# COMP7106B Spring 2024
# Big Data Management
## Lecture 1 Database Review

Prof. Reynold Cheng

20th January, 2024

*Acknowledgement: This presentation is based on Prof. Nikos Mamoulis' COMP3323 slides.*

# Part A

## Database Design and Queries

# Background on relational databases

- Structure of Relational Databases
- Reduction of an E-R Schema to Tables
- Relational Algebra and SQL
- Storage of relations

# Basic Structure of Relational Databases

- Formally, given sets $D_1$, $D_2$, …. $D_n$ a **relation** $r$ is a subset of $D_1 \times D_2 \times \ldots \times D_n$
  Thus a relation is a **set** of n-tuples ($a_1$, $a_2$, …, $a_n$) where each $a_i \in D_i$
- Example: if

  *customer-name* = {Jones, Smith, Curry, Lindsay}
  *customer-street* = {Main, North, Park}
  *customer-city* = {Harrison, Rye, Pittsfield}

  Then *r* = { (Jones, Main, Harrison),
  (Smith, North, Rye),
  (Curry, North, Rye),
  (Lindsay, Park, Pittsfield)}

  is a relation over *customer-name* x *customer-street* x *customer-city*

# Attribute Types

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Examples of simple domain types:
  - integer
  - string
  - date
- NOTE: In advanced (non-relational) database systems, we may have complex attribute types.
- E.g., in spatial databases:
  - point
  - polygon
  - poly-line

# Relation Schema

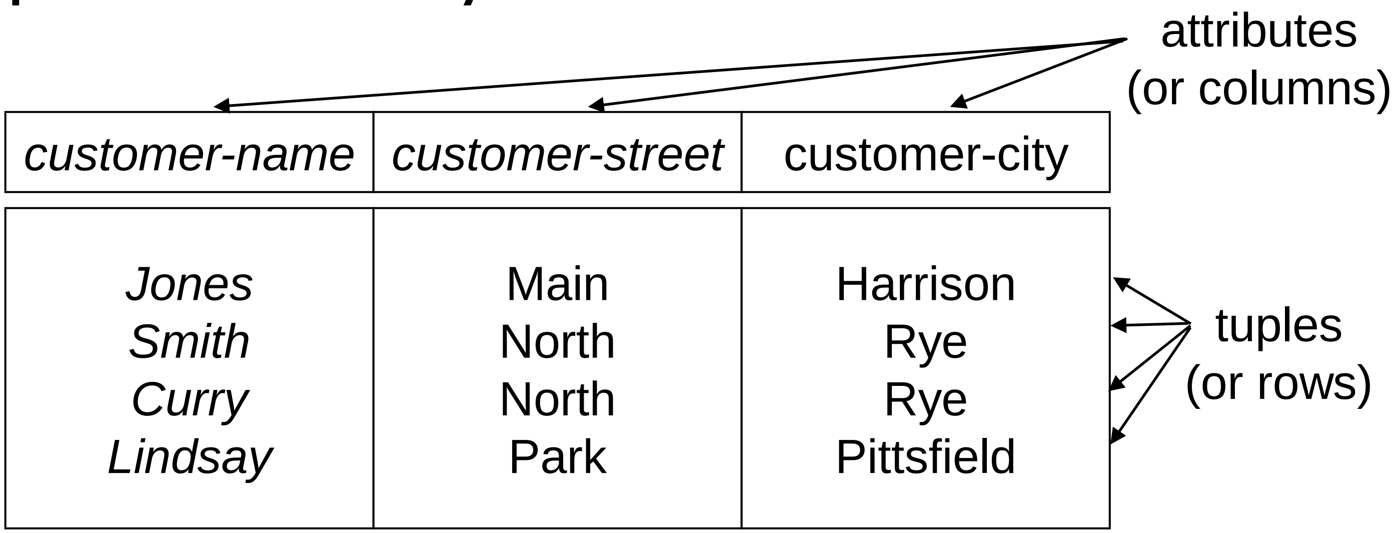
- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*

  E.g. *Account-schema* = (*account-number, branch-name, balance*)
- *r*(*R*) is a *relation* on the *relation schema R*

  E.g. *customer (Customer-schema)*

# Relation Instance

- The current values (*relation instance*) of a relation are specified by a table
- An element *t* of *r* is a *tuple*, represented by a *row* in a table



attributes
(or columns)

| *customer-name* | *customer-street* | customer-city |
|---|---|---|
| *Jones* | Main | Harrison |
| *Smith* | North | Rye |
| *Curry* | North | Rye |
| *Lindsay* | Park | Pittsfield |

tuples
(or rows)

