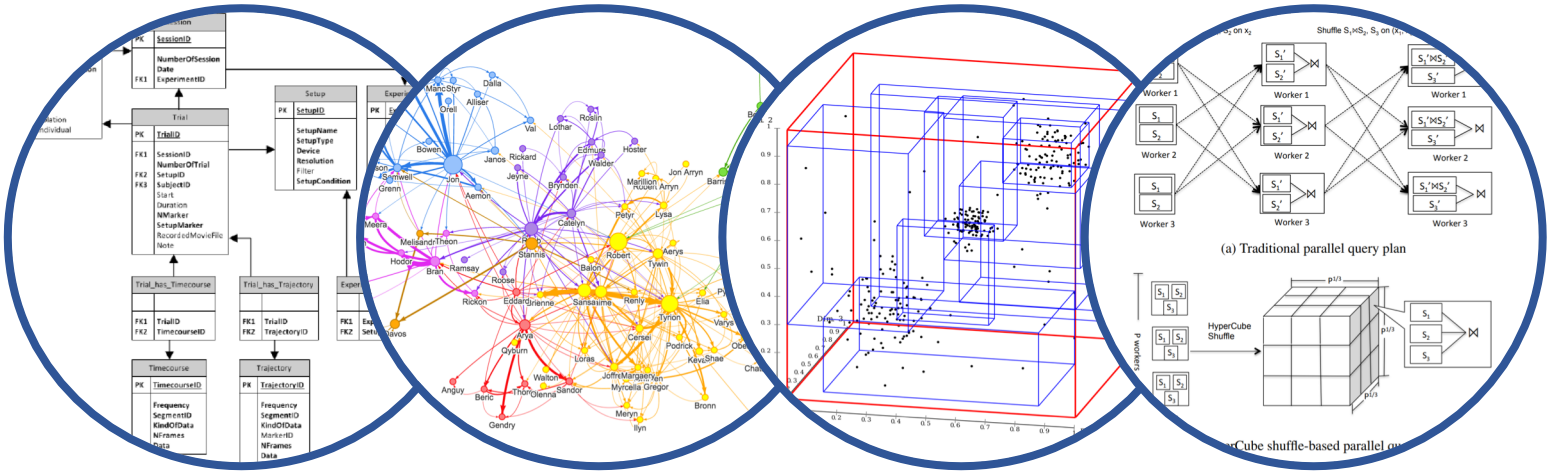


Introduction to Data Management

Database Tuning

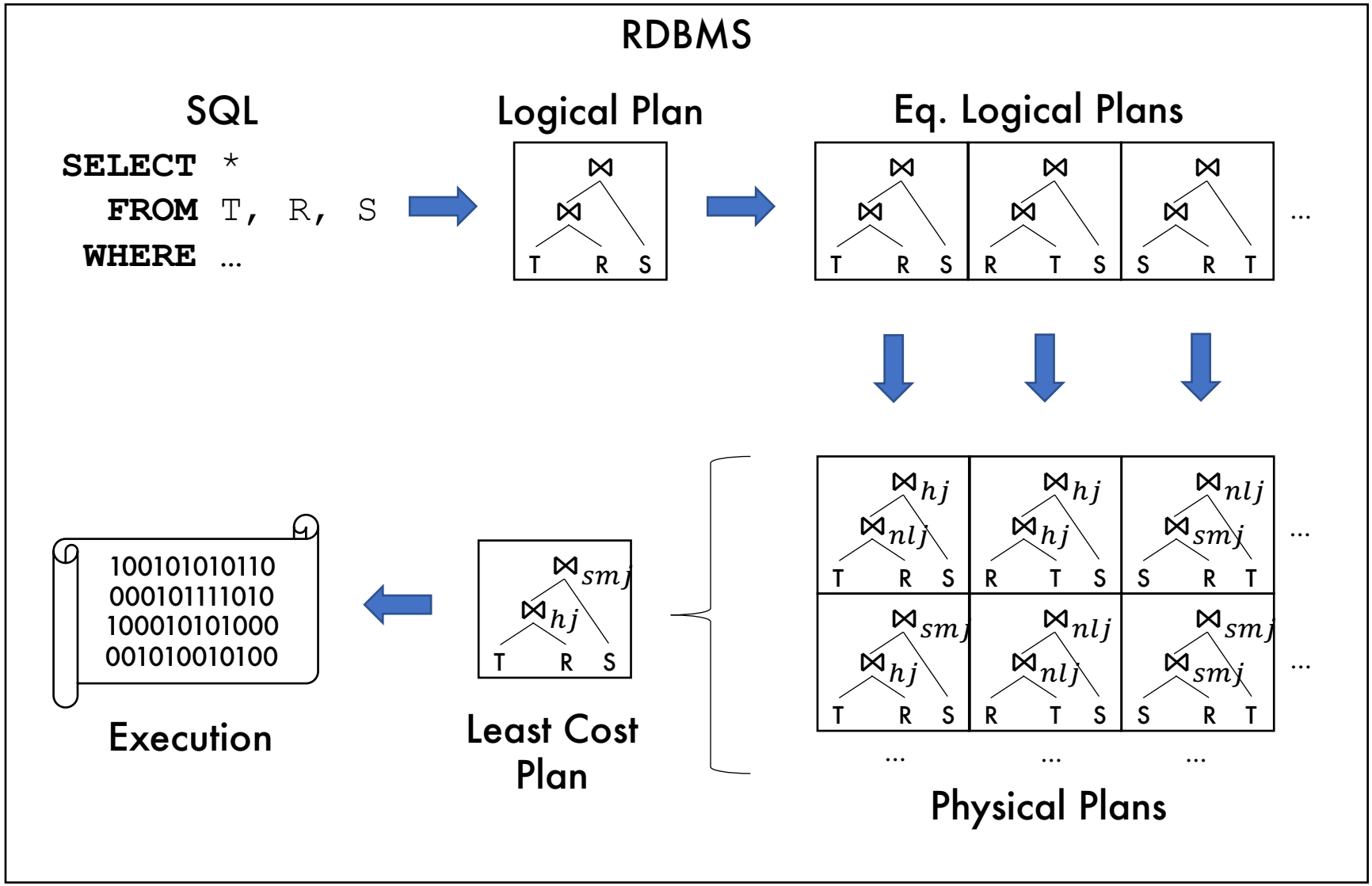
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Goals for Today

- We gave a baseline for what join algorithms (and respective costs) were possible
- Use DB structures to expand optimization options

- Key words to look out for:
  - "Clustered" index
  - "Unclustered" index

Recap – Plan Enumeration



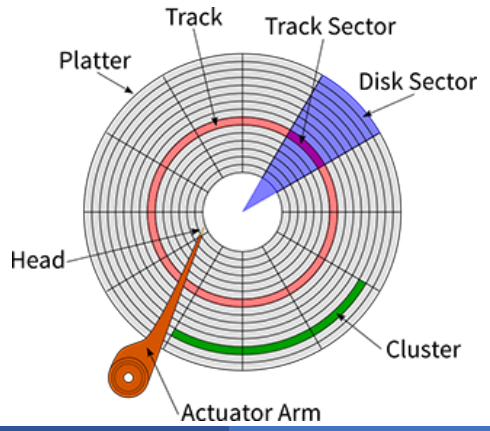
Recap – Assumptions

For this class we make a lot of assumptions

- **Disk-based storage**
  - HDD not SDD
- **Row-based storage**
  - Tuples are stored contiguously
- **IO cost** only
  - One disk access is ~100000x more expensive than one main memory access
- **Cold cache**
  - No data preloaded into main memory

Recap – Disk Storage

- Can only read **1 block per read operation**
  - Usually 512B to 4kB
- **Sequential disk reads are faster than random ones**
  - Cost ~1-2% random scan = full sequential scan



Recap – 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

Recap – Disk Storage

- Tables are stored as files
  - Heap file → Unsorted tuples
    - Nested-Loop Joins
      - Block-at-a-time Nested Loop Join (cost = B(R)+B(R)*B(S))
      - Block-Nested-Loop Join (cost = B(R)+B(R)/N*B(S))
    - Hash Join (B(R)<M, cost = B(R)+B(S))
    - Sort-Merge Join (B(R)+B(S)<M, cost = B(R)+B(S))
  - **Sequential file** → Sorted tuples

Outline

- Index structures
- Index join cost estimation
- Database tuning
- Multiple joins cost estimation
- Views

Indexing

- Indexes (for this class) can be assumed to be **already loaded into memory**
- An index does not have to contain all tuple data
  - Only key values are stored in the index
  - If an index contains all tuple data it is called a "covering index"
- **Indexes are access points for tables**

Index Structures

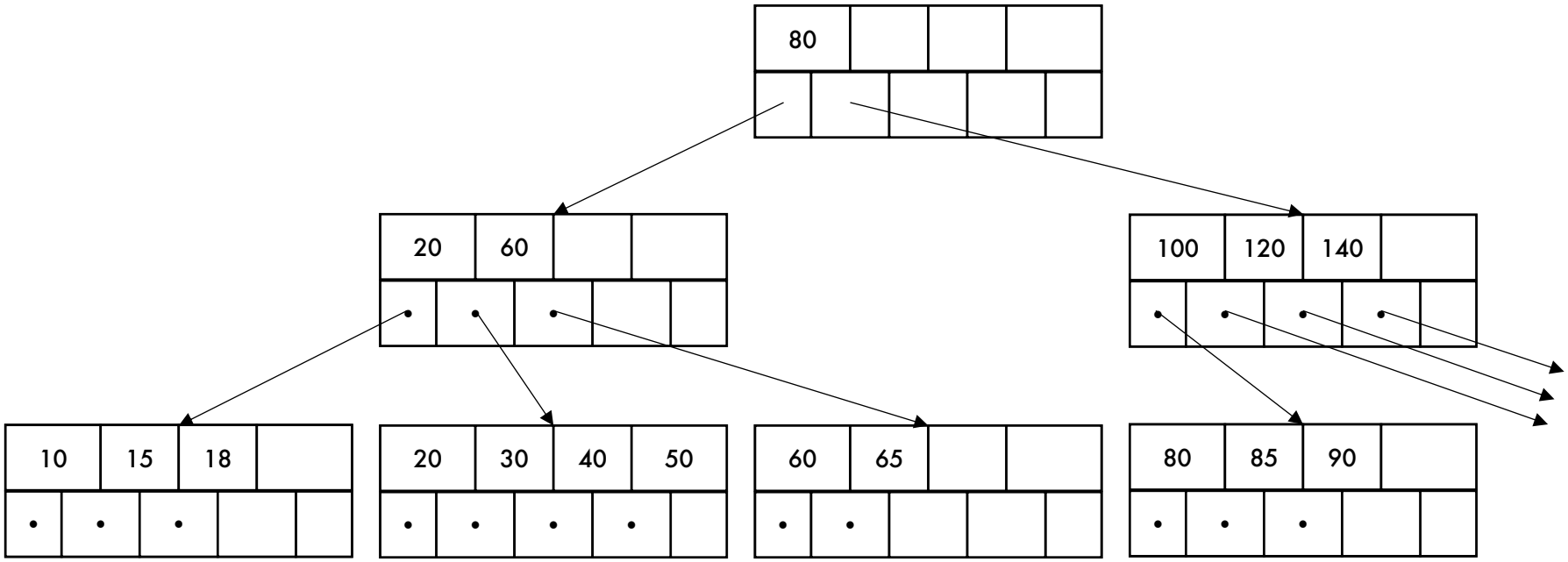
- **B+ Tree Index**
  - Clustered
  - Unclustered
- Hash Index
- R Tree
- Radix Tree
- Bloom Filter
- Hilbert Curves
- …

What is a B Tree?

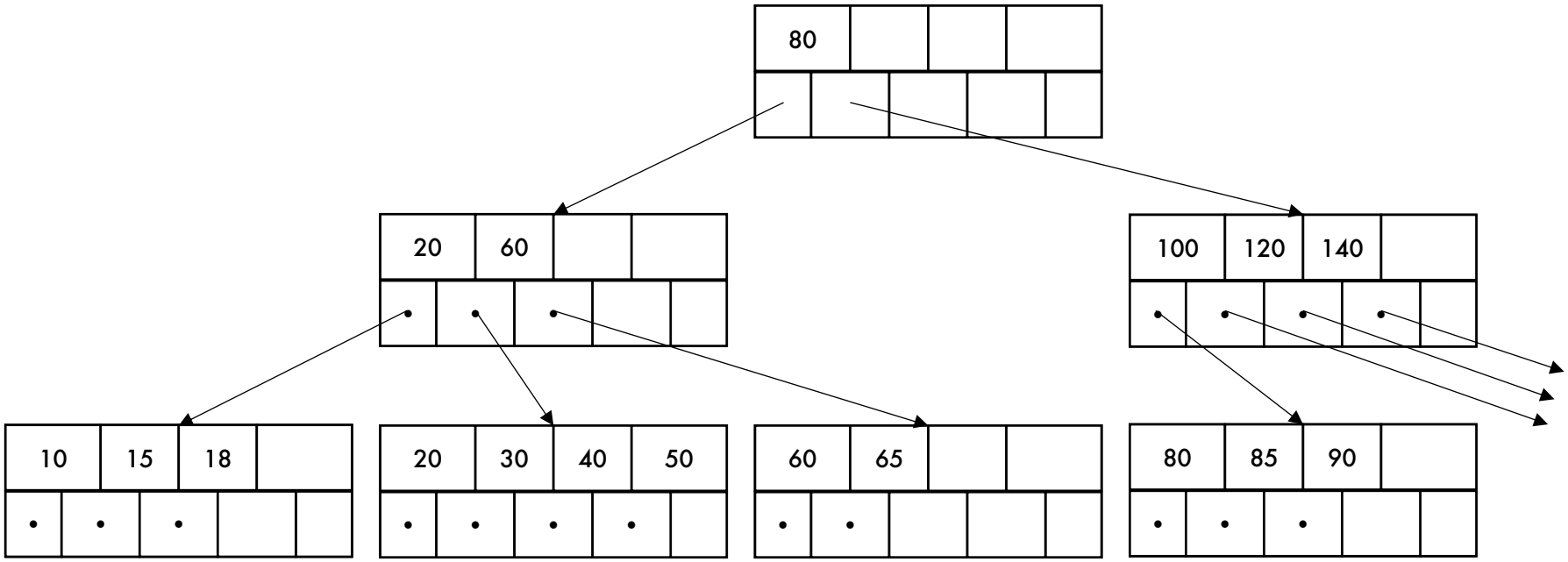
"What, if anything, the *B* stands for has never been established." – Wikipedia

- **Search tree** (like a binary search tree)
  - Nodes annotate max values
  - Large number of children per node
- Tree/node structure that is memory efficient

What is a B Tree?

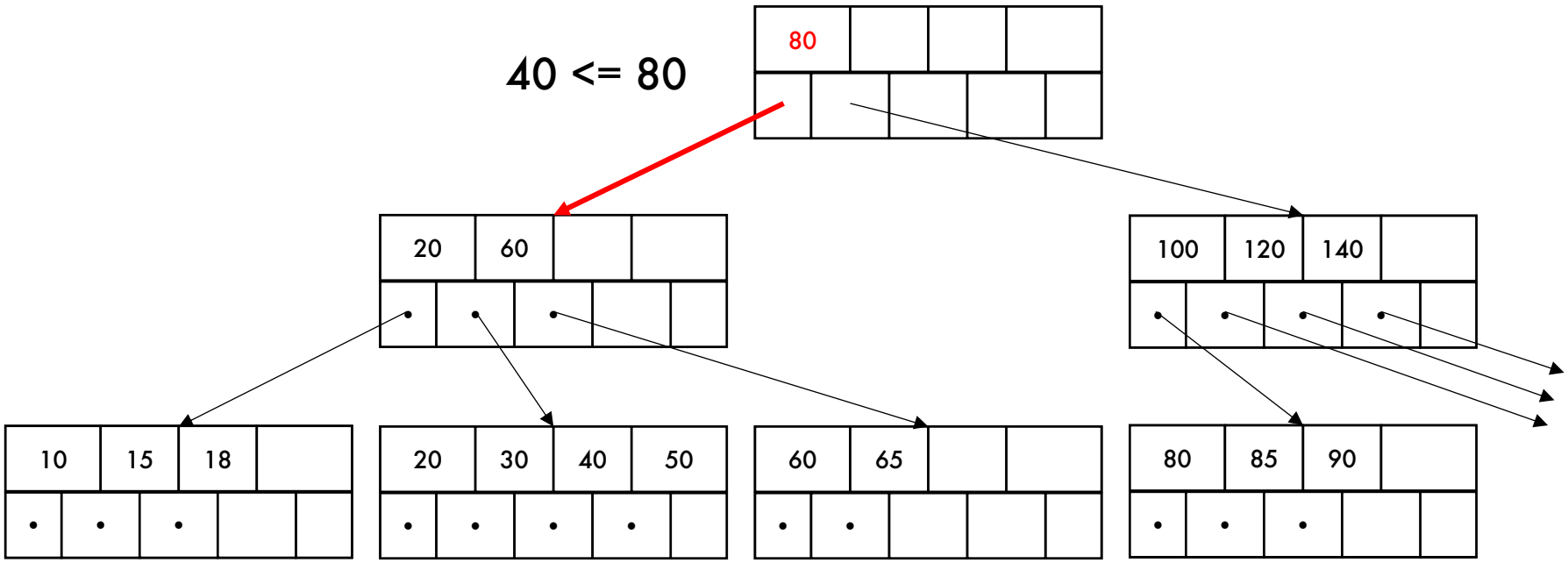


What is a B Tree?



