Recap

- Data-savviness is the future!
- Notion of a DBMS
- The relational data model and algebra: bags and sets
- SQL Queries
- SQL Modifications
- SQL DDL
- Database Design
- Views
- Constraints and triggers
- Next: indexes

Index

Indexes

- Indexes help you look for data matching certain characteristics
  - Much like an index in a book! Or a library!

- An index on an attribute A allows you to find tuples with a specific value for A
  - Think of indexes as data structures or dictionaries storing "index key-value" pairs
    - Index key = values of A
    - Value = locations of tuples
    - Q: what is the correspondence for an index in a book?
  - This may be useful, for example, to find tuples with
    - A = a
    - A <= a
    - a<= A <= b

Why Indexes?

- Often your relations are pretty large
  - millions, billions, trillions of tuples
- Scanning through all the tuples to find those that match a certain condition can be time-consuming
  - SELECT * FROM Film WHERE rental_duration = 5 AND rental_rate = 2.99;
    - Naive approach: Go through all 10000 tuples in Film, verify if rental_duration and rental_rate are 5 and 2.99
    - Or: with an index on rental_duration: go through 100 tuples with rental_duration 5
    - Or: with a multi-dimensional index on rental_duration and rental_rate, go through 10 tuples with values 5 and 2.99 respectively

Also Useful for Joins!

```
SELECT * FROM Film, FilmActor
WHERE Film.film_id = FilmActor.film_id;
```

- Say Film has 10,000 tuples and FilmActor has 50,000 tuples
  - Roughly 5 actors per film (say)
- Q: If you have no indexes, need to consider how many pairs of tuples?
  - 10,000 * 50,000 = 5 * 10^8
- Q: How would you use indexes here?
  - For each film in Film, look up tuples in FilmActor, or vice versa
  - Goes from 5 * 10^8 to 5 * 10^4 tuples considered

Also Useful for Joins!

```
SELECT * FROM Film, FilmActor
WHERE Film.film_id = FilmActor.film_id
AND rental_duration = 5 AND rental_rate = 2.99
```

- Say we had indexes for FilmActor.film_id, Film.film_id, and Film.rental_duration. Q: How would you use indexes here?
  - Lookup films with rental_duration 5 = 100 tuples
  - For each one, look up FilmActor = 100 * 5 tuples matched
  - Verify if rental_rate = 2.99

Declaring Indexes

CREATE INDEX rentIndex ON Film (rental_duration)

- Allow for lookups based on rental_duration

CREATE INDEX rentReleaseIndex ON Film (rental_duration, release_year)

- Imagine this to be a "index key-value" pair dictionary based on concatenating attribute values
- Allow for lookups based on pairs of rental_duration and release_year
- Or just for rental_duration
- Q: Why not release_year only?