*customer*

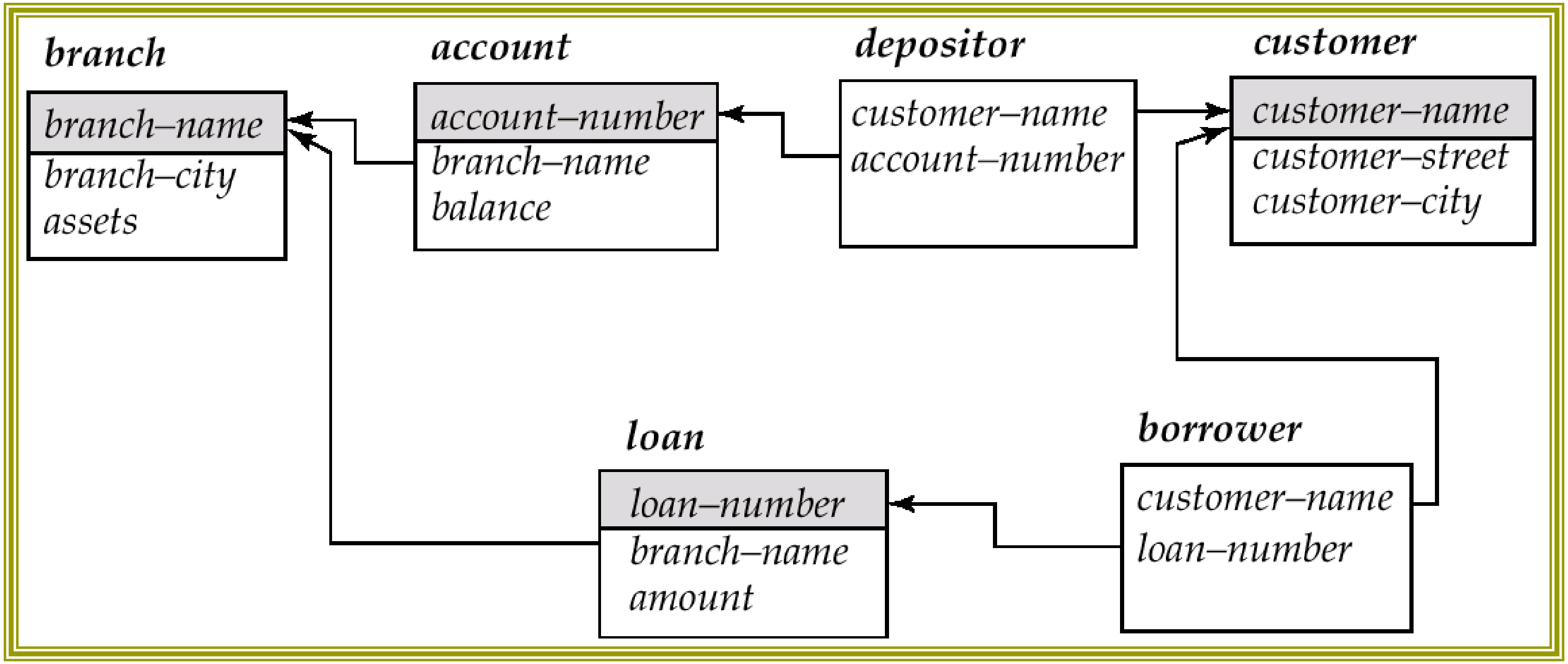
# Database

- A database consists of multiple relations which are inter-related
- Information about an enterprise is broken up into parts, with each relation storing one part of the information

  E.g.:

  - *account* : stores information about accounts
  - *depositor* : stores information about which customer owns which account
  - *customer* : stores information about customers

# Schema Diagram for a Banking Enterprise



branch

branch–name
branch–city
assets

account

account–number
branch–name
balance

depositor

customer–name
account–number

customer

customer–name
customer–street
customer–city

loan

loan–number
branch–name
amount

borrower

customer–name
loan–number

# Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - procedural
  - non-procedural
- “Pure” languages:
  - Relational Algebra
  - Relational Calculus
- Pure languages form underlying basis of query languages that people use.