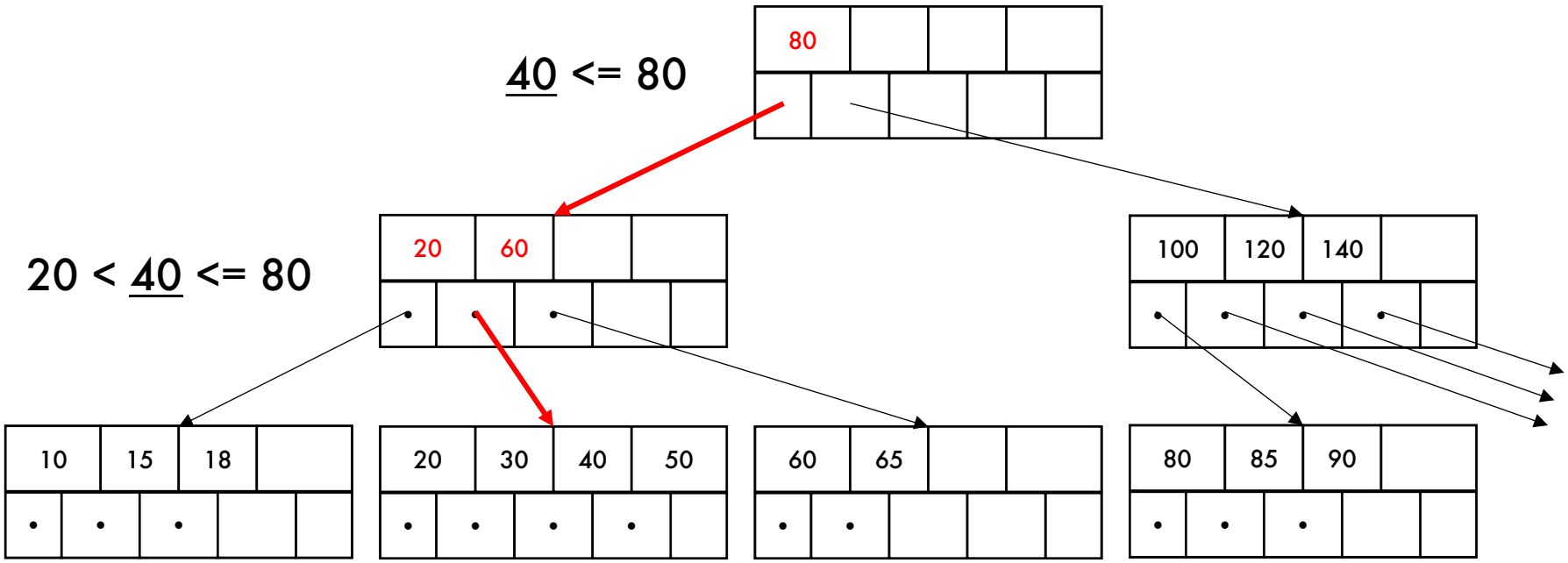
Find the value 40

What is a B Tree?



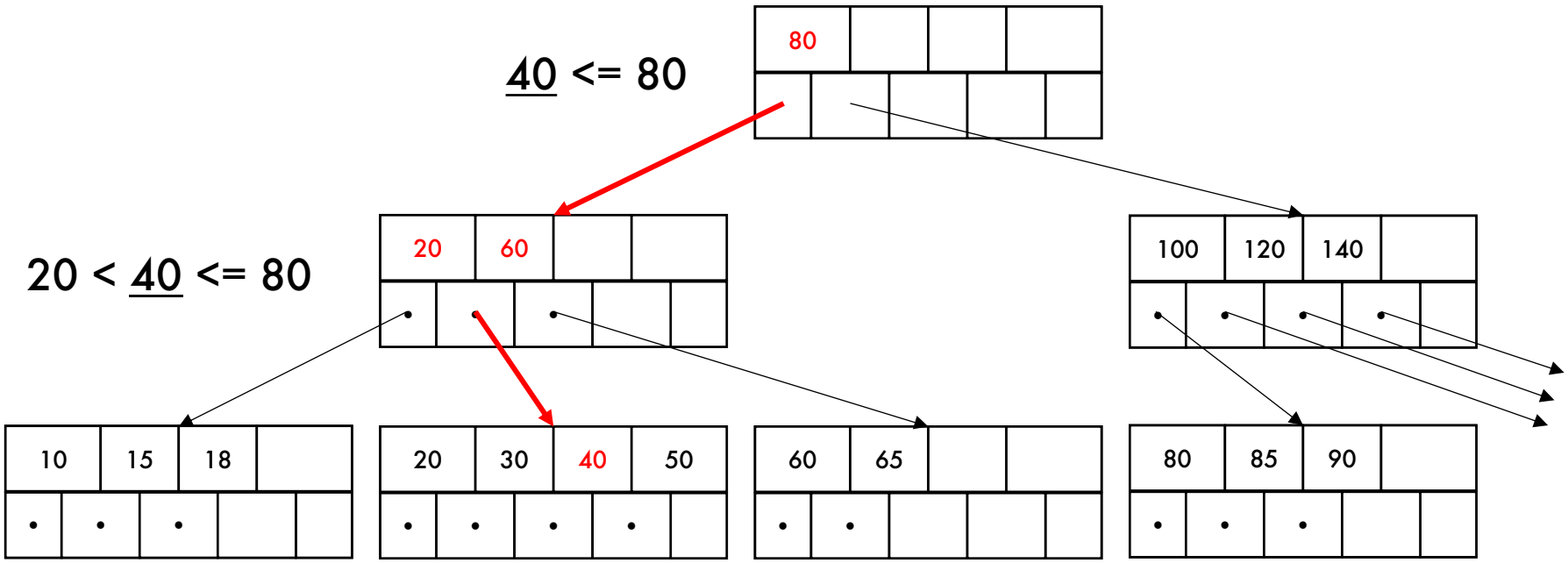
Find the value 40

What is a B Tree?



Find the value 40

What is a B Tree?

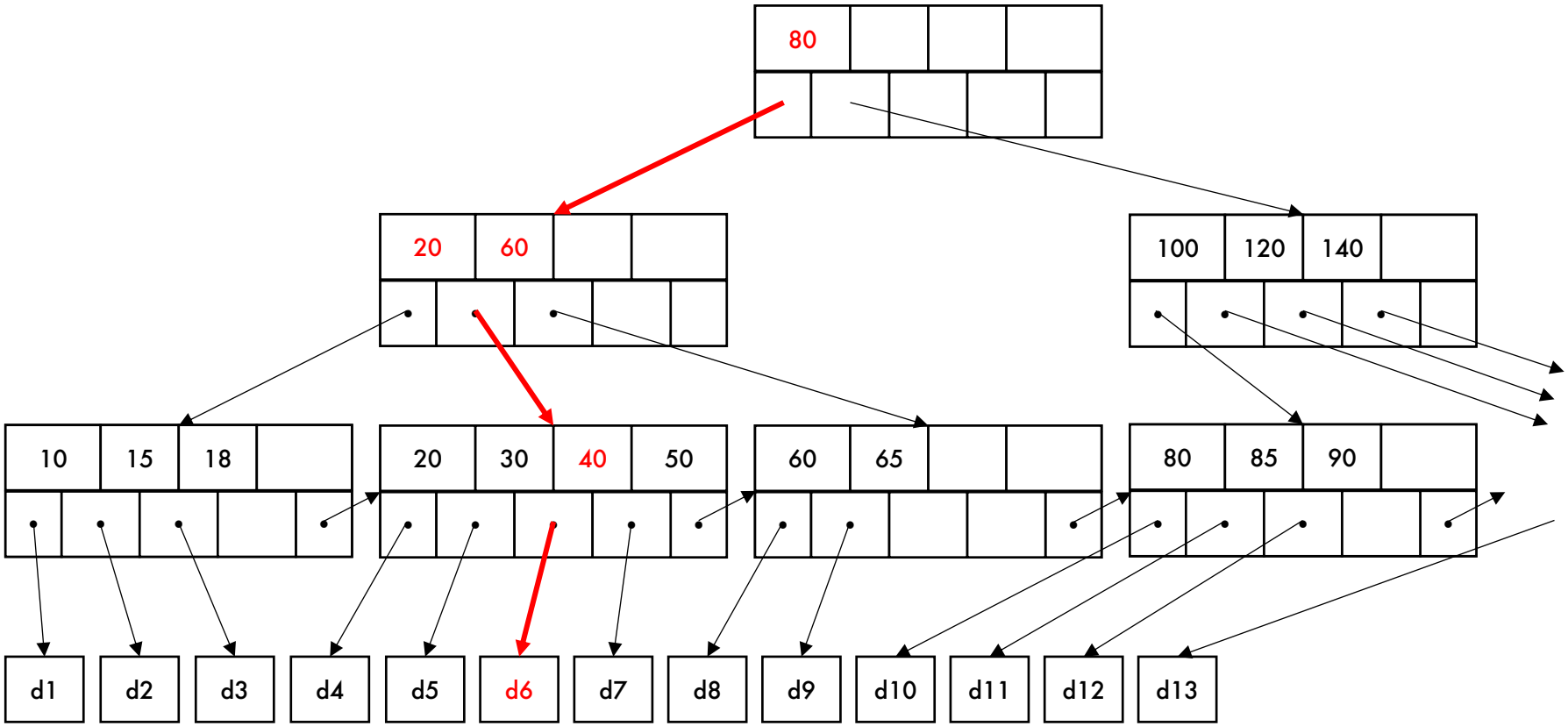


Find the value 40

How is a B+ Tree Different?

- Leaf nodes point to data
  - Data is searchable by **key** value annotated by the node labels
- Leaf nodes form a linked list

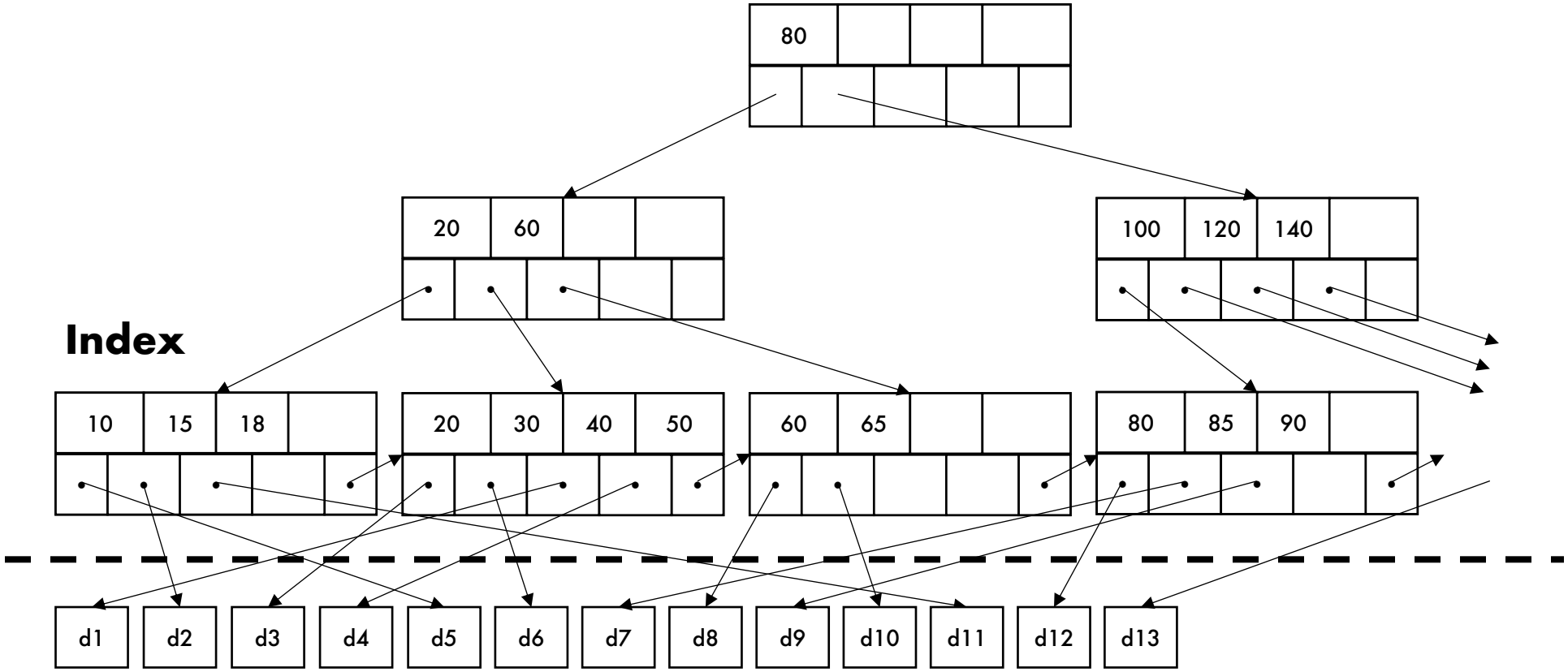
How is a B+ Tree Different?



Find the data associated with the key value 40
(same search process)

Clustered vs Unclustered Index

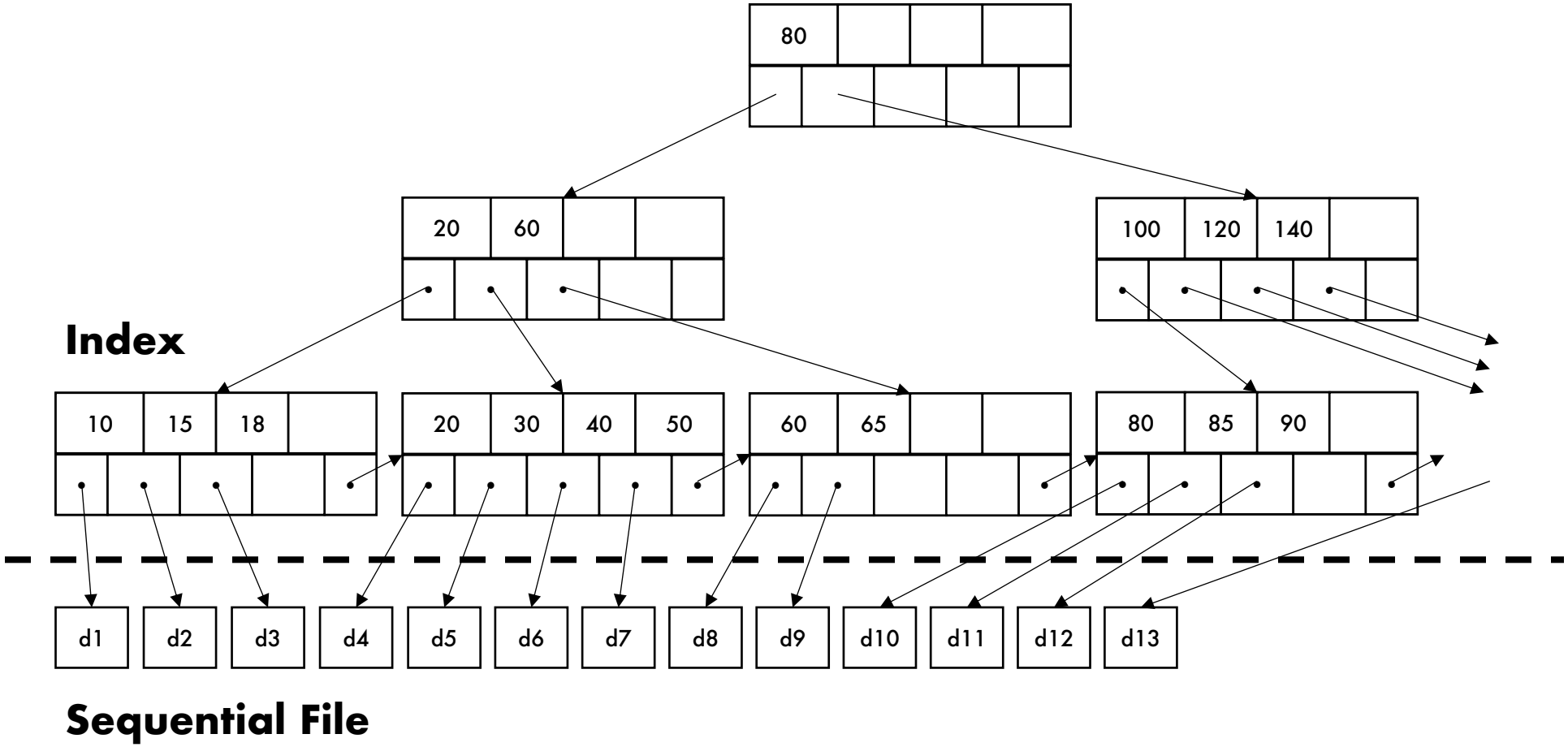
- An **unclustered index** may exist without any ordering on disk (any number per table)