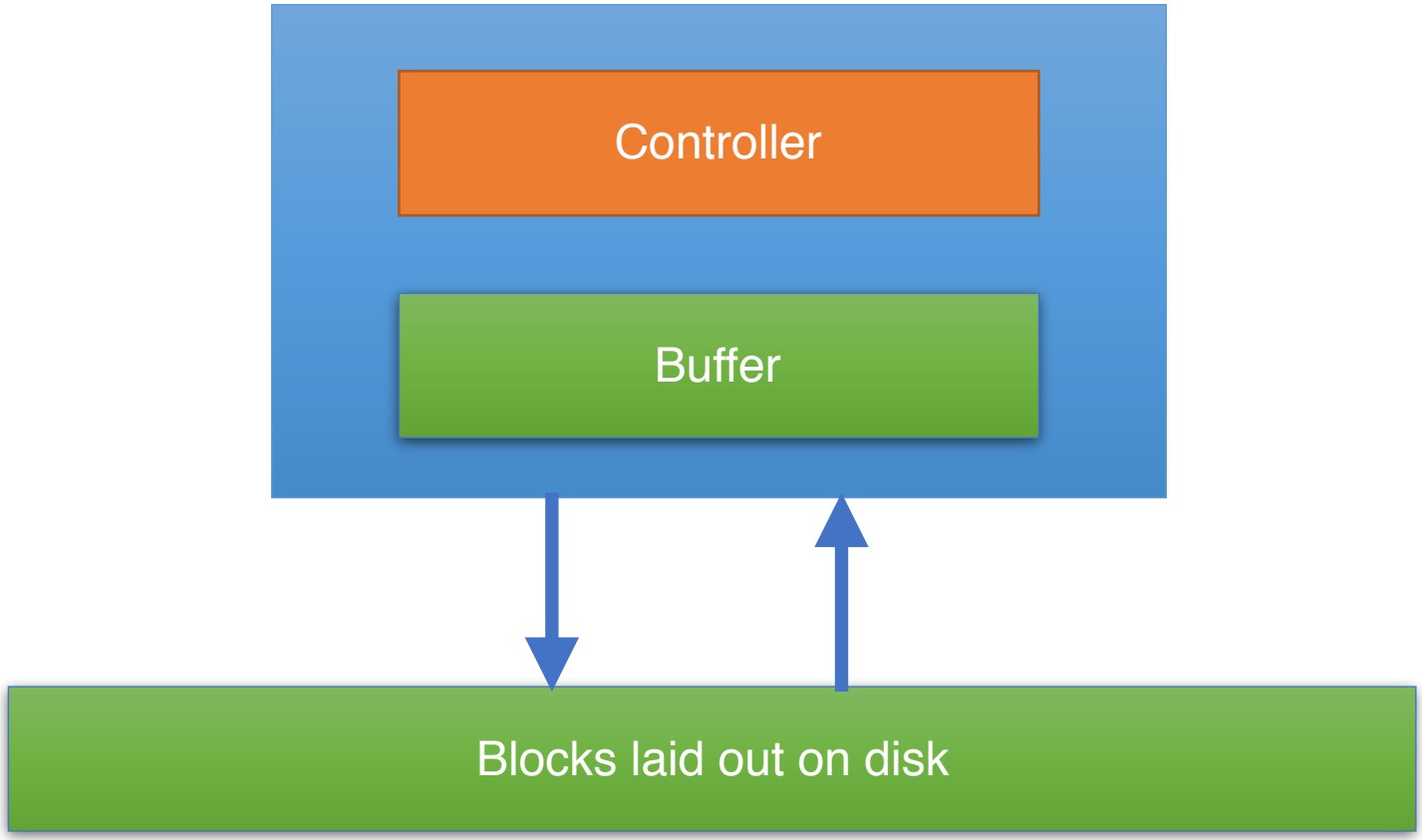
DROP INDEX rentReleaseIndex

Indexes Seem Magical…

- Q: what are downsides of indexes?
- A: the maintenance costs during updates, deletes, inserts

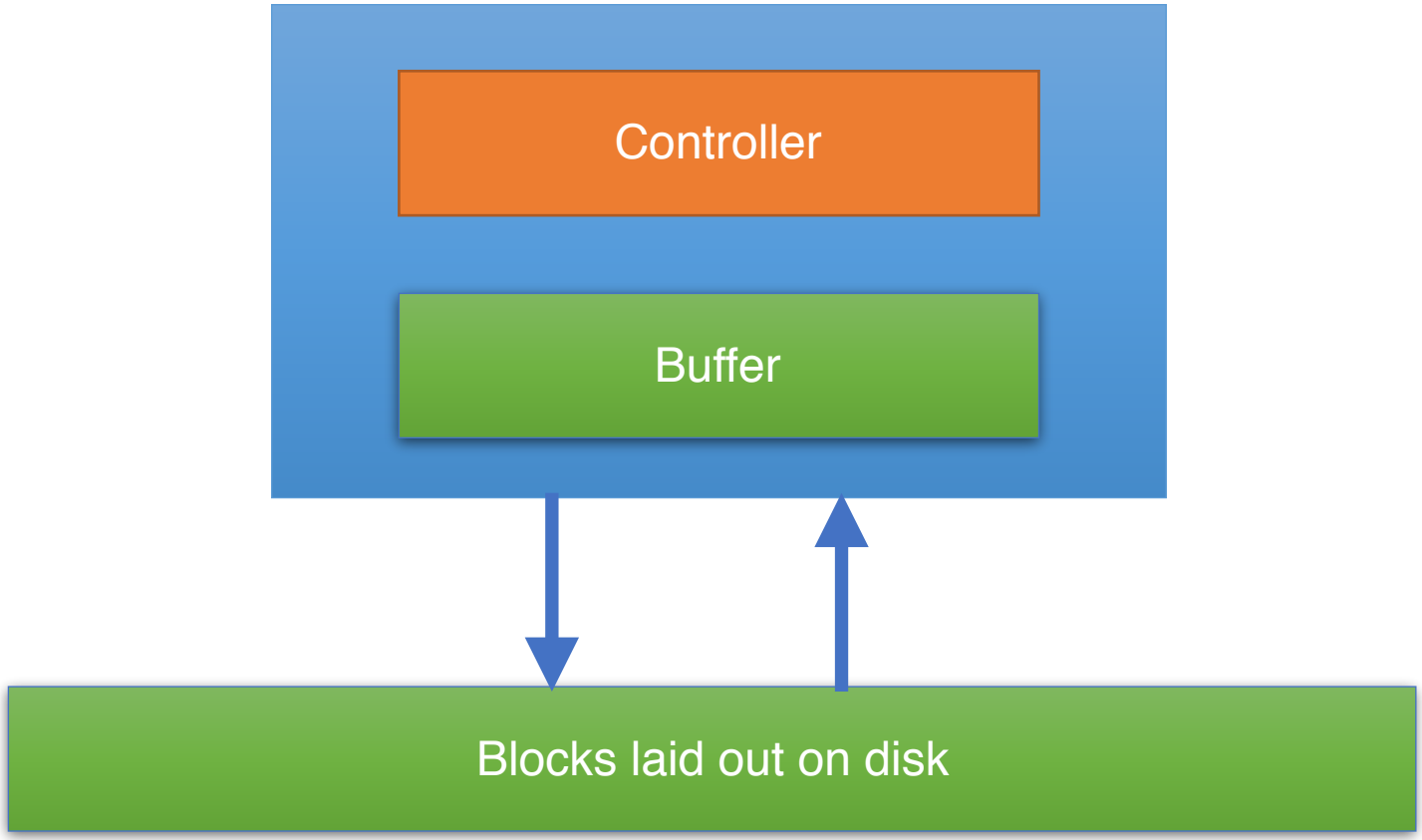
- If an index is not maintained, it may lead to inaccurate query results
- e.g., if a new tuple is inserted but it is not reflected in the index, it won't be returned as part of queries