# Relational Algebra

- Procedural language
- Six basic operators
  - select, project, union, set difference, Cartesian product, rename
- Additional operations
  - set intersection, natural join, θ-join, division, generalized projection, assignment, aggregation, outer joins, ...
- The operators take one or more relations as inputs and give a new relation as a result.
- In Relational Algebra all inputs and outputs are relations. They are sets of tuples.
  - In SQL the output of an operator is a multiset of tuples

# Select Operation – Example

- Relation *r*

| *A* | *B* | *C* | *D* |
|---|---|---|---|
| $\alpha$ | $\alpha$ | *1* | *7* |
| $\alpha$ | $\beta$ | *5* | *7* |
| $\beta$ | $\beta$ | *12* | *3* |
| $\beta$ | $\beta$ | *23* | *10* |

- $\sigma_{A=B \wedge D > 5}\,(r)$

| *A* | *B* | *C* | *D* |
|---|---|---|---|
| $\alpha$ | $\alpha$ | *1* | *7* |
| $\beta$ | $\beta$ | *23* | *10* |

# Select Operation

- Notation: $\sigma_{p}(r)$
- *p* is called the selection predicate
- Defined as:

$$\sigma_p(\boldsymbol{r}) = \{t \mid t \in r \textbf{ and } p(t)\}$$

  Where *p* is a formula in propositional calculus consisting of terms connected by : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
  Each term is one of:

  <attribute> *op* <attribute> or <constant>

  where *op* is one of: $=, \neq, >, \geq. <. \leq$

- Example of selection:

  $\sigma_{\textit{branch-name}=\text{“Perryridge”}}(\textit{account})$

# Project Operation – Example

- Relation *r*:

| A | B | C |
|---|---|---|
| α | 10 | 1 |
| α | 20 | 1 |
| β | 30 | 1 |
| β | 40 | 2 |

- $\Pi_{A,C}(r)$

| A | C |
|---|---|
| α | 1 |
| α | 1 |
| β | 1 |
| β | 2 |

=

| A | C |
|---|---|
| α | 1 |
| β | 1 |
| β | 2 |

# Project Operation

- Notation:

  $$\Pi_{A1,\ A2,\ \ldots,\ Ak}\ (r)$$

  where $A_1$, $A_2$ are attribute names and $r$ is a relation name.
- The result is defined as the relation of $k$ columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- E.g. To eliminate the *branch-name* attribute of *account*

  $$\Pi_{account\text{-}number,\ balance}\ (account)$$

# Set-union Operation – Example

- Relations *r, s:*

| *A* | *B* |
|---|---|
| $\alpha$ | 1 |
| $\alpha$ | 2 |
| $\beta$ | 1 |

*r*

| *A* | *B* |
|---|---|
| $\alpha$ | 2 |
| $\beta$ | 3 |

*s*

r ∪ s:

| *A* | *B* |
|---|---|
| $\alpha$ | 1 |
| $\alpha$ | 2 |
| $\beta$ | 1 |
| $\beta$ | 3 |

# Union Operation

- Notation $r \cup s$
- Defined as:

  $r \cup s = \{t \mid t \in r \textbf{ or } t \in s\}$

- Union must be taken between *compatible* relations.
  - *r* and *s* must have the *same arity*
  - attribute domains of *r* and *s* must be compatible

# Set-Intersection Operation - Example

- Relation r, s:

| A | B |
|---|---|
| α | 1 |
| α | 2 |
| β | 1 |

*r*

| A | B |
|---|---|
| α | 2 |
| β | 3 |

*s*

- r ∩ s

| A | B |
|---|---|
| α | 2 |

# Intersection Operation

- Notation $r \cap s$
- Defined as:

  $r \cap s = \{t \mid t \in r \textbf{ and } t \in s\}$
- Intersection must be taken between *compatible* relations.
  - *r* and *s* must have the *same arity*
  - attribute domains of *r* and *s* must be compatible

# Set Difference Operation – Example

- Relations *r, s:*

| *A* | *B* |
|---|---|
| $\alpha$ | *1* |
| $\alpha$ | *2* |
| $\beta$ | *1* |

*r*

| *A* | *B* |
|---|---|
| $\alpha$ | *2* |
| $\beta$ | *3* |

*s*

*r* – *s*:

| *A* | *B* |
|---|---|
| $\alpha$ | *1* |
| $\beta$ | *1* |

# Set Difference Operation

- Notation $r - s$
- Defined as:

  $r - s = \{t \mid t \in r$ **and** $t \notin s\}$

- Set differences must be taken between *compatible* relations.
  - *r* and *s* must have the *same arity*
  - attribute domains of *r* and *s* must be compatible

# Cartesian-Product Operation-Example

Relations *r*, *s*:

| A | B |
|---|---|
| α | 1 |
| β | 2 |

*r*

| C | D | E |
|---|---|---|
| α | 10 | a |
| β | 10 | a |
| β | 20 | b |
| γ | 10 | b |