Sequential File with a different key or **Heap File**

Clustered vs Unclustered Index

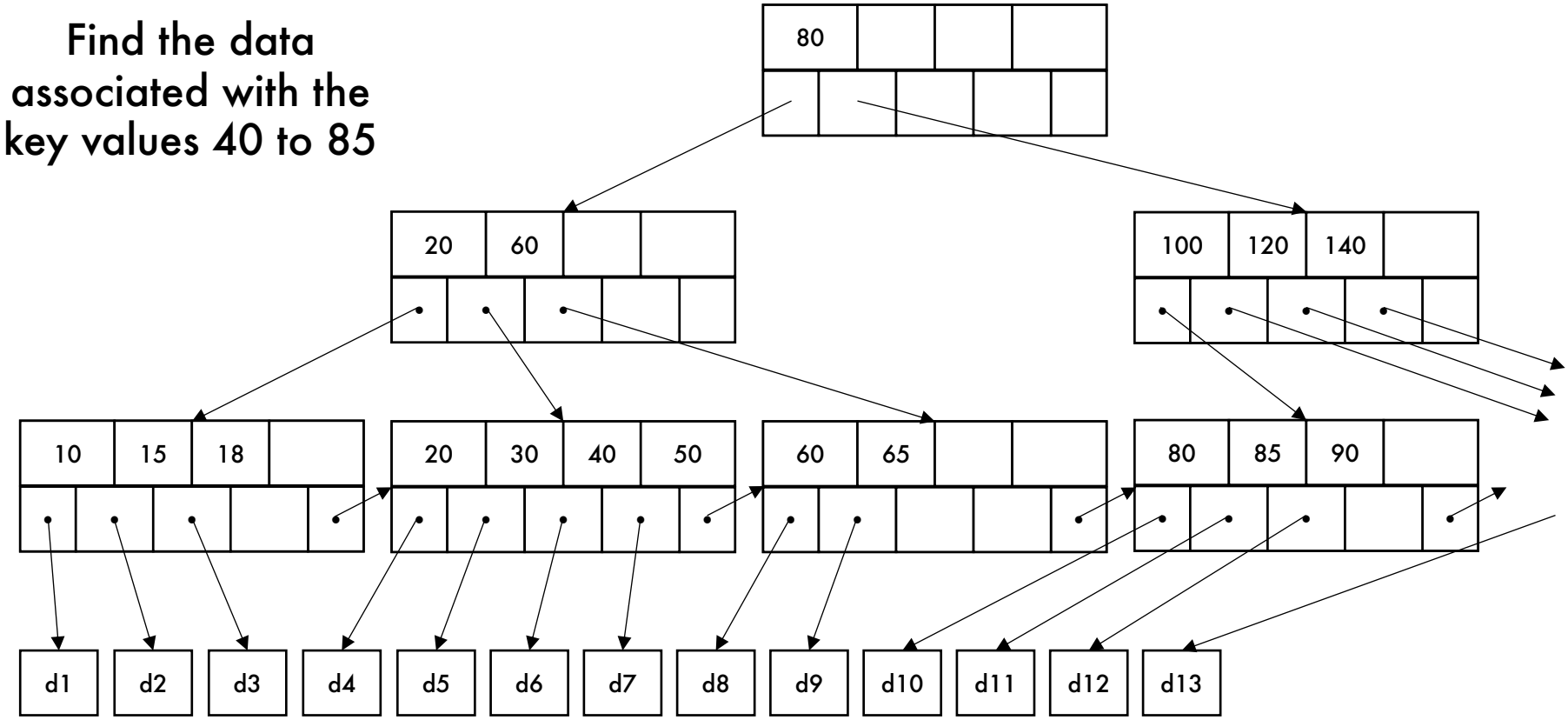
A **clustered index** is one that has the **same key ordering** as what is on disk (one per table)



Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

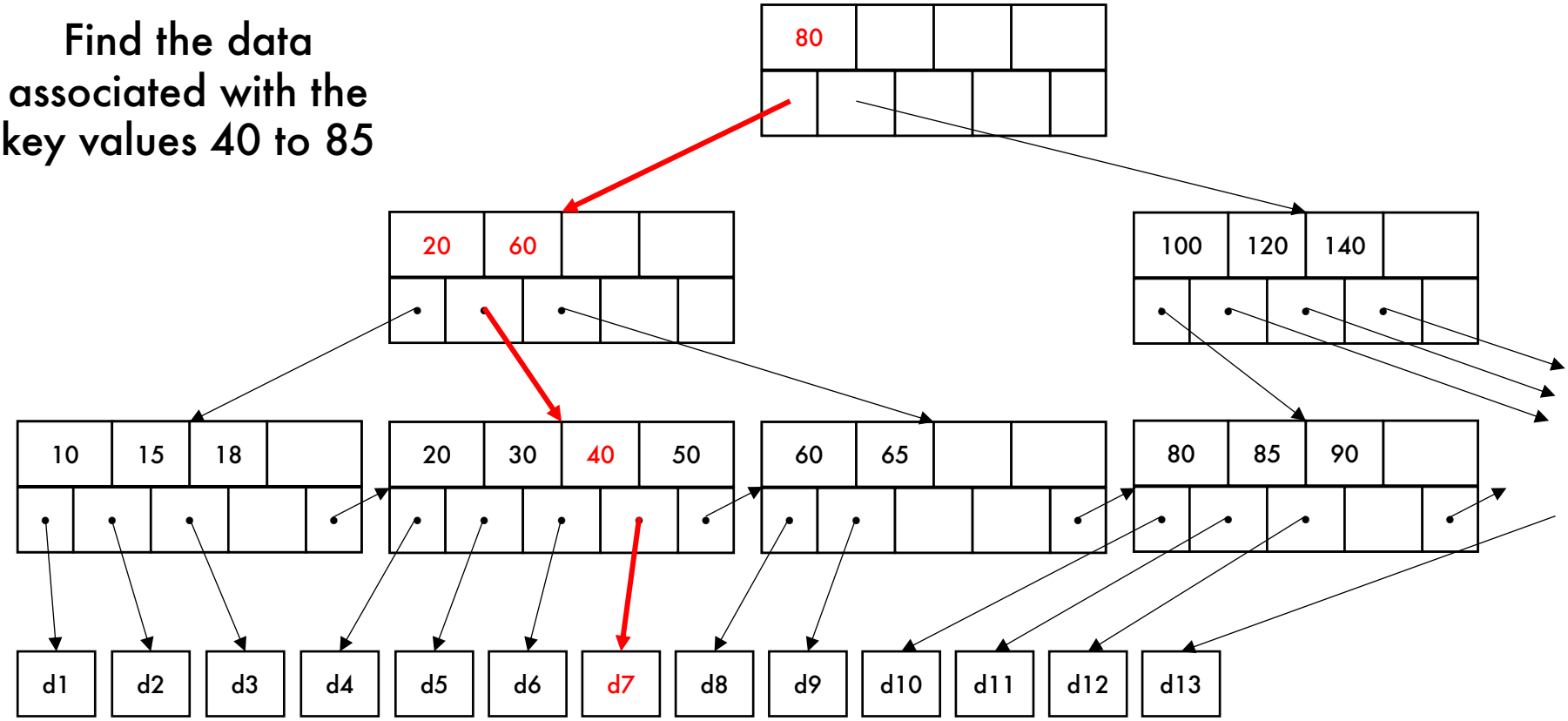
Find the data associated with the key values 40 to 85



Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

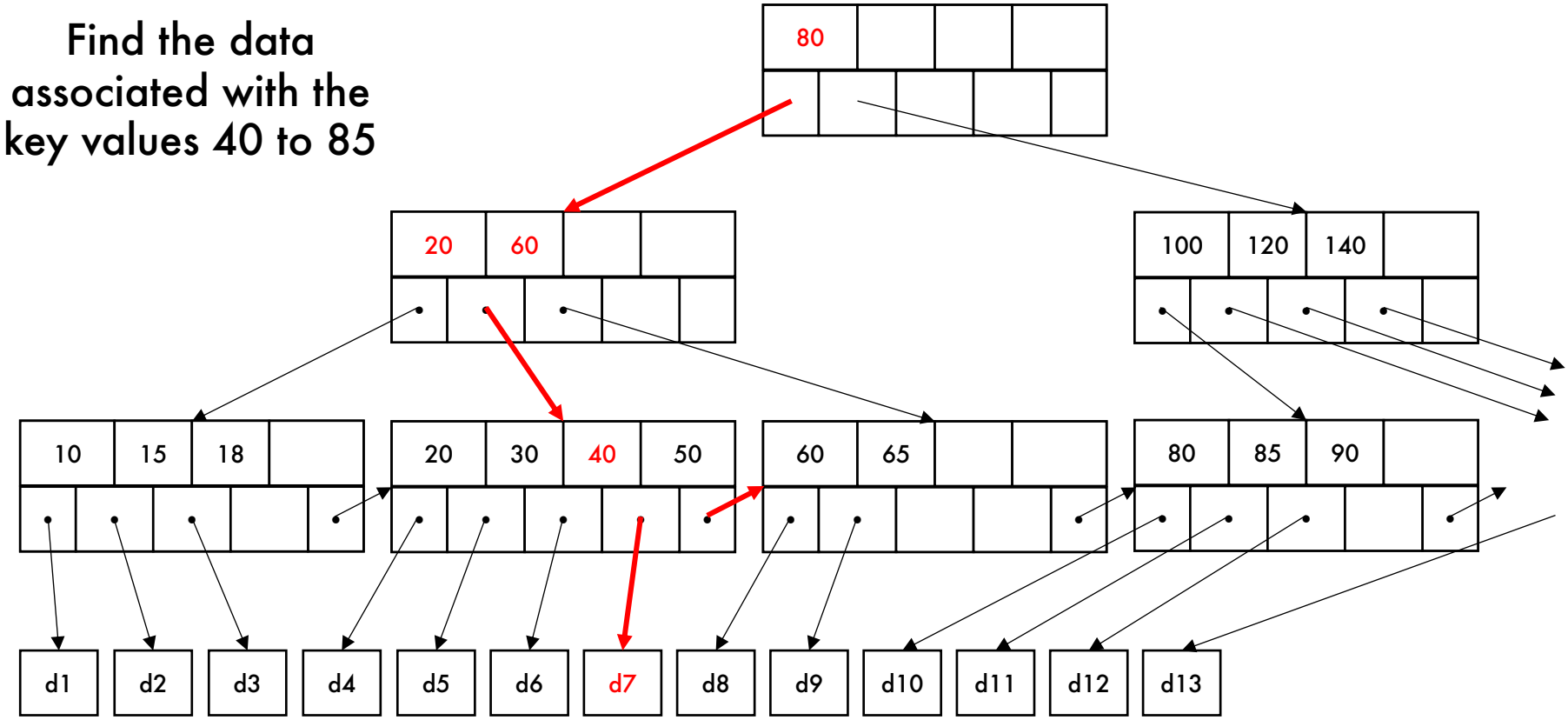
Find the data associated with the key values 40 to 85



Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

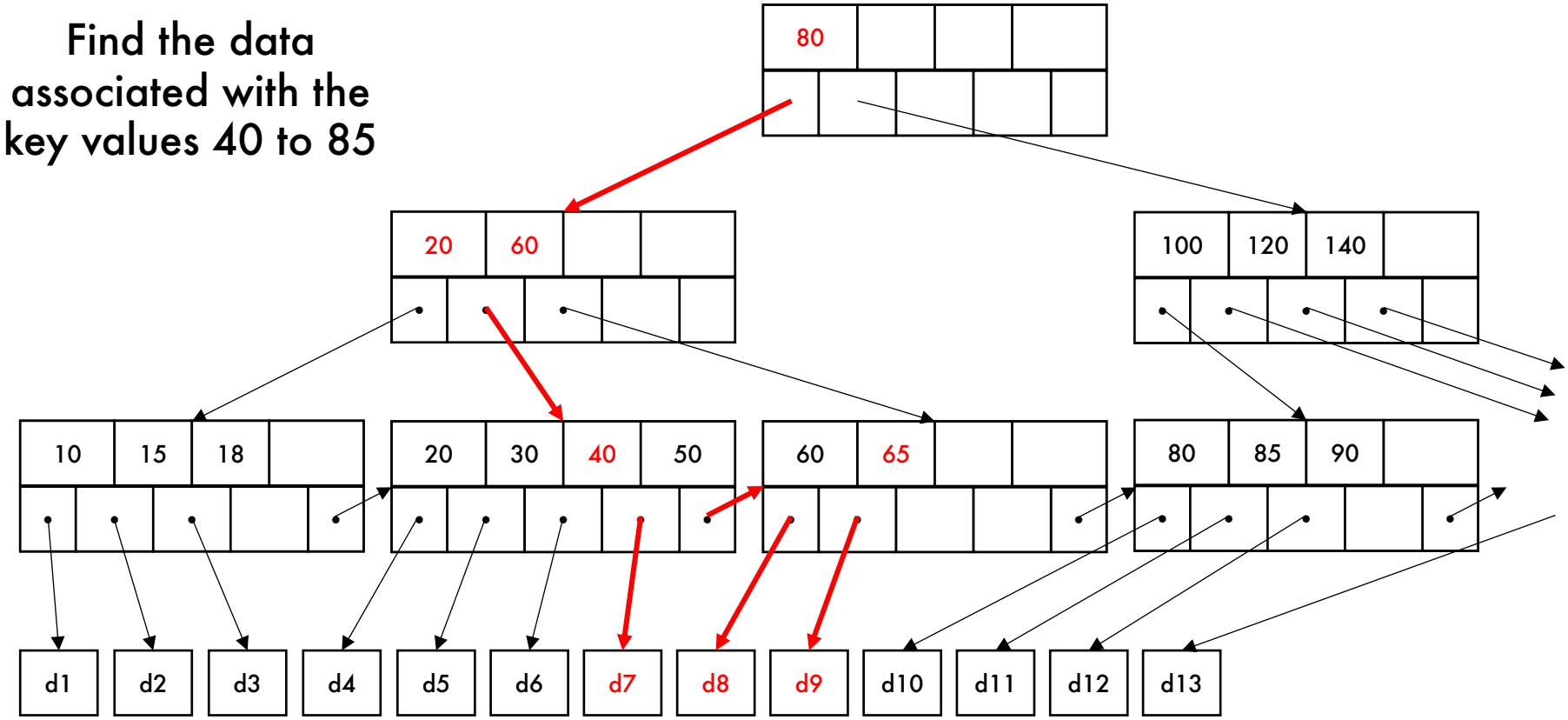
Find the data associated with the key values 40 to 85



Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

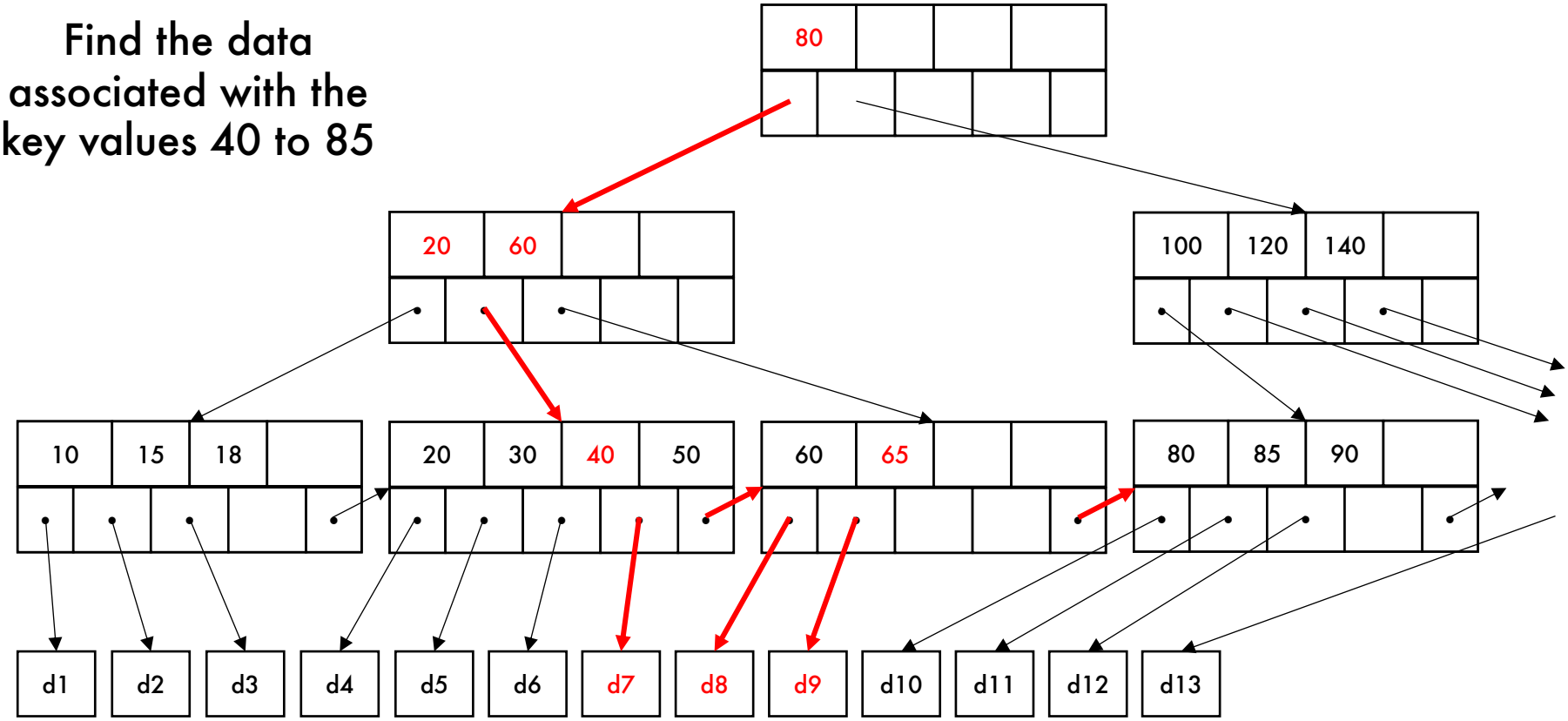
Find the data associated with the key values 40 to 85



Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

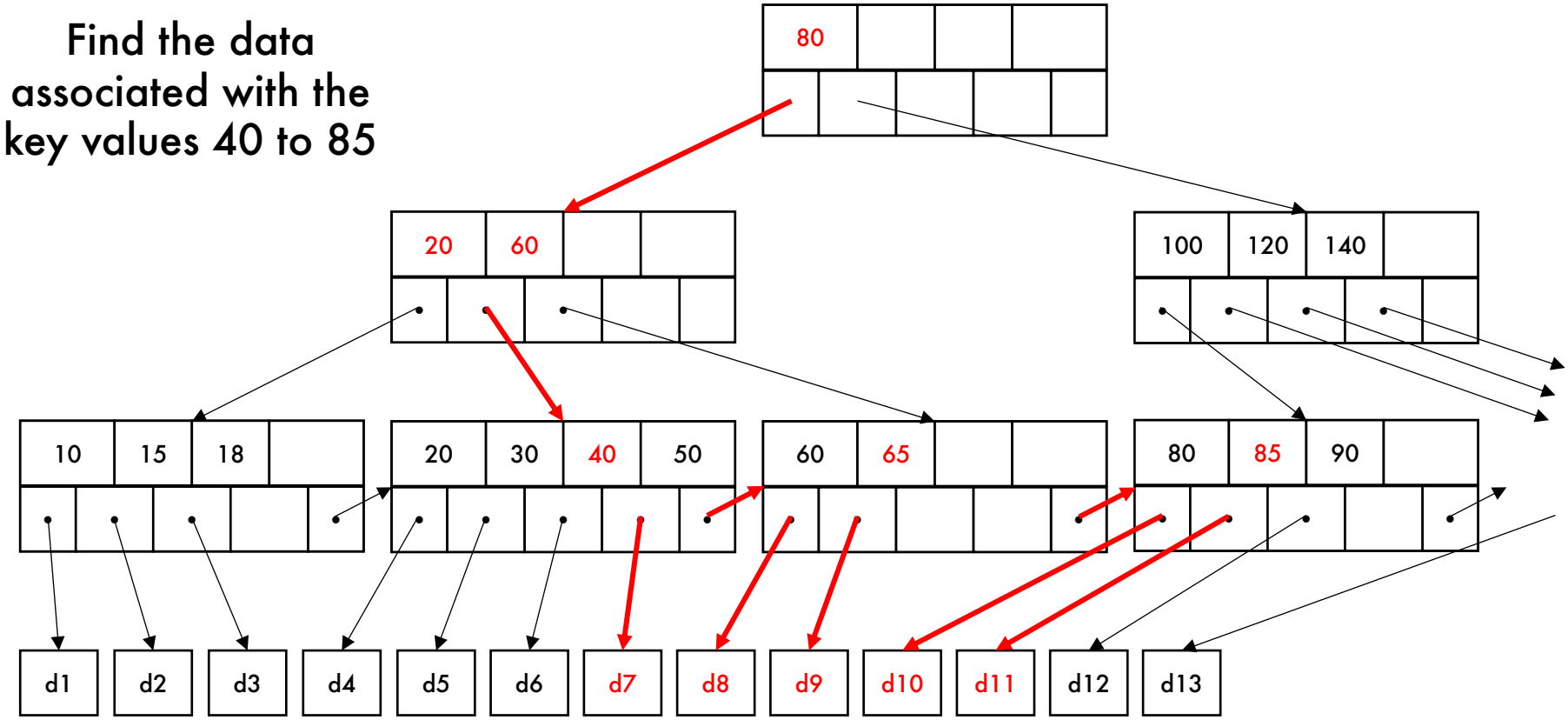
Find the data associated with the key values 40 to 85



Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

Find the data associated with the key values 40 to 85



Estimating Amount of Data Read

- **Selectivity Factor** (X) → Proportion of total data needed
- Assuming uniform distribution of data values on numeric attribute a in table R, if the condition is:
  - a = c → $X \cong \frac{1}{V(a,R)}$
  - a < c → $X \cong \frac{c-\min(a,R)}{\max(a,R)-\min(a,R)}$
  - c1 < a < c2 → $X \cong \frac{c2-c1}{\max(a,R)-\min(a,R)}$
  - cond1 AND cond2 → $X \cong X_1 * X_2$
- Disclaimer: More thorough selectivity estimation will use a histogram

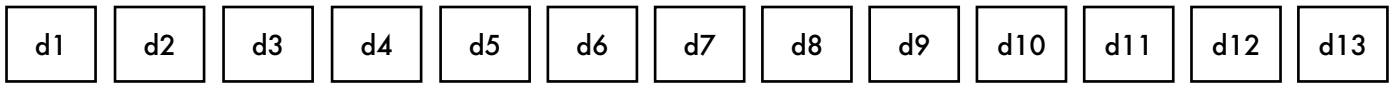
Index-Based Selection

- For reference, a full sequential scan of data costs B(R) IOs
- Provided some condition to read data:
  - Full **sequential scan → B(R)**
  - Scan on **clustered index → X*B(R)**
    - Able to read a contiguous chunk of the file
  - Scan on **unclustered index → X*T(R)**
    - Worst case would read a different block everytime

Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

Without an index, finding a value must be done the "old fashioned way"

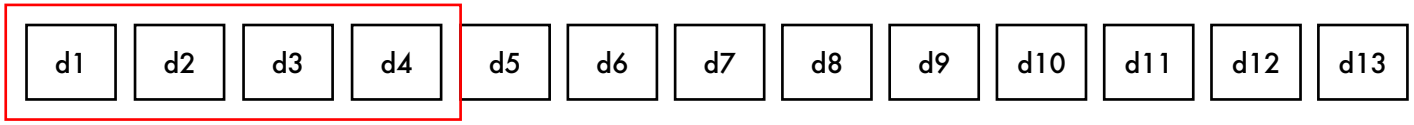


Disk

Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

Without an index, finding a value must be done the "old fashioned way"

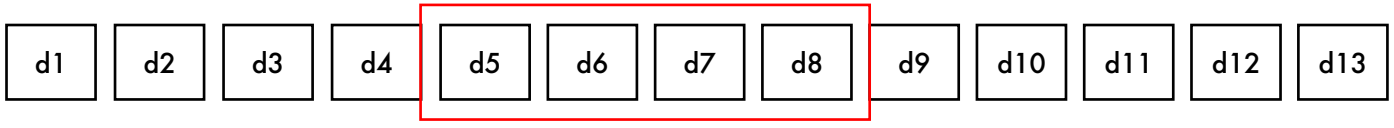


Disk

Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

Without an index, finding a value must be done the "old fashioned way"

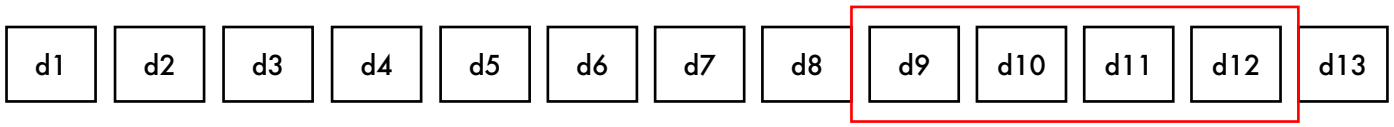


Disk

Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

Without an index, finding a value must be done the "old fashioned way"

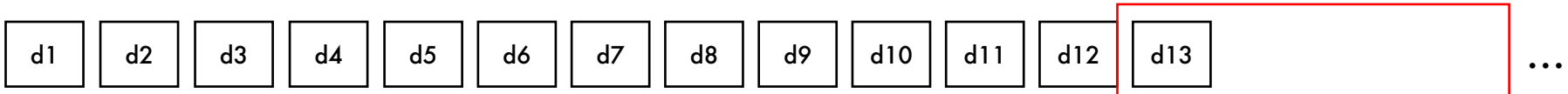


Disk

Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

Without an index, finding a value must be done the "old fashioned way"



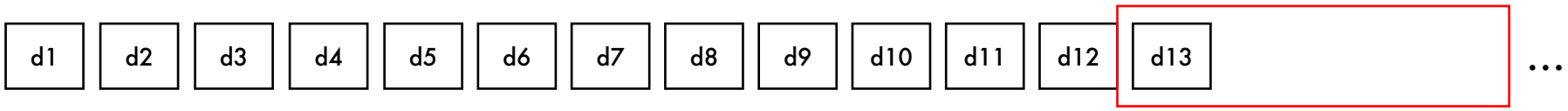
Disk

Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

Without an index, finding a value must be done the "old fashioned way"

Total cost: B(R)

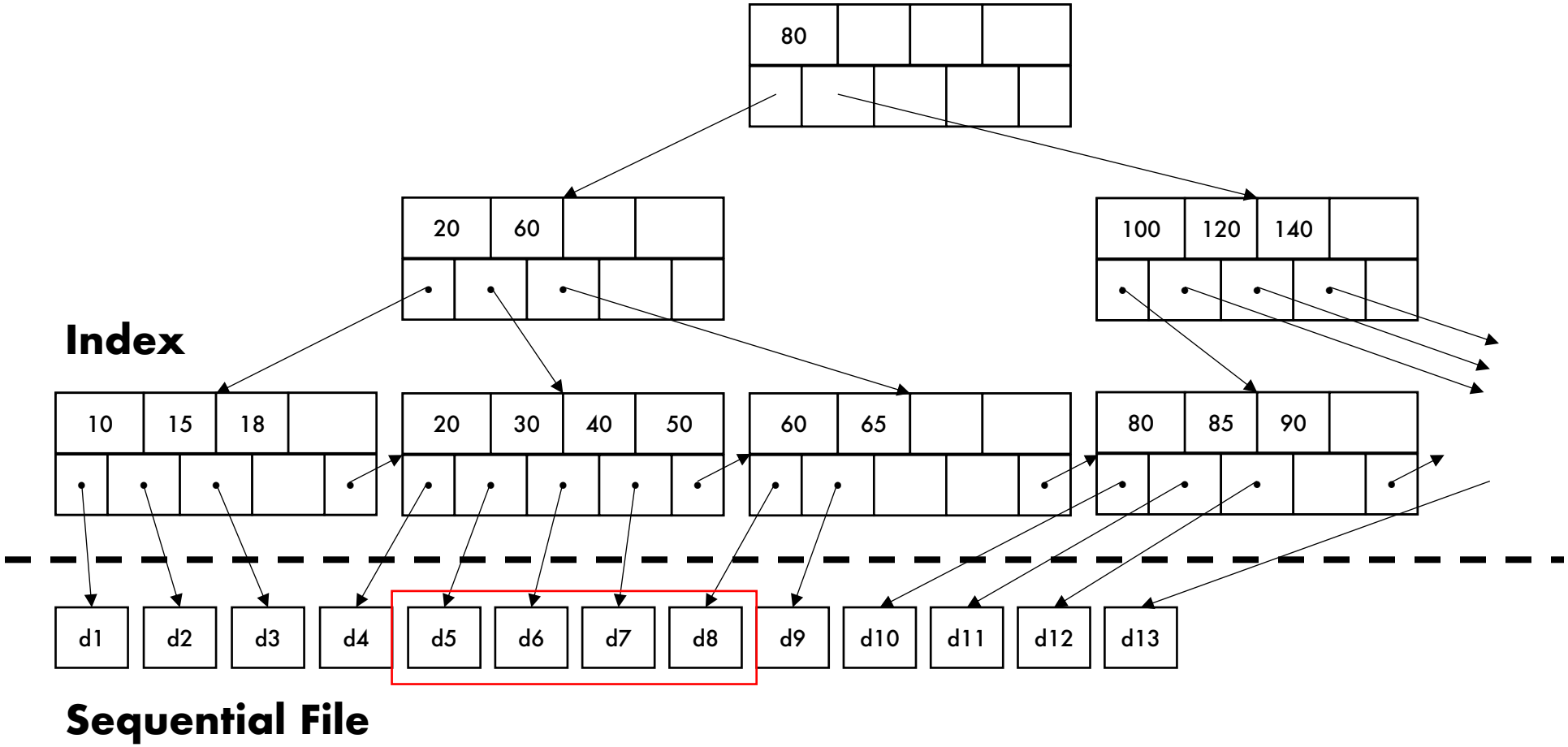


Disk

Clustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

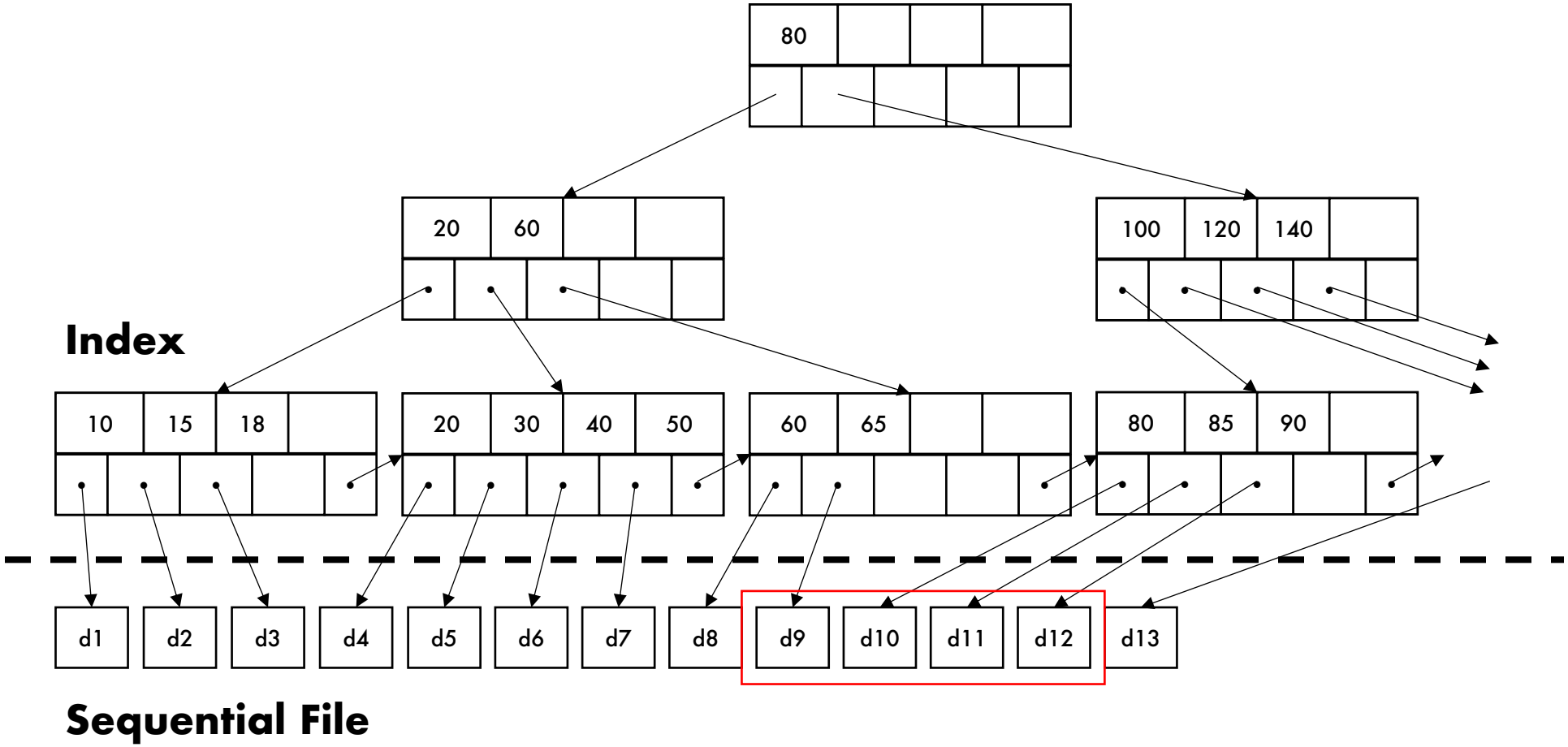
With a clustered index, I start scanning blocks in the range they are at



Clustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

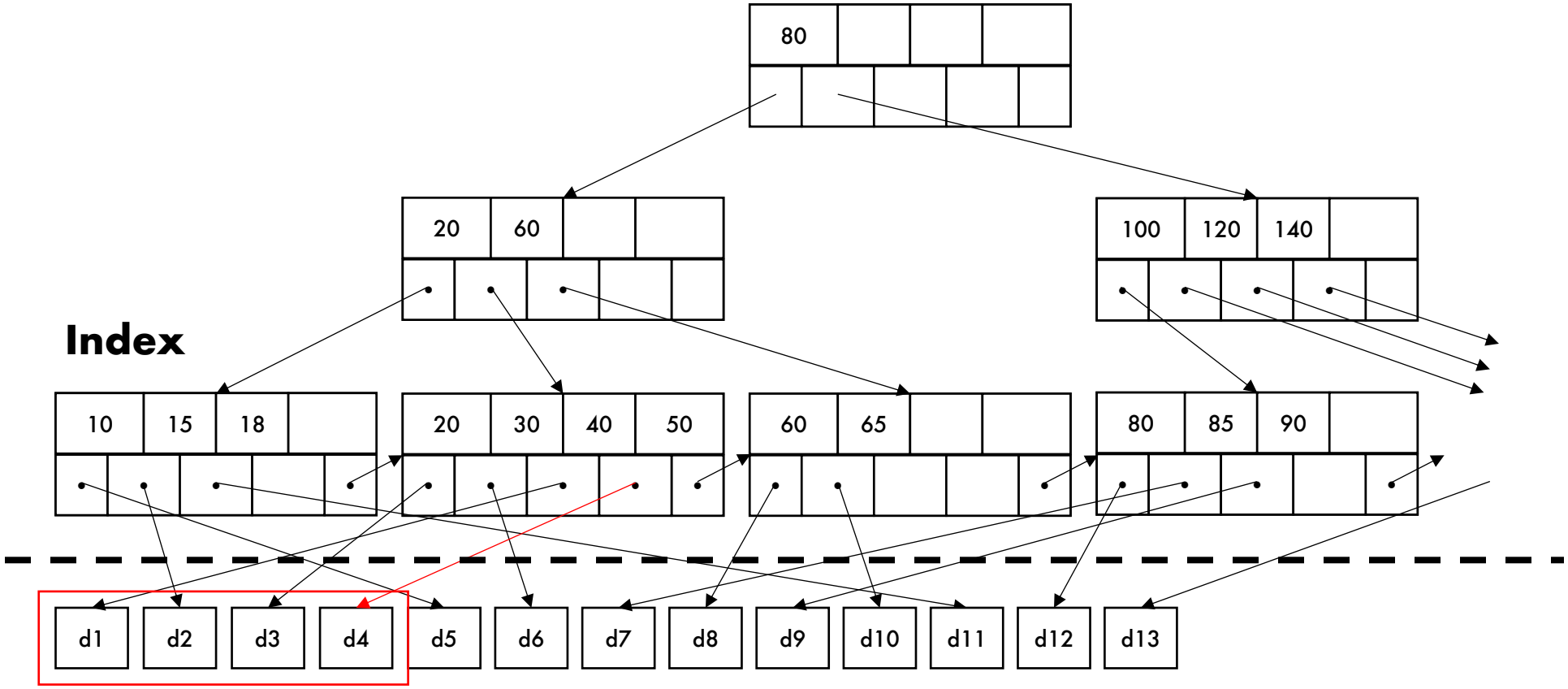
With a clustered index, I start scanning blocks in the range they are at



Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With an unclustered index, I start scanning tuples wherever they occur

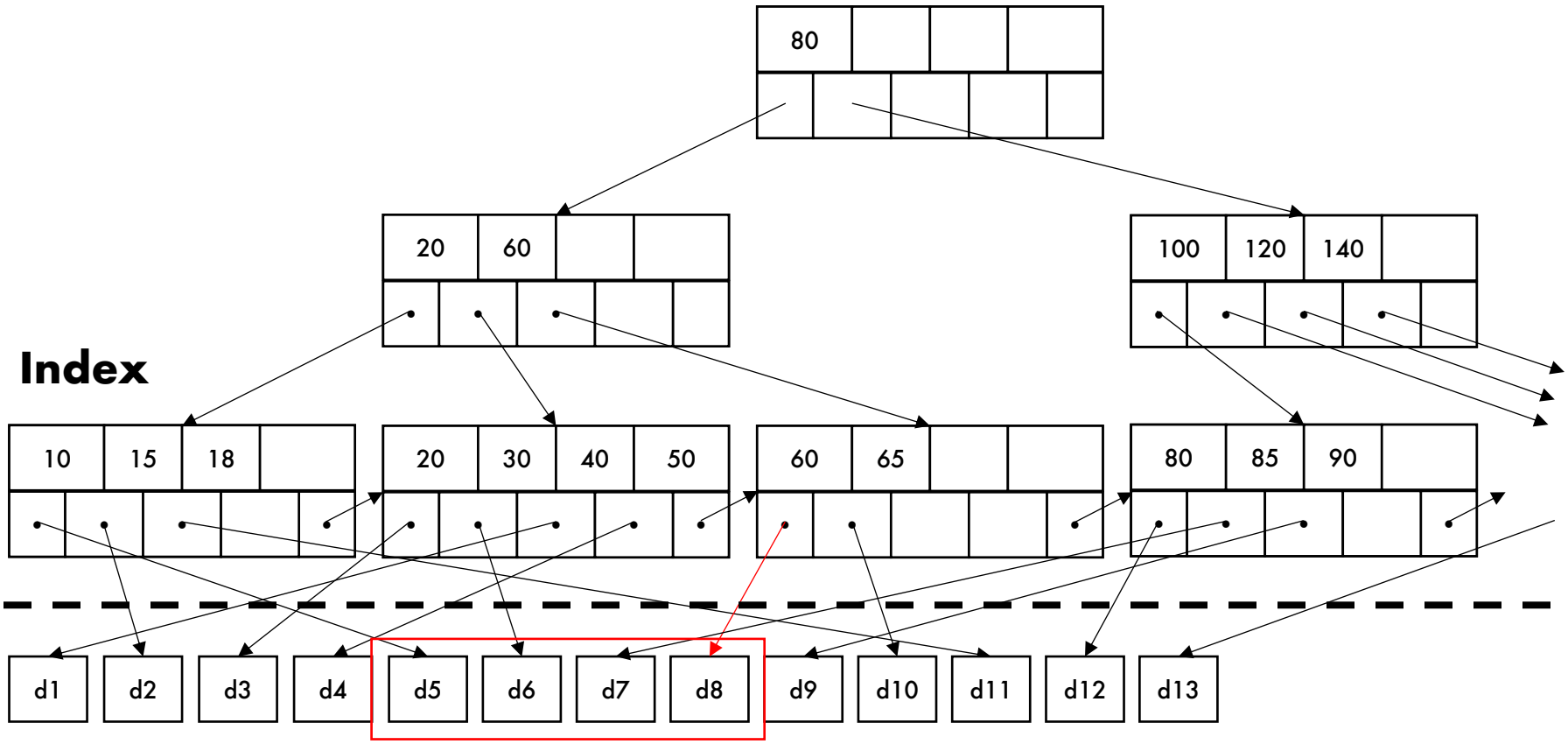


Sequential File with a different key or **Heap File**

Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With an unclustered index, I start scanning tuples wherever they occur

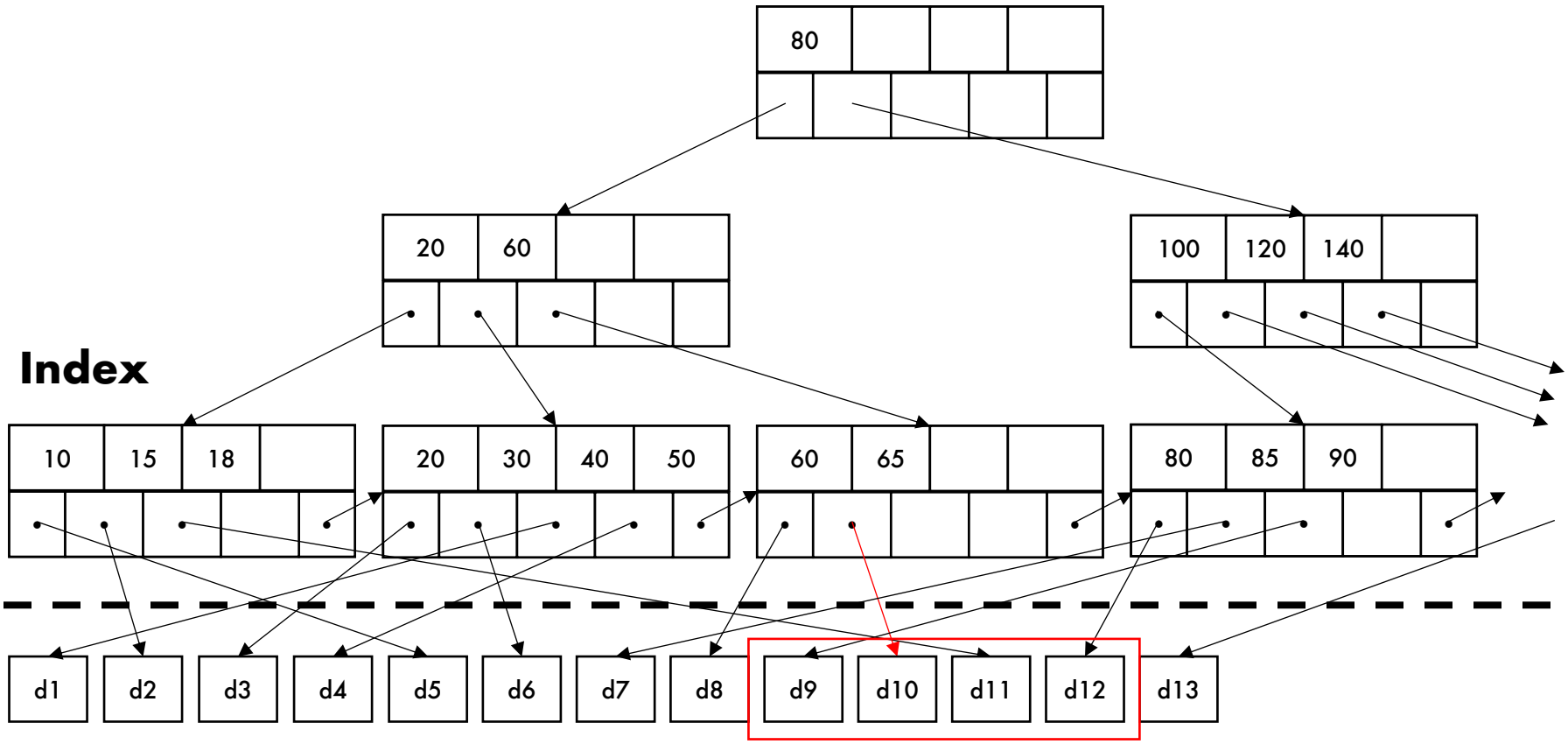


Sequential File with a different key or **Heap File**

Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With an unclustered index, I start scanning tuples wherever they occur

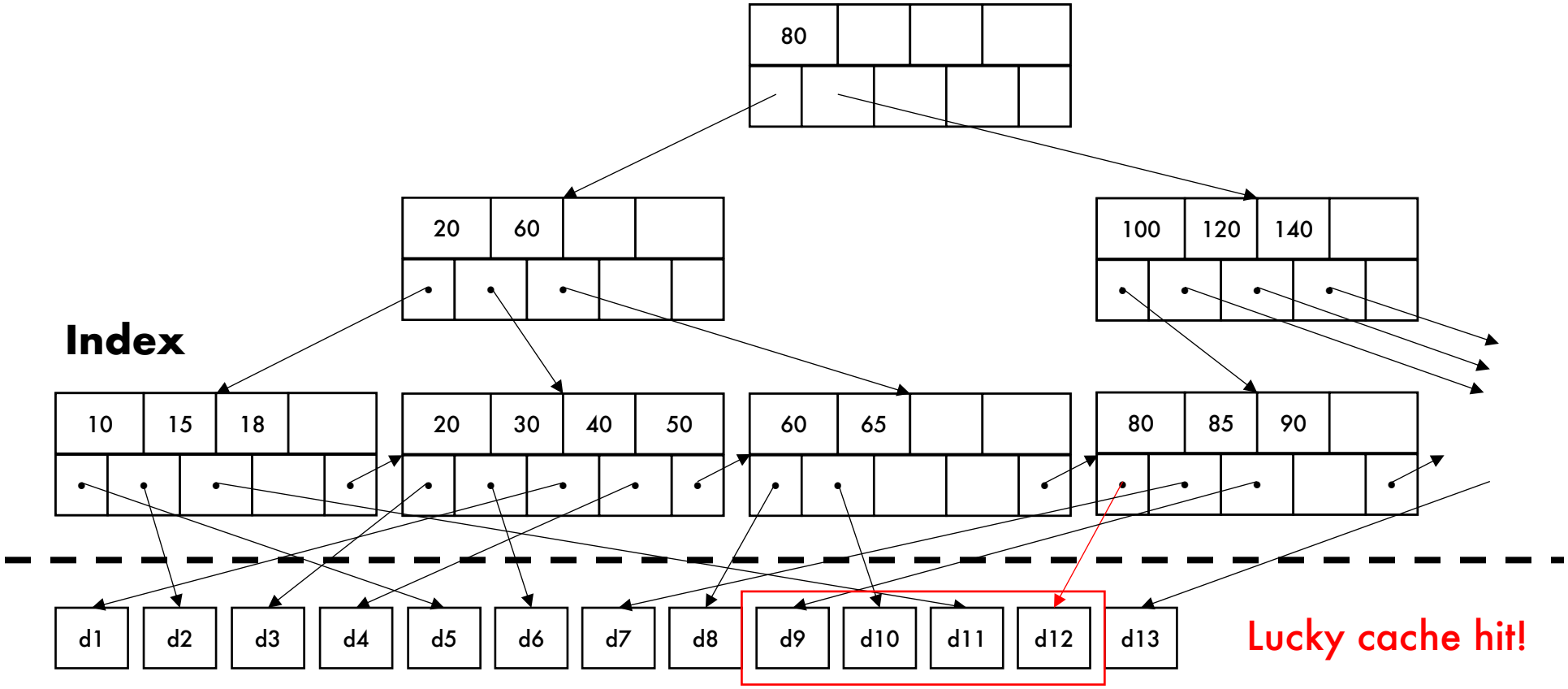


Sequential File with a different key or **Heap File**

Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With an unclustered index, I start scanning tuples wherever they occur