Aside: A Mental Model for a Database



- Data is laid out on blocks/ pages on stable storage (e.g., disk)
- Each block may have many 100s-1000s of tuples
- Data is retrieved a block at a time
- Stored in main-memory buffers
- Written back to stable storage

Aside: A Mental Model for a Database



- Usually, computation is cheap relative to I/O
- So main cost in SQL queries is the cost of fetching the blocks and writing them out

OK, Going Back to Indexes…

- Which indexes should we create?
- Answer: it depends!

- Many databases automatically create indexes for PRIMARY KEY or UNIQUE attributes. Several reasons:
  - It also helps enforcing the constraints! Updating the index can serve as a check for the constraint
  - Many queries lookup based on these attributes
    - SELECT * FROM Film, FilmActor WHERE Film.film_id = FilmActor.film_id
  - It is a high-cardinality attribute (as many values as number of tuples!)

Why does cardinality matter?

- Databases retrieve data as "blocks"
- Say you are indexing based on attribute A, with 100 distinct values. You have 10,000 tuples. Each block contains 100 tuples.
- Q: How many blocks do we have?
- A: 100 blocks

- Assuming that the tuples with a given value of A (A = a) are randomly distributed, every block, on average, every block has one tuple with A=a.
- Thus, we can't avoid looking at all blocks => the index doesn't help.
- Q: What if we had a 1000 distinct values instead of 100? How many blocks would we have to look at then?
- A: 10 blocks, so there may be some savings.

Why does cardinality matter?

- Databases retrieve data as "blocks"
- Say you are indexing based on attribute A, with 100 distinct values. You have 10,000 tuples. Each block contains 100 tuples.

- However, if you know anything about disks, you will know that reading sequentially is much much faster than jumping across the disk
  - Even reading 1 in 10 blocks may not be enough to justify gains
  - Usually, 1 in 100 is enough to justify gains

- So, indexes on low cardinality attributes are not a great idea
- Exception: if data is "clustered" on that attribute
  - If we knew that all values of A = a are together (as opposed to randomly shuffled), we can simply read one block

How Do We Decide?

