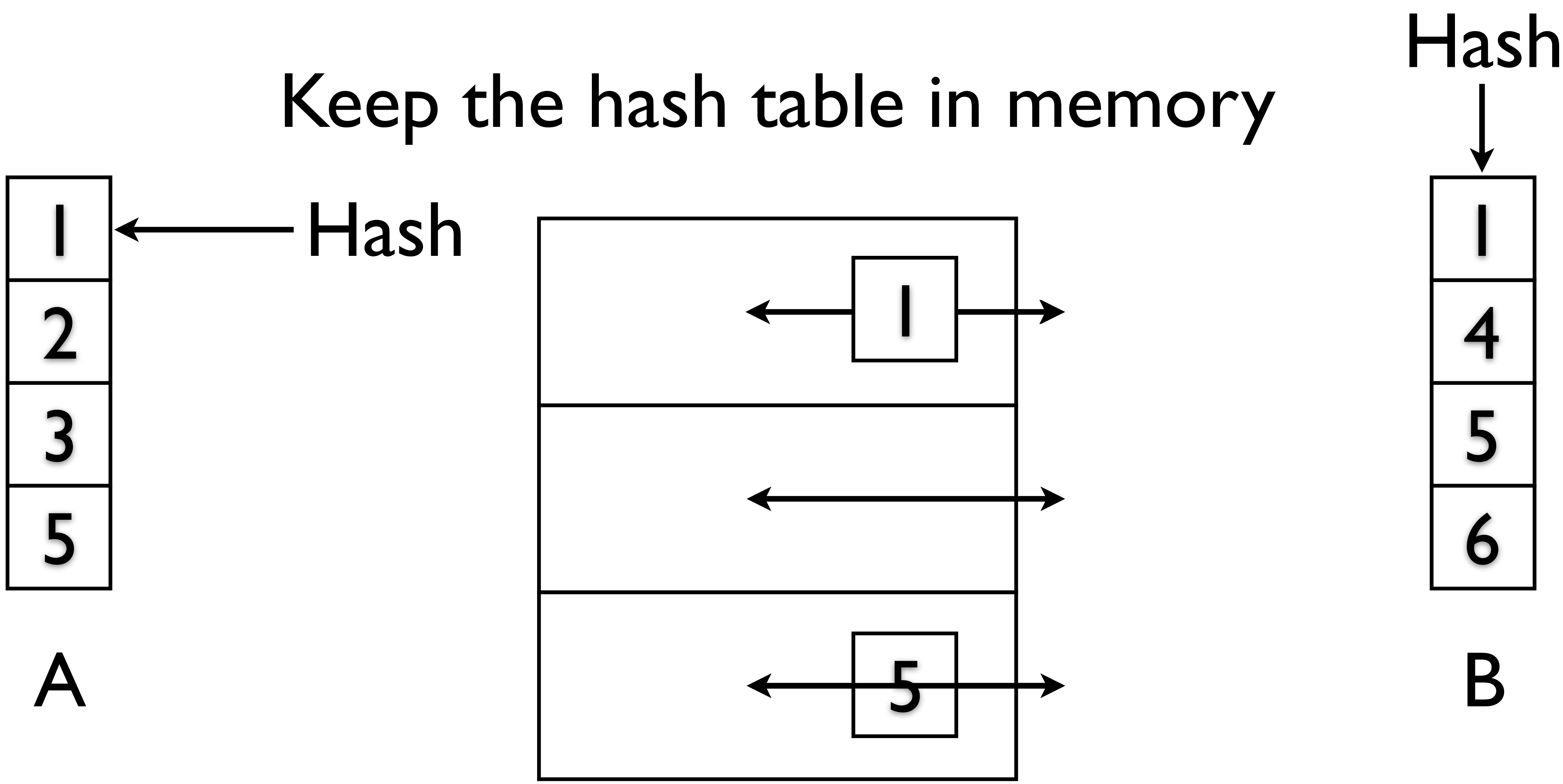
(subdivide buckets using a different hash fn if needed)