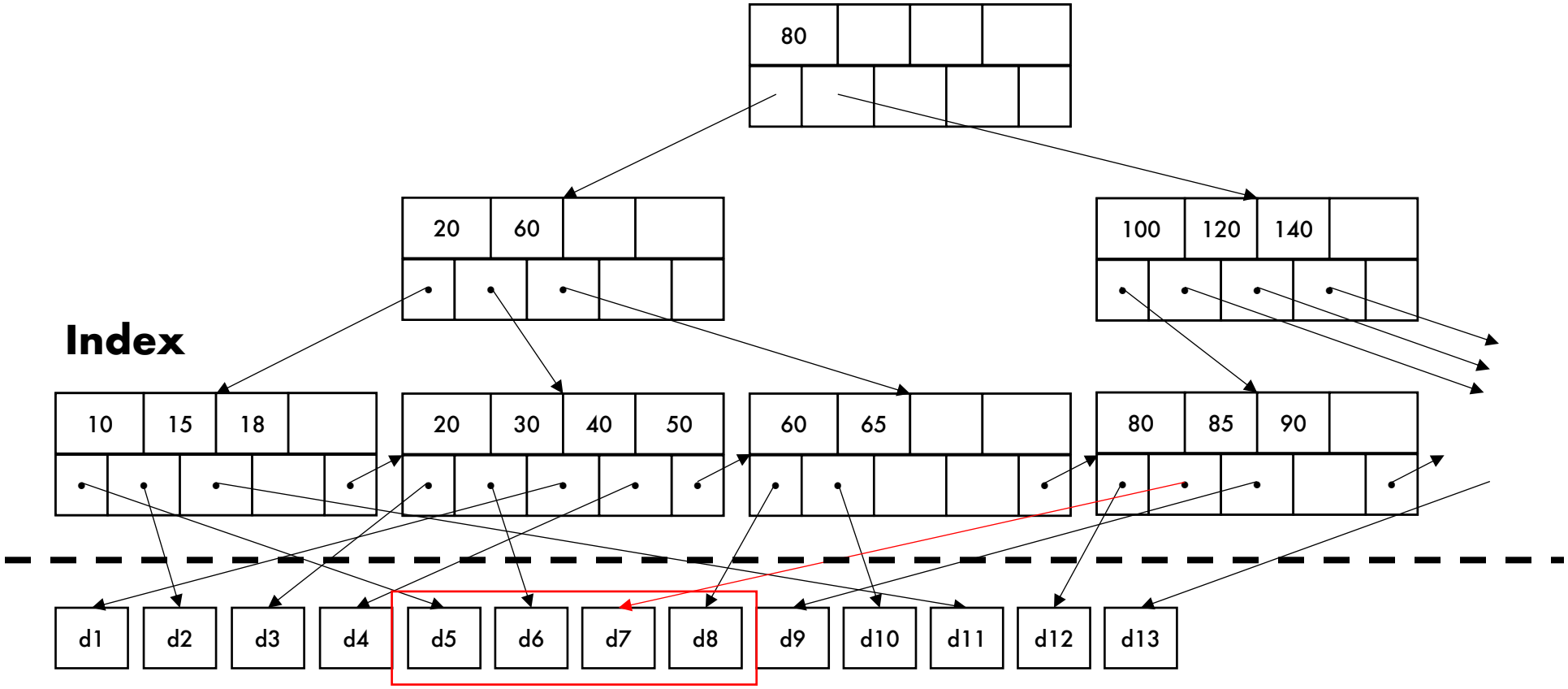


Sequential File with a different key or **Heap File**

Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With an unclustered index, I start scanning tuples wherever they occur



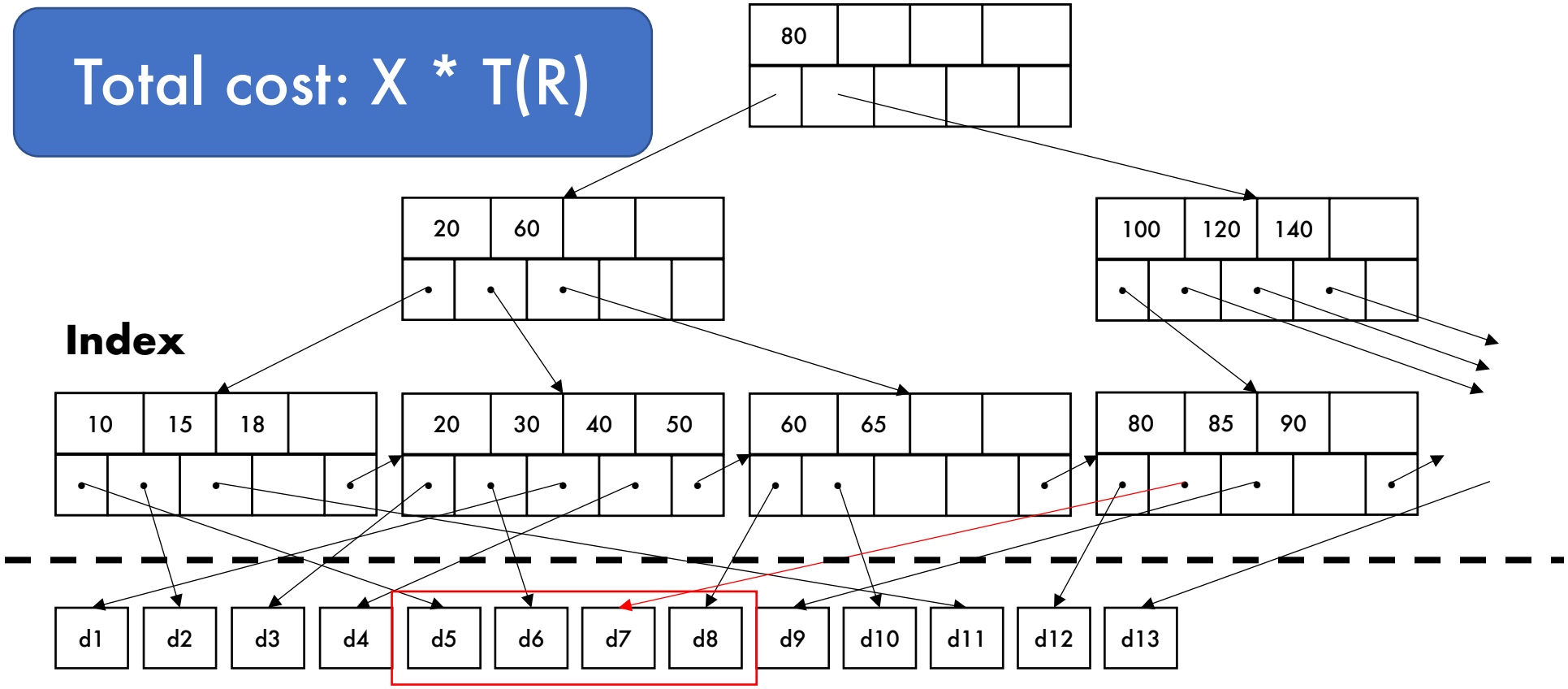
Sequential File with a different key or **Heap File**

Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With an unclustered index, I start scanning tuples wherever they occur

Total cost: X * T(R)



Sequential File with a different key or **Heap File**

Index Expectations

- Using an index in the wrong scenario can lead to a slowdown!
- Common example: Full sequential scan vs unclustered index scan with **high X value** and/or **small tuple size** (large T(R):B(R) ratio)

Known:
B(R) = 100
T(R) = 10000

Consider a query with X=1/10

Sequential scan
= B(R)
= **100**

Index scan
= X*T(R)
= 1/10 * 10000
= **1000**

Index Expectations

- Using an index in the wrong scenario can lead to a slowdown!
- Common example: Full sequential scan vs unclustered index scan with **high X value** and/or **small tuple size** (large T(R):B(R) ratio)

Having indexes doesn't mean you will see a speedup!

Known:
B(R) = 100
T(R) = 10000

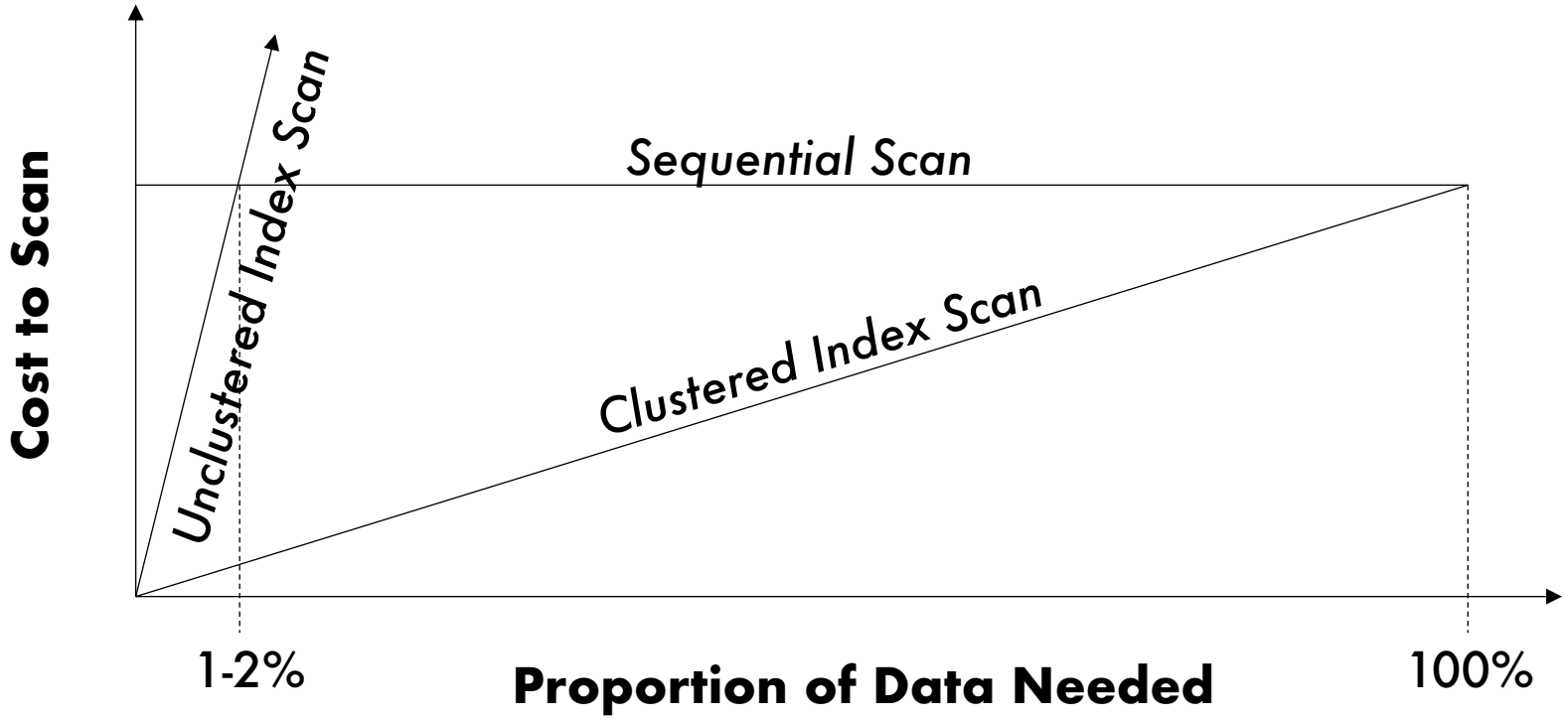
Consider a query with X=1/10

Sequential scan
= B(R)
= **100**

Index scan
= X*T(R)
= 1/10 * 10000
= **1000**

Index Expectations

- **Sequential disk reads are faster than random ones**
  - Cost ~1-2% random scan = full sequential scan



Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)

CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Reorders data on disk! (Fails if another clustered index exists)

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);
```

Fine to create indexes on non-numeric data (just not discussed in this class)

```
CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Reorders data on disk! (Fails if another clustered index exists)

Index-Based Equijoin

- Assume index exists on the join attribute a of S

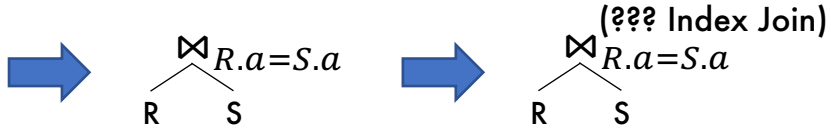
```
SELECT *
  FROM R, S
 WHERE R.a = S.a
```

- **Clustered Index Join**
  - Perform a clustered index scan for each tuple of R
  - B(R)+T(R)(X*B(S)) = **B(R)+T(R)(B(S)/V(a,S))**
- **Unclustered Index Join**
  - Perform an unclustered index scan for each tuple of R
  - B(R)+T(R)(X*T(S)) = **B(R)+T(R)(T(S)/V(a,S))**

Index-Based Equijoin

- Assume index exists on the join attribute a of S

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



Can't scan per block of R since tuples in blocks don't have the same attribute values

- **Clustered Index Join**
  - Perform a clustered index scan for each tuple of R
  - B(R)+T(R)(X*B(S)) = **B(R)+T(R)(B(S)/V(a,S))**
- **Unclustered Index Join**
  - Perform an unclustered index scan for each tuple of R
  - B(R)+T(R)(X*T(S)) = **B(R)+T(R)(T(S)/V(a,S))**

Leveraging Indexes

- Often for applications, workloads can be well described
  - Lab 1 Flights application
    - Search method → query on city name values
  - Data visualization software (e.g. Tableau)
    - 2D plot → query on graph axis bounds
- **Create indexes to match expected query workload**

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclusted index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclusted index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

This range query would benefit from a clustered index on score

Only one can exist!

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

This range query would benefit from a clustered index on age

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

What indexes could we make on Users?

IDs are unique so an unclusted index would do fine.

Things to consider:

- What is the expected result size for each query?
- Do either of these queries need to be returned ASAP?

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclusted index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Without more information, default to clustering on the index that will be used more (clustered index on score)

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclusted index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Hack:

- Create a covering index primarily keyed on score
- Create a covering index primarily keyed on age

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

What indexes could we make on Users?

IDs ed
inde

Essentially a sorted copy of the table. Fast but space inefficient and table updates are slow.

Hack:

- Create a covering index primarily keyed on score
- Create a covering index primarily keyed on age

"I'm not about that code monkey life"

- Choosing how to configure a database system is an interesting (i.e. hard) problem
- A database that is used by many people will often need one or more dedicated personnel to manage it (Database Administrator)
  - Logical design (multi-team coordination)
  - Physical design (hardware and system considerations)
  - Permission management (visibility and security)
  - Integration (company acquisitions and mergers)
  - …

Multiple Joins

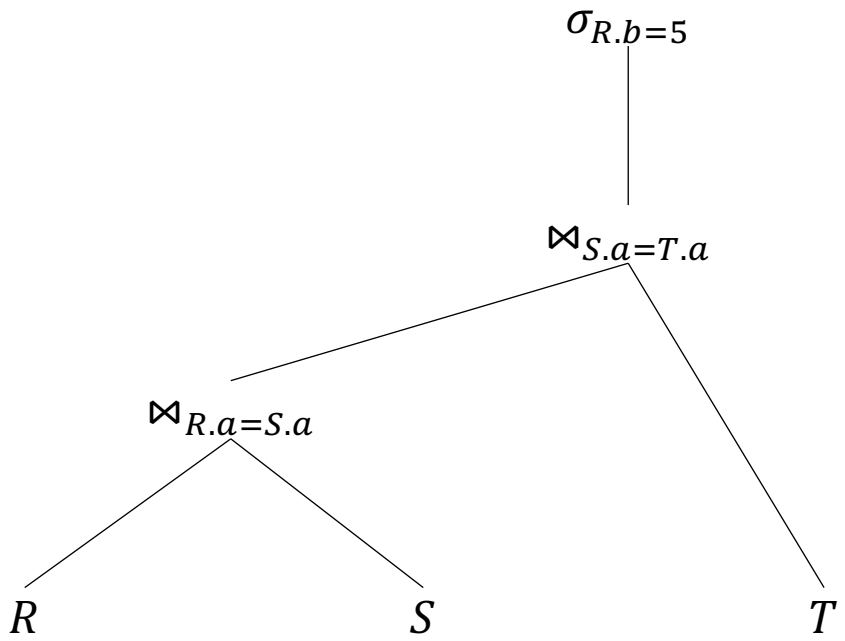
- **Pipelined Execution**
  - Tuples are processed through the entire query plan
  - Fast
- Blocking Execution
  - Subplans are computed and stored before parent operation can start
  - Simple

Pipelined Execution

- Iterator interface of RA operators (**Volcano Iterator Model**)
  - open() on every operator at start
  - close() on every operator at end
  - next() to get the next tuple from a child operator or input table

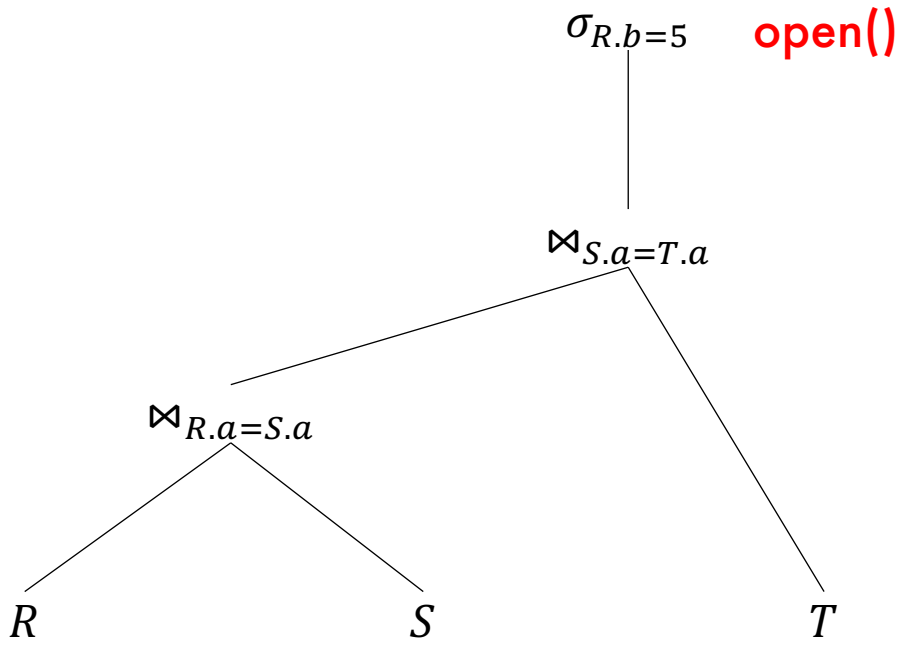
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