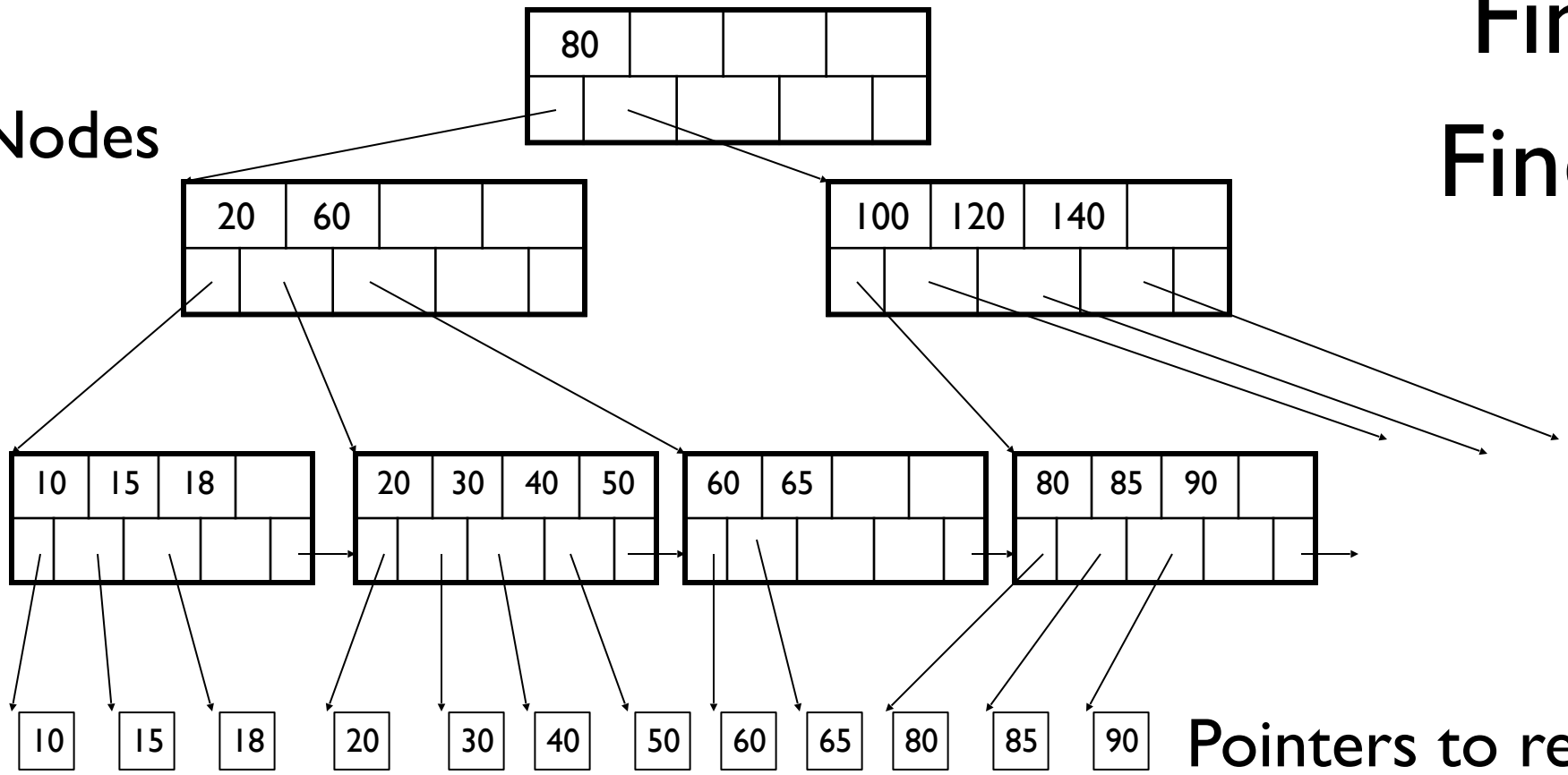
- Good to create attributes used often in WHERE clauses
- Good to create on key attributes (but usually done by default)
- Good on attributes that are "clustered"
- Not very good on attributes where there are many tuples per value of attribute
- Not very good on tables that are modified more often than queried

Types of Indexes: B+ Tree

- The most popular type of index in databases are B+Trees
- A generalization of binary trees
- Properties:
  - High fanout
    - Each "node" is a block
    - If "b" is fanout is ~1000, b^4 = 10^12
  - Self-balancing
    - Maintains a certain fill-factor (usually half) by splitting nodes and percolating changes to maintain balance
    - We won't worry about those algorithms in this class

At least halfway filled, except root

Nodes



Find A = 30

Find A = [30,63]