*s*

*r* x *s*:

| A | B | C | D | E |
|---|---|---|---|---|
| α | 1 | α | 10 | a |
| α | 1 | β | 10 | a |
| α | 1 | β | 20 | b |
| α | 1 | γ | 10 | b |
| β | 2 | α | 10 | a |
| β | 2 | β | 10 | a |
| β | 2 | β | 20 | b |
| β | 2 | γ | 10 | b |

# Cartesian-Product Operation

- Notation *r* x *s*
- Defined as:

  *r* x *s* = {*t q* | *t* ∈ *r* **and** *q* ∈ *s*}

# Composition of Operations

- Can build expressions using multiple operations
- Example:
    - $\sigma_{A=C}(r\ x\ s)$

$r \times s$

| A | B | C | D | E |
|---|---|---|---|---|
| $\alpha$ | 1 | $\alpha$ | 10 | a |
| $\alpha$ | 1 | $\beta$ | 10 | a |
| $\alpha$ | 1 | $\beta$ | 20 | b |
| $\alpha$ | 1 | $\gamma$ | 10 | b |
| $\beta$ | 2 | $\alpha$ | 10 | a |
| $\beta$ | 2 | $\beta$ | 10 | a |
| $\beta$ | 2 | $\beta$ | 20 | b |
| $\beta$ | 2 | $\gamma$ | 10 | b |

$\sigma_{A=C}(r \times s)$

| A | B | C | D | E |
|---|---|---|---|---|
| $\alpha$ | 1 | $\alpha$ | 10 | a |
| $\beta$ | 2 | $\beta$ | 20 | a |
| $\beta$ | 2 | $\beta$ | 20 | b |

# Natural Join Operation

- Notation: $r \bowtie s$

- Let *r* and *s* be relations with schemas *R* and *S* respectively.
  Then, $r \bowtie s$ is a relation with schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples *tr* from *r* and *ts* from *s*.
  - If *tr* and *ts* have the same value on each of the attributes in $R \cap S$, add a tuple *t* to the result, where
    - *t* has the same value as *tr* on *r*
    - *t* has the same value as *ts* on *s*

# Natural Join Operation

- Example:
  - $R = (A, B, C, D)$
  - $S = (E, B, D)$
  - Result schema = $(A, B, C, D, E)$
  - $r \bowtie s$ is defined as
    - $\Pi_{r.A,\ r.B,\ r.C,\ r.D,\ s.E}\ (\sigma_{r.B\ =\ s.B\ \wedge\ r.D\ =\ s.D}\ (r\ \mathrm{x}\ s))$

# Natural Join Operation – Example

- Relations r, s:

| A | B | C | D |
|---|---|---|---|
| α | 1 | α | a |
| β | 2 | γ | a |
| γ | 4 | β | b |
| α | 1 | γ | a |
| δ | 2 | β | b |

r

| B | D | E |
|---|---|---|
| 1 | a | α |
| 3 | a | β |
| 1 | a | γ |
| 2 | b | δ |
| 3 | b | ∈ |

s

$r \bowtie s$

| A | B | C | D | E |
|---|---|---|---|---|
| α | 1 | α | a | α |
| α | 1 | α | a | γ |
| α | 1 | γ | a | α |
| α | 1 | γ | a | γ |
| δ | 2 | β | b | δ |

# Natural Join Operation – Example

- Relation *loan*

| *loan-number* | *branch-name* | *amount* |
|---|---|---|
| L-170 | Downtown | 3000 |
| L-230 | Redwood | 4000 |
| L-260 | Perryridge | 1700 |

n Relation *borrower*

| *customer-name* | *loan-number* |
|---|---|
| Jones | L-170 |
| Smith | L-230 |
| Hayes | L-155 |

- *Relation loan* $\bowtie$ *borrower*

| *loan-number* | *branch-name* | *amount* | *customer-name* |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |

# Aggregate Functions and Operations

- **An** **aggregation function** takes a collection of values and returns a single value as a result.