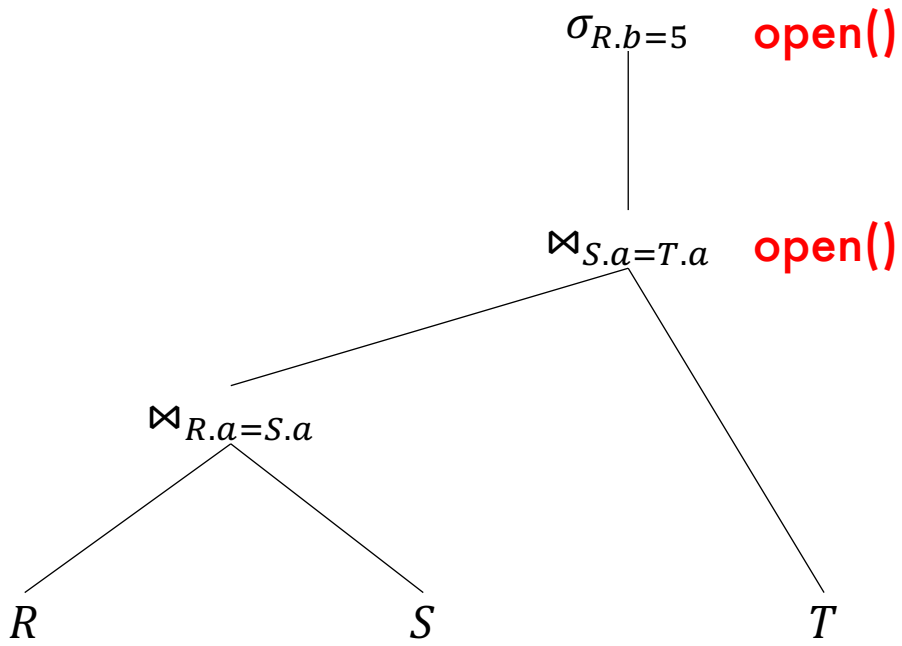
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



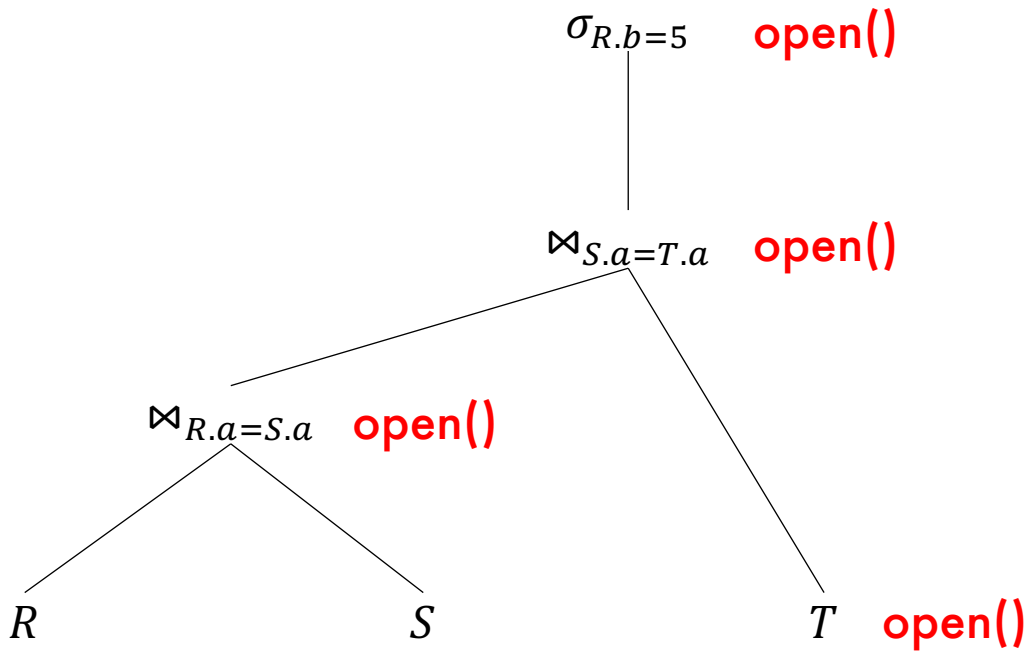
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



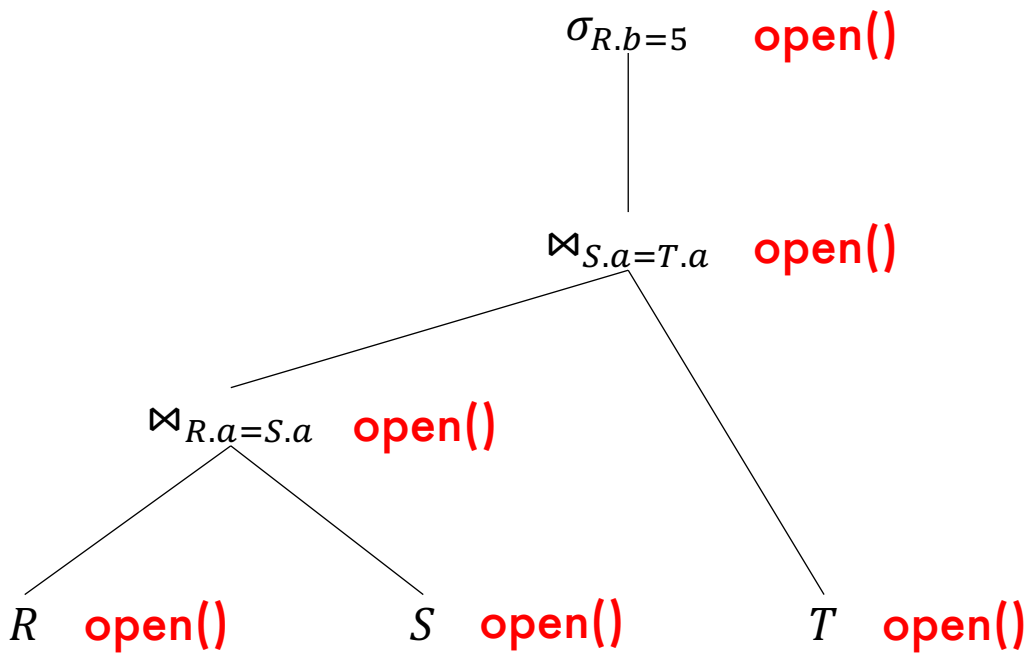
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



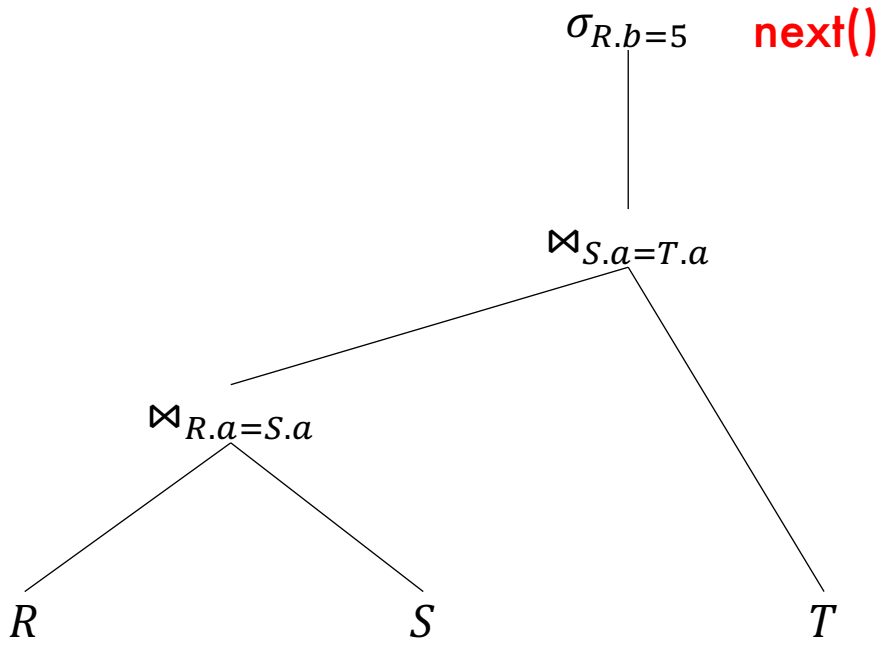
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



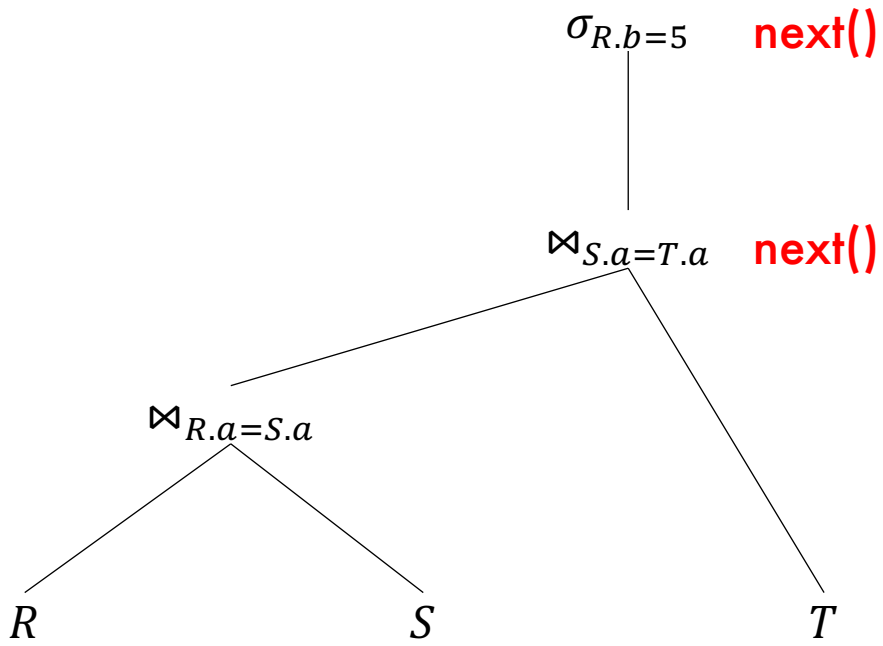
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



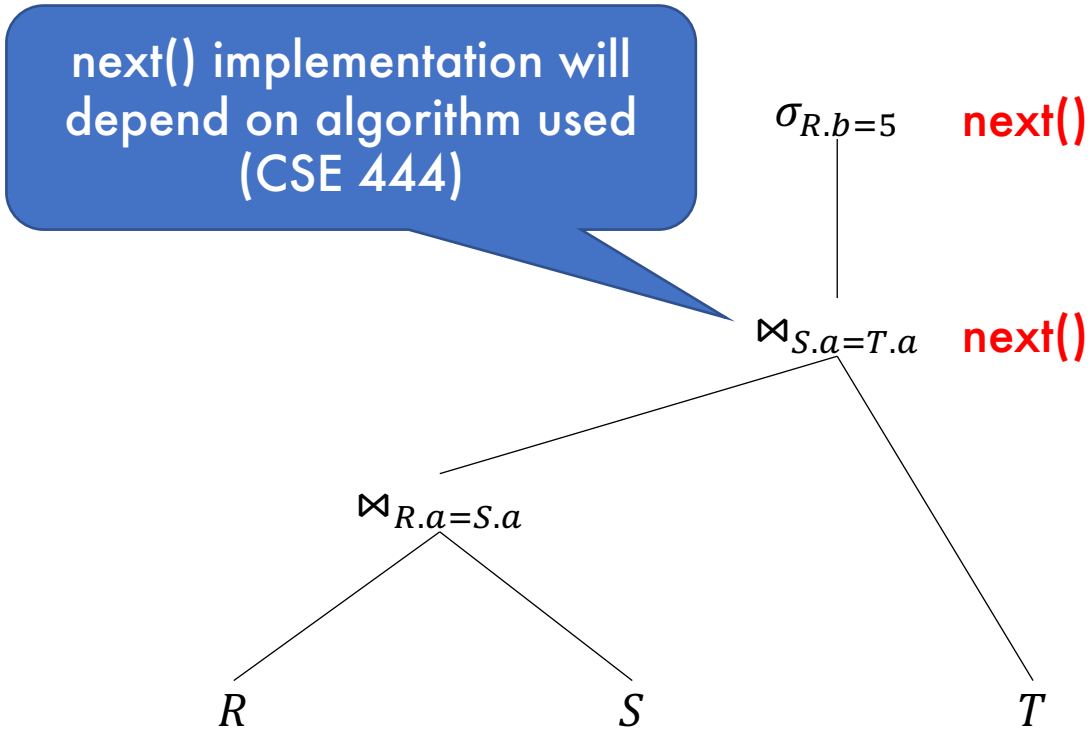
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



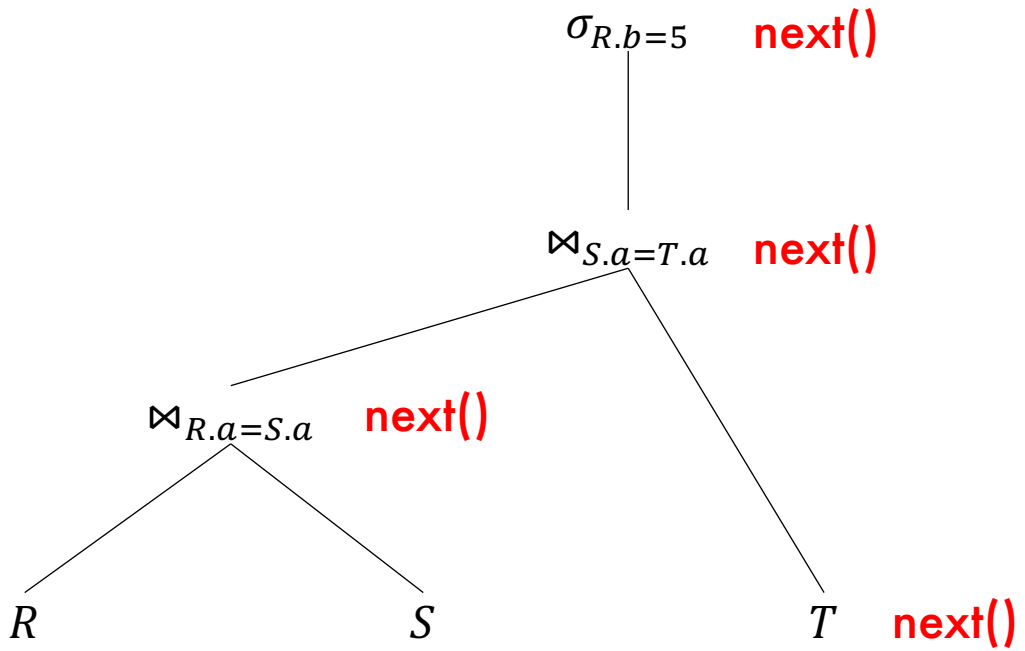
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



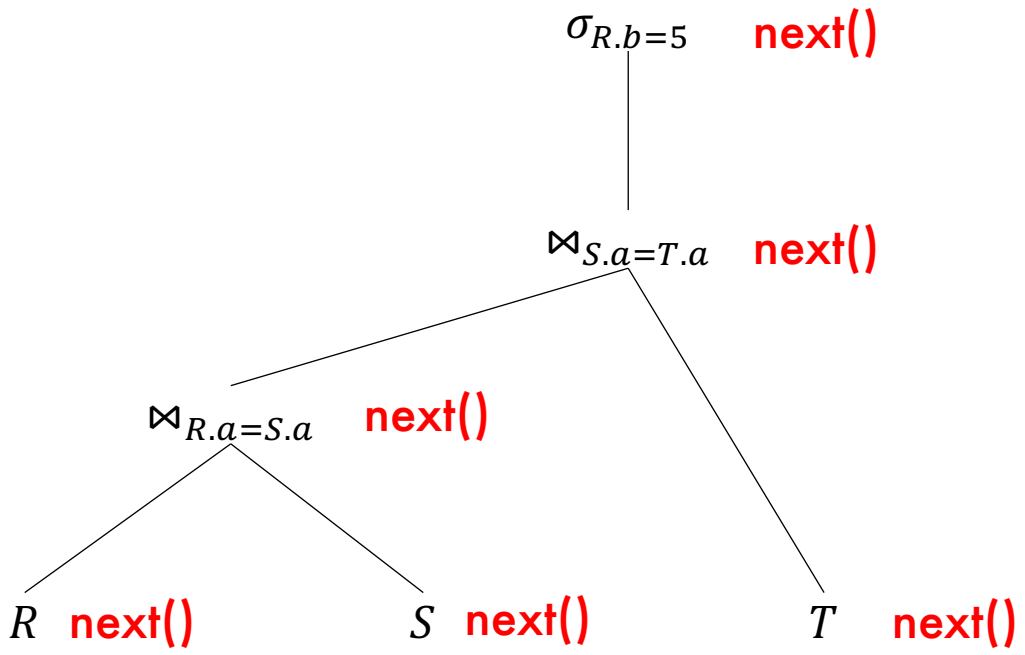
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