Pointers to records

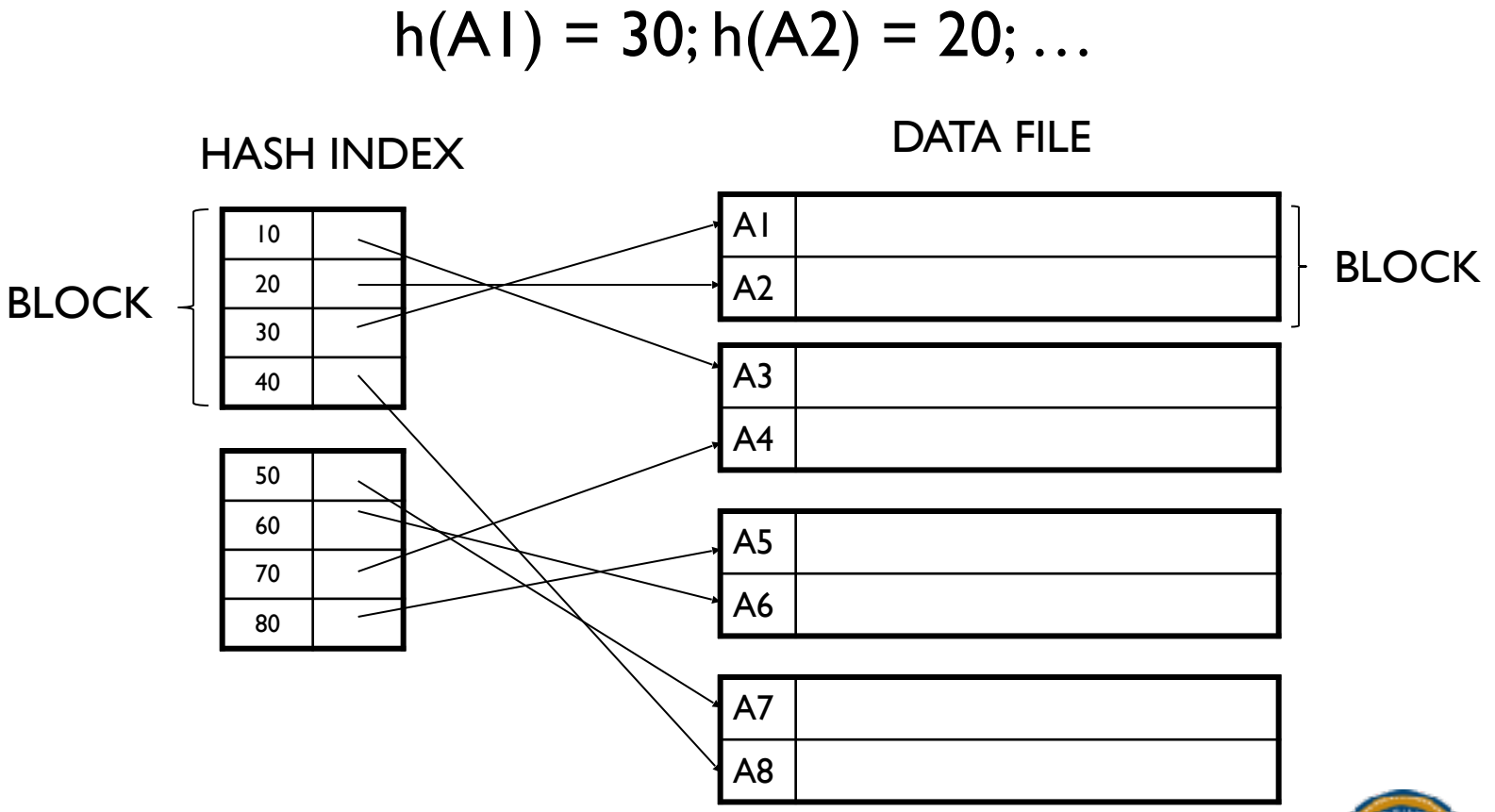
Types of Indexes: B+ Tree

- Logarithmic complexity
  - For searching (as we saw)
  - For insert, delete, update (not covered)
- In practice, depth of tree = number of nodes (blocks) retrieved is small

- Can handle both value lookups as well as range lookups
- In PostgreSQL this is the default index type