  **avg**: average value<br>
  **min**: minimum value<br>
  **max**: maximum value<br>
  **sum**: sum of values<br>
  **count**: number of values

# Aggregate Functions and Operations

- **Aggregate operation** in relational algebra
  - $_{G1, G2, \ldots, Gn}\ \mathcal{g}\ _{F1(A1), F2(A2), \ldots, Fn(An)}\ (E)$
- $E$ is any relational-algebra expression
- $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
- Each $F_i$ is an aggregate function
- Each $A_i$ is an attribute name

# Aggregate Operation – Example

- Relation *r*:

| *A* | *B* | *C* |
|---|---|---|
| $\alpha$ | $\alpha$ | 7 |
| $\alpha$ | $\beta$ | 7 |
| $\beta$ | $\beta$ | 3 |
| $\beta$ | $\beta$ | 10 |

$\mathcal{G}_{\mathbf{sum(c)}}(\mathrm{r})$

| *sum-C* |
|---|
| 27 |

# Aggregate Operation – Example

- Relation *account* grouped by *branch-name*:

| *branch-name* | *account-number* | *balance* |
|---|---|---|
| Perryridge | A-102 | 400 |
| Perryridge | A-201 | 900 |
| Brighton | A-217 | 750 |
| Brighton | A-215 | 750 |
| Redwood | A-222 | 700 |

$$_{\textit{branch-name}}\,\mathcal{G}_{\,\textit{sum(balance)}}\,(\textit{account})$$

| *branch-name* | *balance* |
|---|---|
| Perryridge | 1300 |
| Brighton | 1500 |
| Redwood | 700 |

# SQL: Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

  **select** $A_1, A_2, \ldots, A_n$
  **from** $r_1, r_2, \ldots, r_m$
  **where** $P$

  - $A_i$*s* represent attributes
  - $r_i$*s* represent relations
  - $P$ is a predicate.
- This query is equivalent to the relational algebra expression.

$$\Pi_{A1,\, A2,\, \ldots,\, An}(\sigma_P\, (r_1 \;\; \text{x}\; r_2 \;\; \text{x} \;\; \ldots \;\; \text{x} \;\; r_m))$$

- The result of an SQL query is a multiset of tuples.

# The select Clause

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after **select.**
- Q1: Find the names of all branches in the *loan* relations, and remove duplicates

```
select distinct branch-name
from loan
```

- Q2: If distinct is not used duplicates are not removed.

```
select branch-name
from loan
```

**loan**

| *loan-number* | *branch-name* | *amount* |
|---|---|---|
| L-170 | Downtown | 3000 |
| L-220 | Downtown | 4000 |
| L-260 | Perryridge | 1700 |

**Q2**

| *branch-name* |
|---|
| Downtown |
| Downtown |
| Perryridge |

**Q1**

| *branch-name* |
|---|
| Downtown |
| Perryridge |

# The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

  **select** *loan-number*
  **from** *loan*
  **where** *branch-name* = ‘Perryridge’ **and** *amount* > 1200
- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

# The from Clause

- The **from** clause lists the relations involved in the query
  - corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower x loan*

```
select *
from borrower, loan
```

n To express a join in SQL, the tables should be included in the *from* list and the join conditions should appear in the *where list*. Recall that a join can be expressed by a Cartesian product followed by selections on the join attributes

n Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

```
select customer-name, borrower.loan-number, amount
        from borrower, loan
        where   borrower.loan-number = loan.loan-number  and
            branch-name = ‘Perryridge’
```

# Aggregate Functions – Group By

- Find the number of depositors for each branch.

```
select branch-name, count (distinct customer-name)
        from depositor, account
        where depositor.account-number = account.account-number
        group by branch-name
```

Note: In the **select** clause outside of aggregate functions we must have:

- attributes that appear in the **group by** list and/or
- aggregate functions on attributes of each group