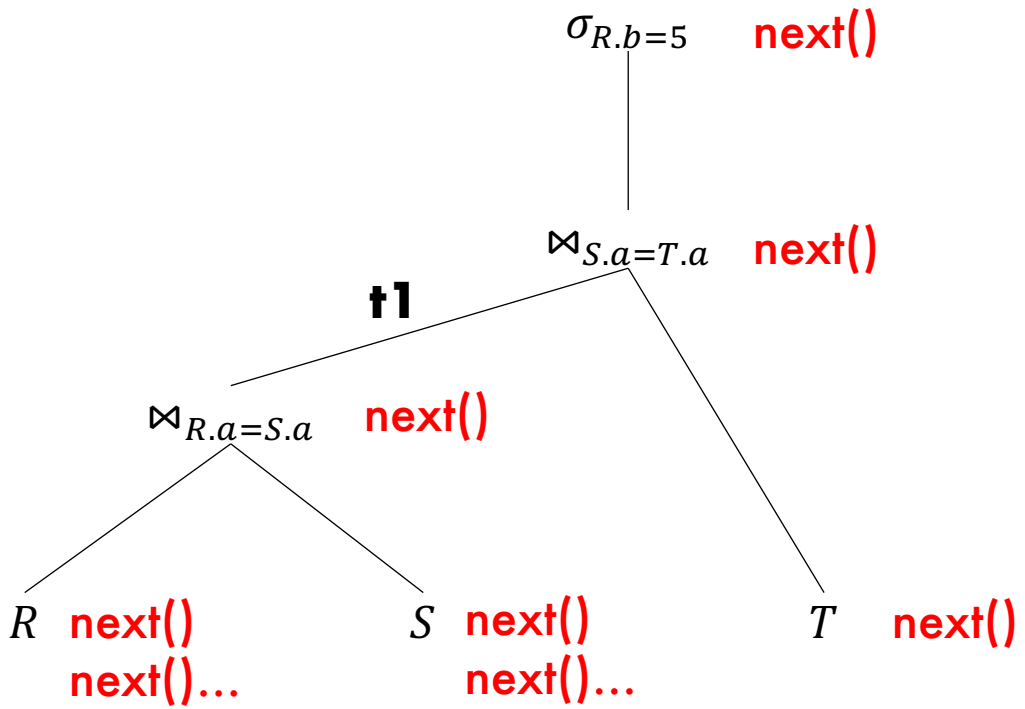
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



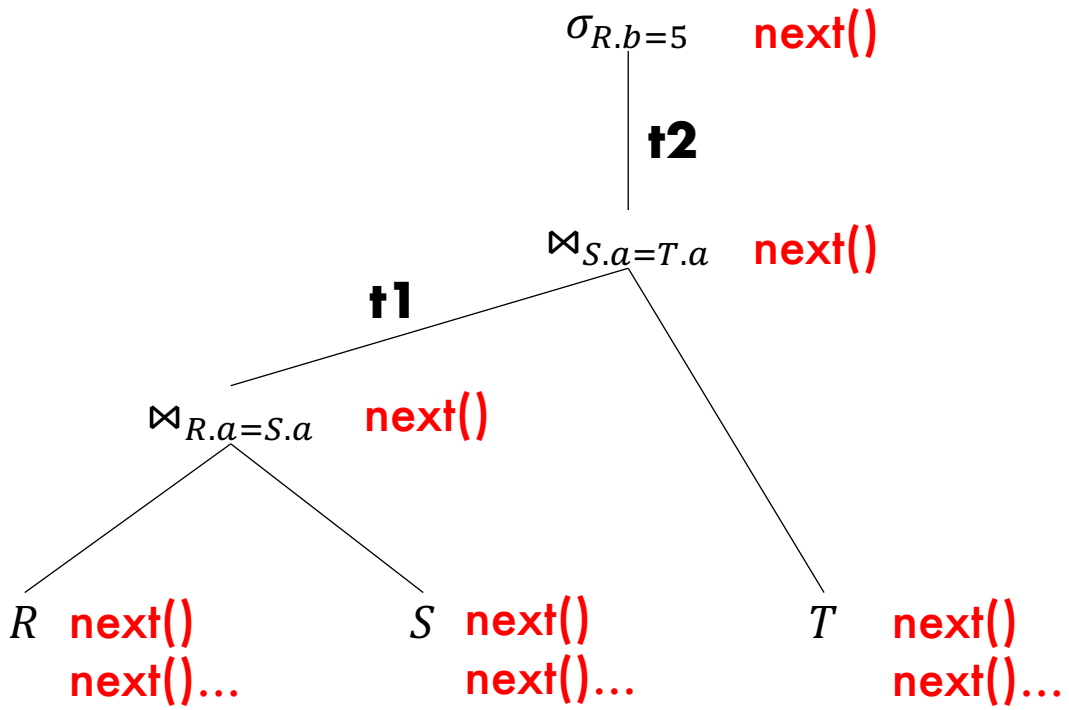
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



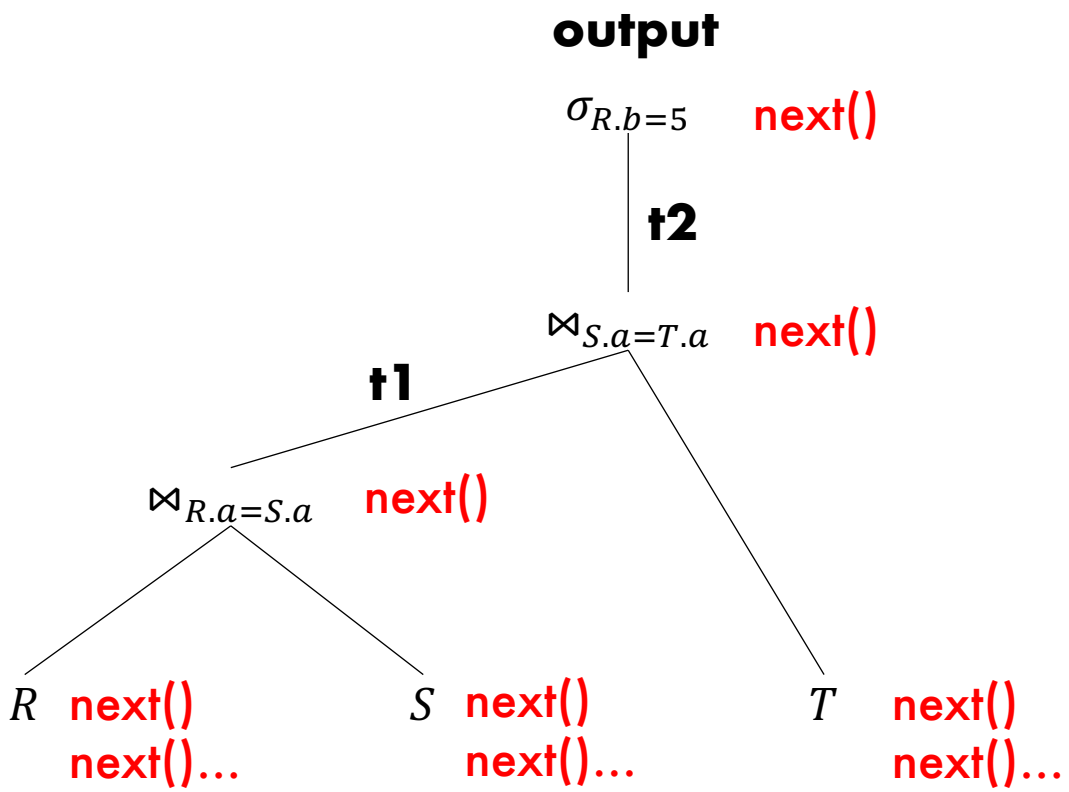
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



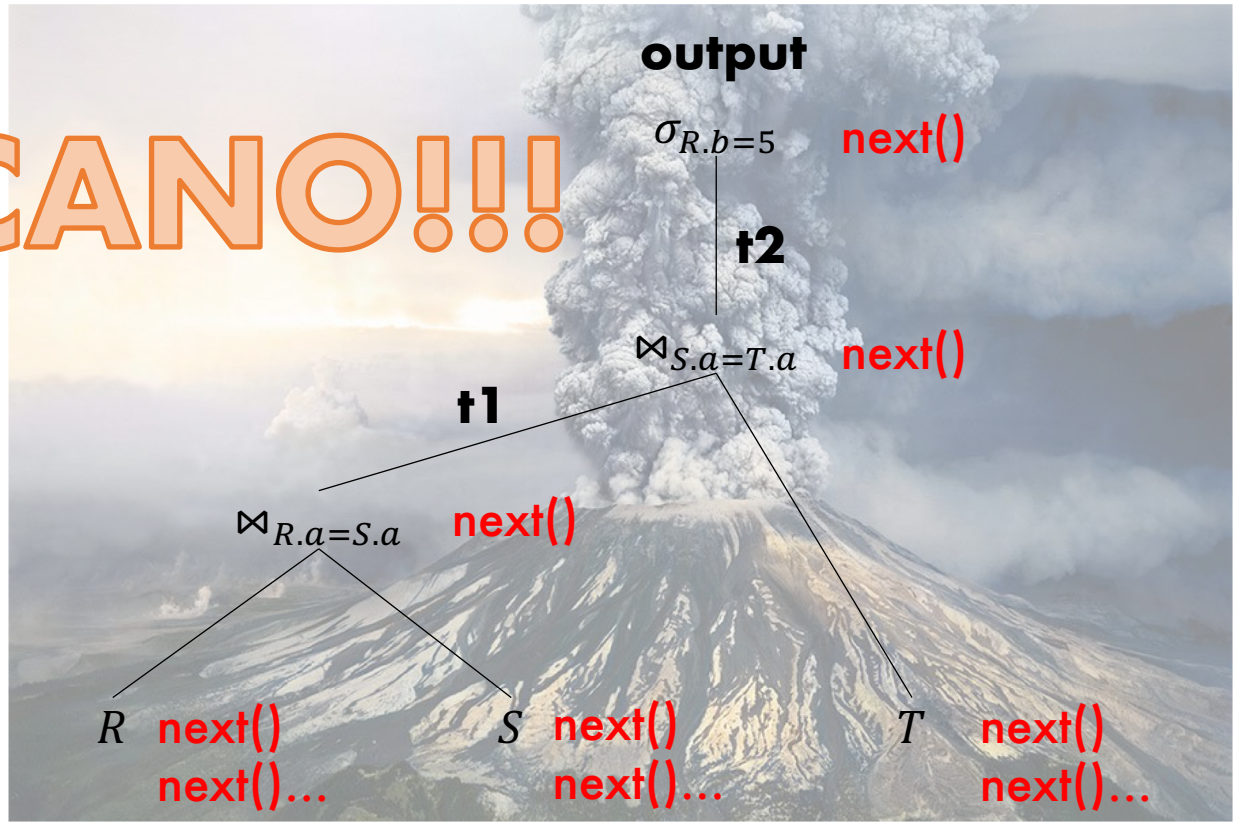
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



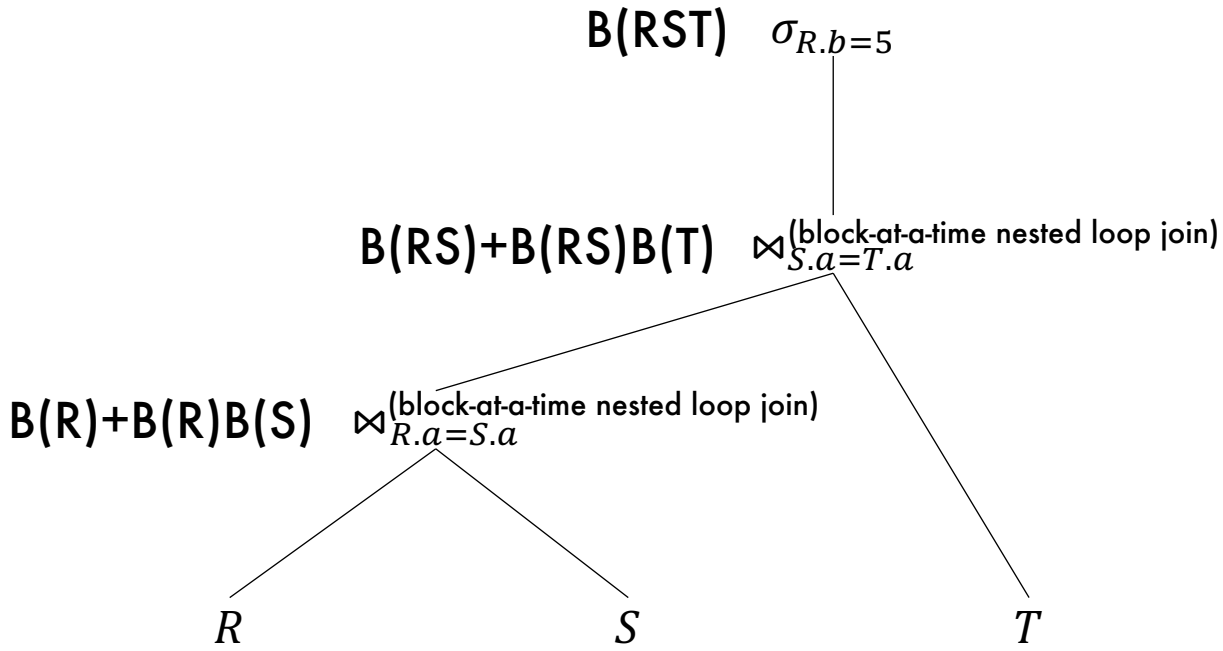
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



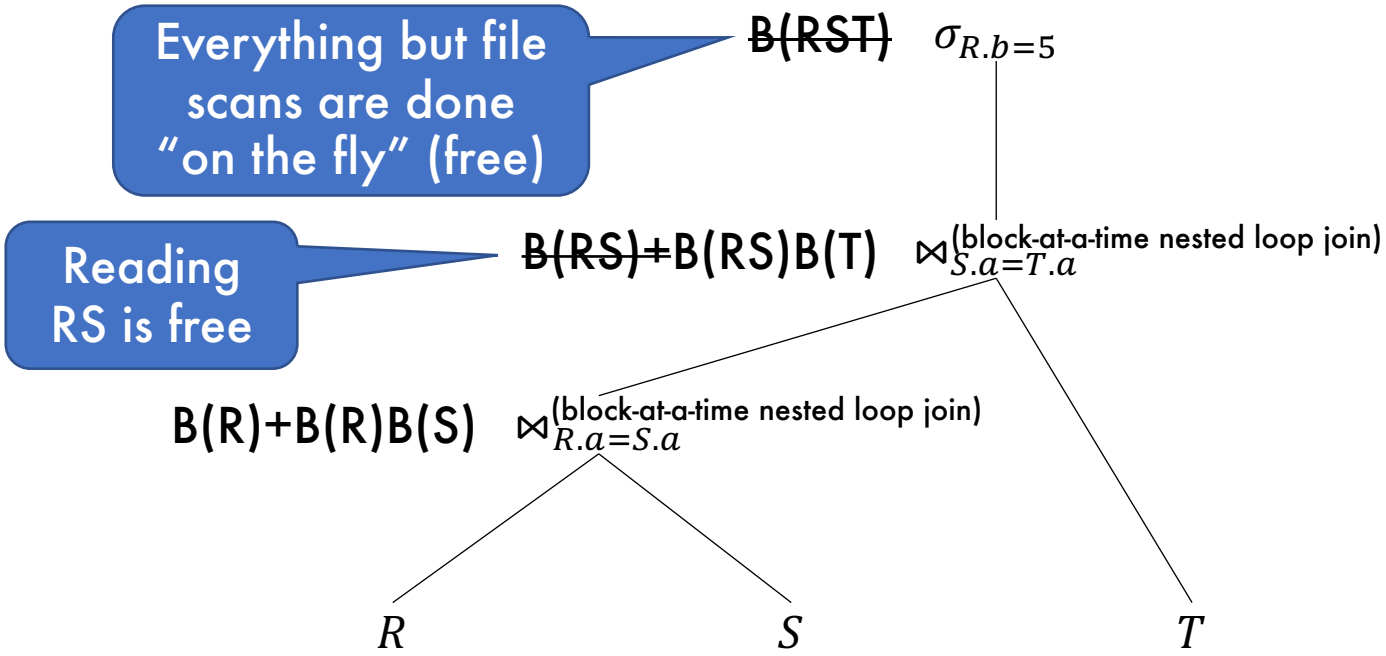
Full Plan Cost Estimation

- Estimated IO cost for a plan can be lowered under pipelined execution
- Generated tuples are in main memory so future access is "free"



Full Plan Cost Estimation

- Estimated IO cost for a plan can be lowered under pipelined execution
- Generated tuples are in main memory so future access is "free"



Full Plan Cost Estimation

- Estimated IO cost for a plan can be lowered under pipelined execution
- Generated tuples are in main memory so future access is "free"