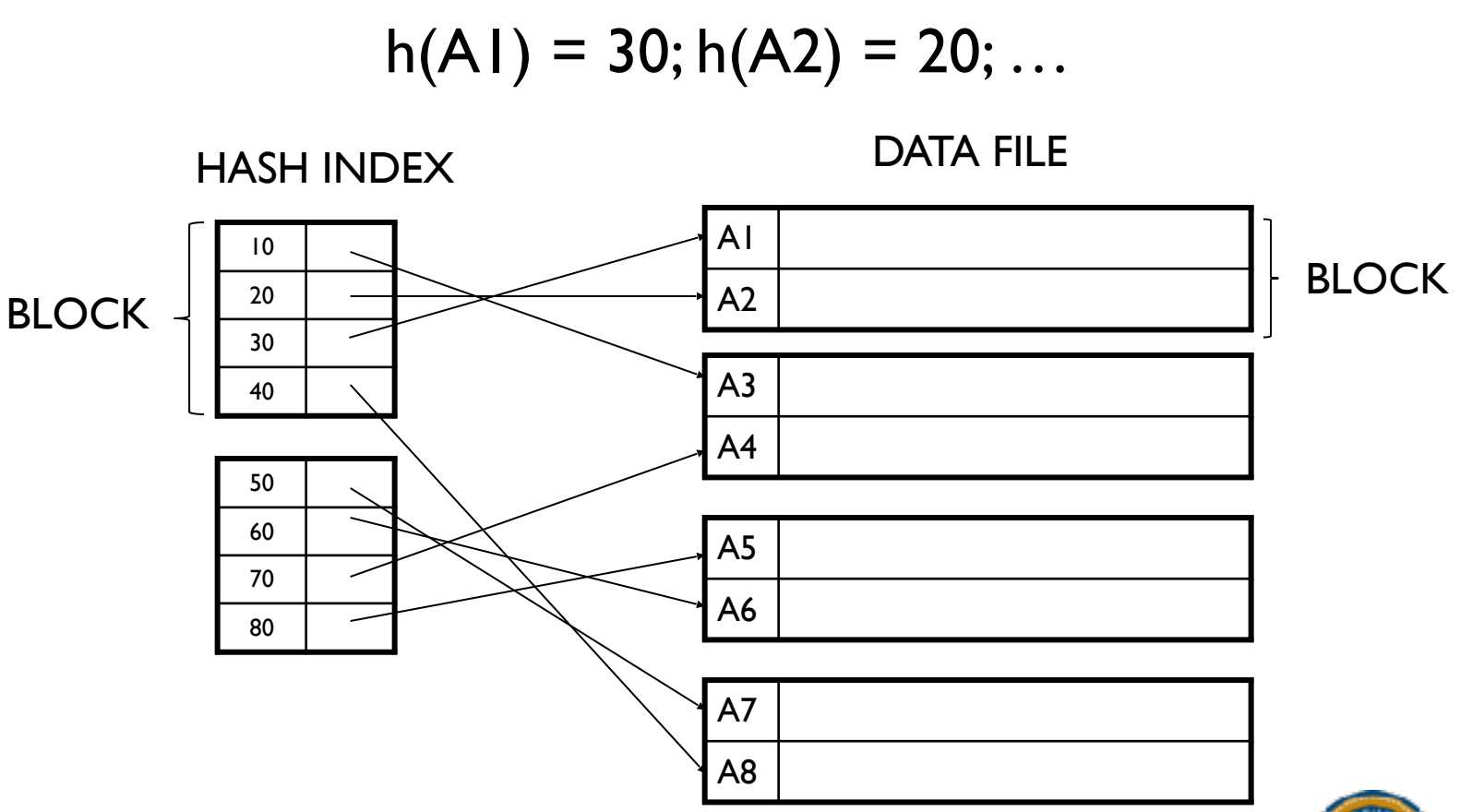
Types of Indexes: Hash Indexes

- Like hash-based dictionaries or maps in programming languages
  - A hash function applied to value a, i.e., h(a) yields location where pointers for tuples with A = a are found
- Main challenge: how to "extend" the hash table as values of A grow
  - See linear or extendible hashing



Types of Indexes: Hash Indexes

- Constant complexity O(1)
- Can only handle value lookups, not range lookups
  - Why?
- In PostgreSQL done as follows:
  - CREATE INDEX <name> ON <table> USING hash (column);



Other Types of Indexes

- Inverted Indexes: valuable for keyword search (more later, hopefully!)
  - In PostgreSQL, known as GIN indexes
- R-Tree: valuable for range-based lookups in k-dimensions
  - e.g., searching for things in rectangular range in a map
  - In PostgreSQL, a special case of a GIST index
- Lots of other types of indexes!
  - Partial indexes
  - Indexes on expressions
  - …

Indexes Summary

- Indexes are great!
- They speed up certain types of queries
- But we should use them carefully
  - Only on high cardinality = "selective" attributes like keys
  - Or on frequently queried but rarely updated attributes
- Lots of different types of indexes: but B+trees are most popular; hash indexes are a close second