# Aggregate Functions – Having Clause

- Find the names and average account balances of all branches where the average account balance is more than $1,200.

```
select branch-name, avg (balance)
        from account
        group by branch-name
        having avg (balance) > 1200
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

# Query Evaluation and Optimization

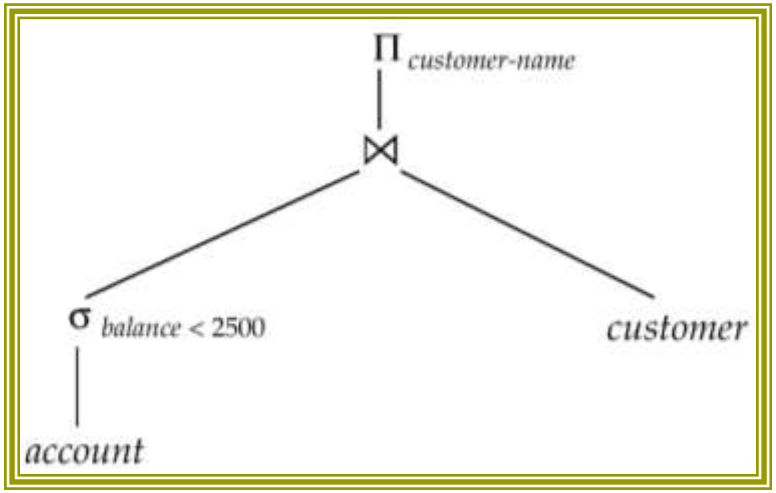
- Query evaluation
  - Basic operations
    - Selections
    - Joins
    - Other operations (projection, aggregation)
  - Transformation of queries into a tree of operations

*customer*(cid, customer-name,…)
*account*(account-number,cid,balance)

Compute

$\Pi_{\text{customer-name}}$

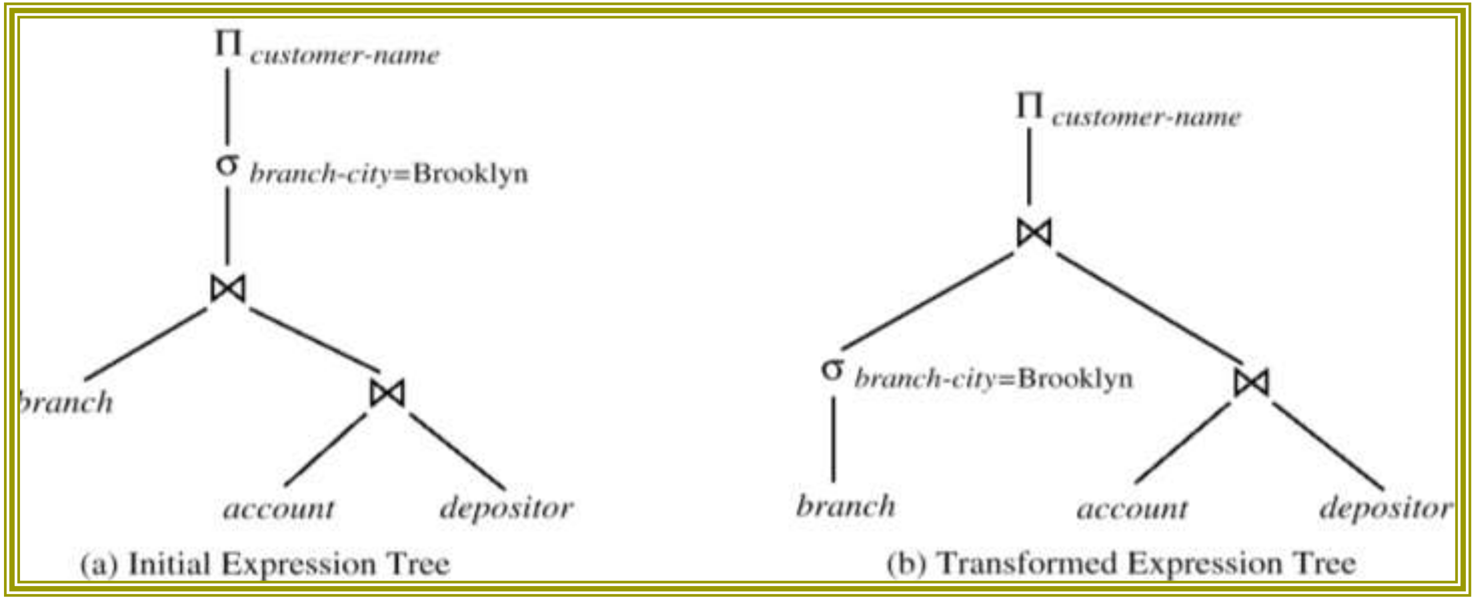
$(\sigma_{\text{balance} < 2\ 500}(\textit{account}) \bowtie \textit{customer})$



# Query Evaluation and Optimization

- Many equivalent expressions to the original query can be derived
- The query optimizer uses statistical data and appropriate algorithms to compute an expression of low evaluation cost.