Implementing: Joins

Solution 5 (Grace/Hybrid Hash)

Keep the hash table in memory

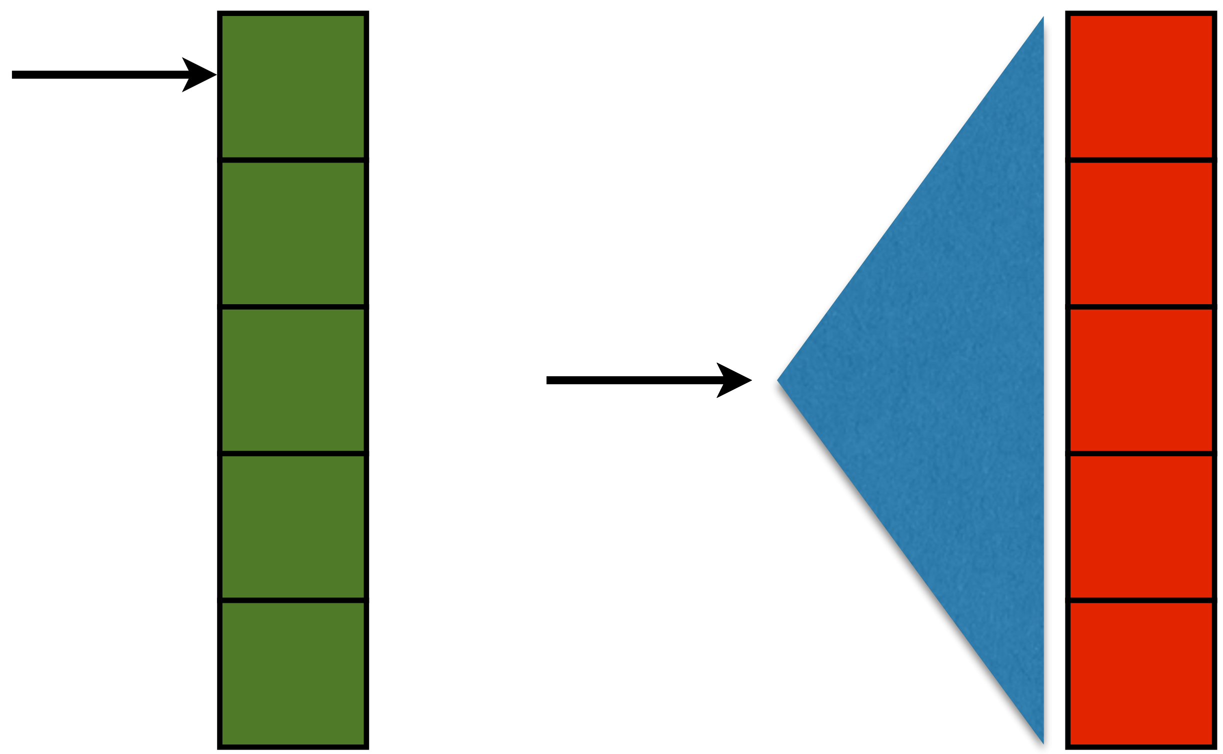


(Essentially a more efficient nested loop join)

Implementing: Joins

Solution 6 (Index-Nested-Loop)

Like nested-loop, but use an index to make the inner loop much faster!



What are the tradeoffs of each algorithm?

What properties do we care about?

How do the algorithms compare?

Implementing: Joins

Tradeoffs

	Pipelined?	Memory Requirements?	Predicate Limitation?
Nested Loop	1/2	1 Table	No
Block-Nested Loop	No	2 'Blocks'	No
Index-Nested Loop	1/2	1 Tuple (+Index)	Single Comparison
Sort-Merge	If Data Sorted	Same as reqs. of Sorting Inputs	Equality Only
Hash	No	Max of 1 Page per Bucket and All Pages in Any Bucket	Equality Only
Grace Hash	1/2	Hash Table	Equality Only

Extra Content - External Sort