$\Pi_{\text{customer-name}}(\sigma_{\text{branch-city = Brooklyn}}\ (\textit{branch} \bowtie \textit{account} \bowtie \textit{depositor}))$



# Storage of databases – Physical Storage Media

- **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.

A DDR3 RAM

- **Main memory**:
  - fast access
  - generally too small (or too expensive) to store the entire database
    - capacities of up to several Gigabytes widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
  - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

# Physical Storage Media (Cont.)

- **Magnetic-disk**
  - Data is stored on spinning disk, and read/written magnetically
  - Primary medium for the long-term storage of data; typically stores entire database.
  - Data must be moved from disk to main memory for access, and written back for storage
    - Much slower access than main memory
  - **direct-access** – possible to read data on disk in any order, unlike magnetic tape
  - Capacities range up to several TB currently
    - Much larger capacity and lower cost/byte than main memory/flash memory
    - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
  - Survives power failures and system crashes
    - disk failure can destroy data, but is very rare

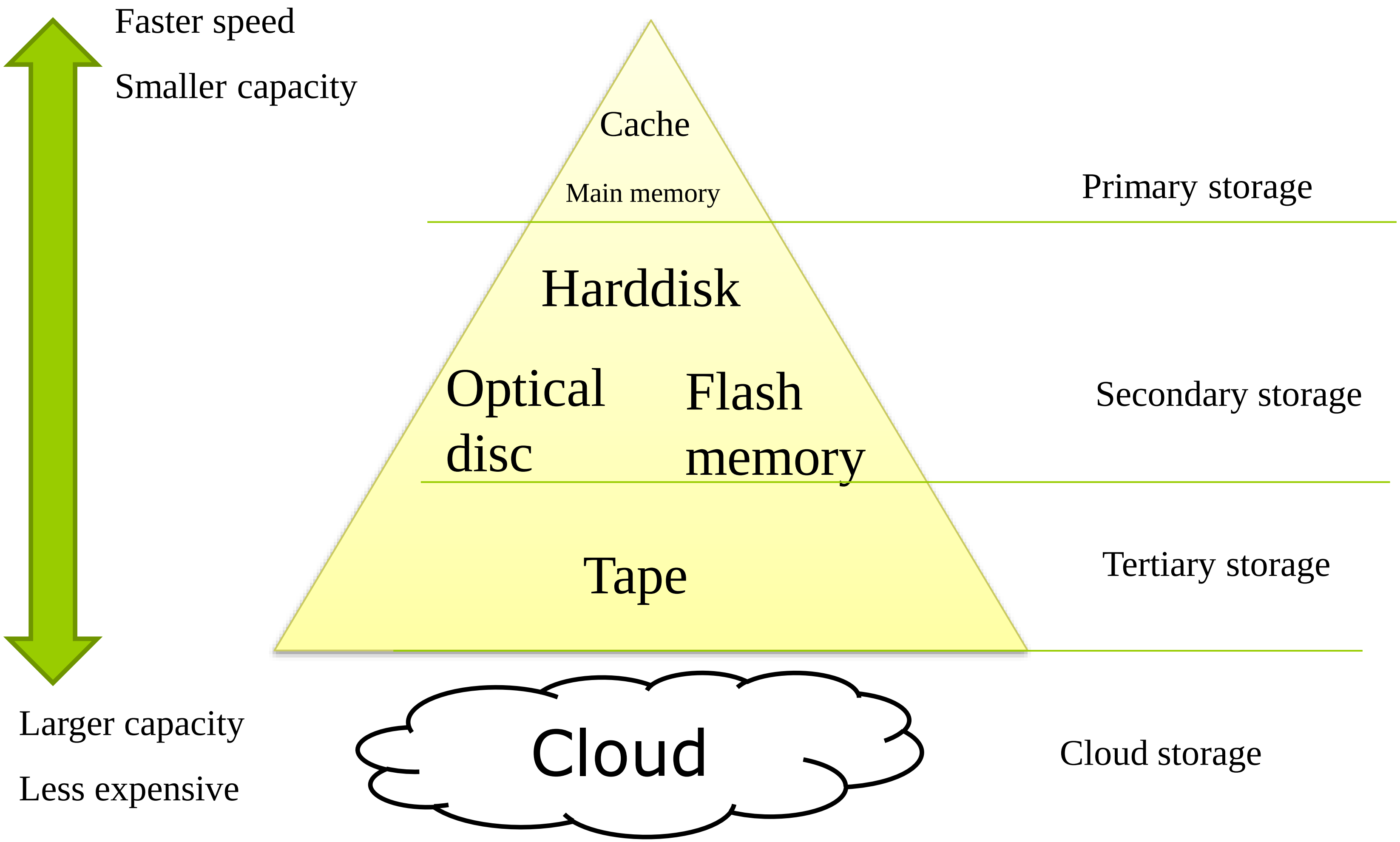
# Storage Hierarchy

- **primary storage**: Fastest media but volatile (cache, main memory).
- **secondary storage**: next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage**: lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage

IBM System Storage
TS1140 Tape Drive

Cartridge capacity : 4TB.

# Storage: memory hierarchy



Faster speed

Smaller capacity

Cache

Main memory

Primary storage

Harddisk

Optical disc

Flash memory

Secondary storage

Tape

Tertiary storage

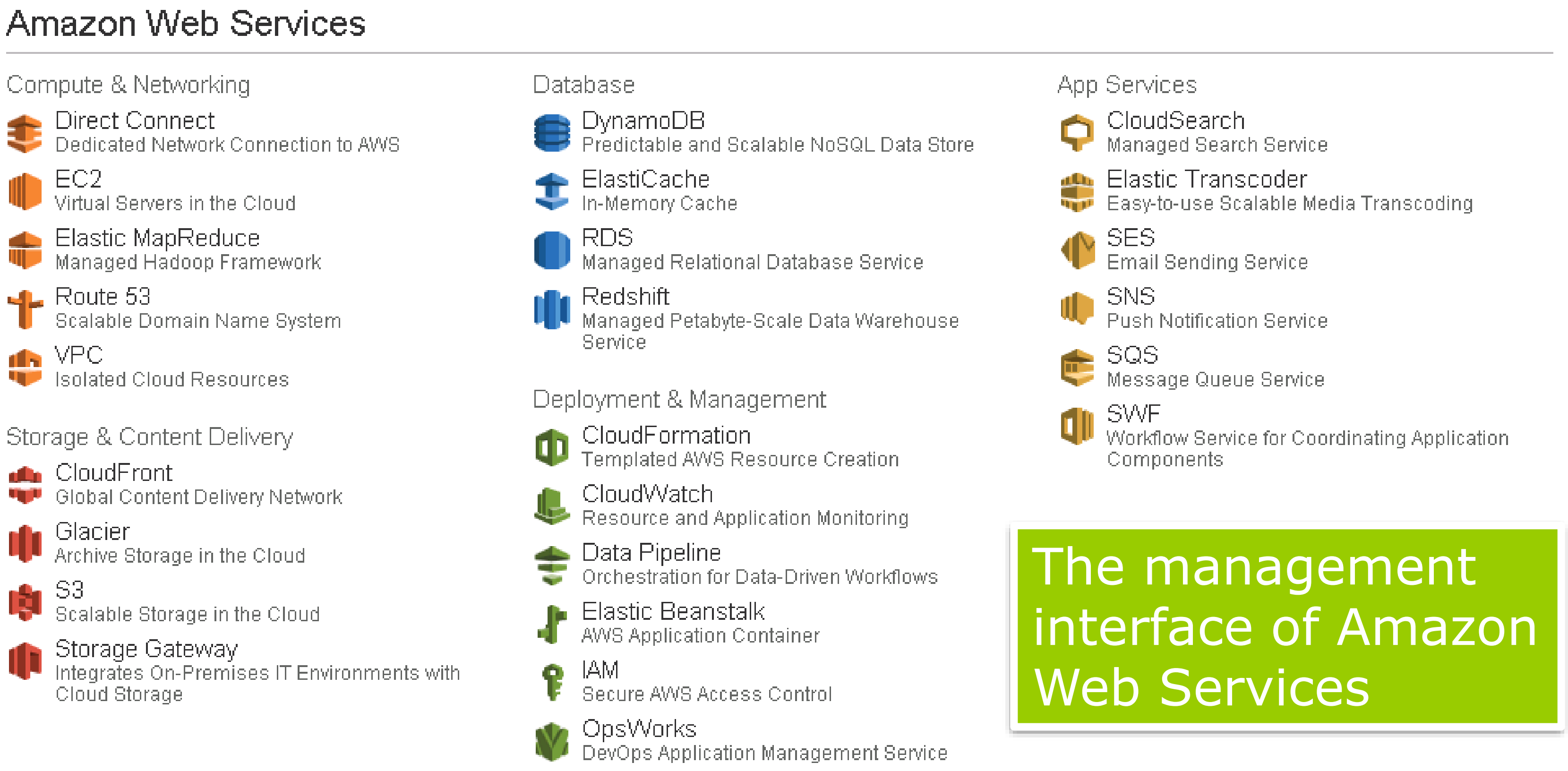
Cloud

Cloud storage

Larger capacity

Less expensive

# Example: Amazon Web Services



## Amazon Web Services

Compute & Networking

- Direct Connect
  Dedicated Network Connection to AWS
- EC2
  Virtual Servers in the Cloud
- Elastic MapReduce
  Managed Hadoop Framework
- Route 53
  Scalable Domain Name System
- VPC
  Isolated Cloud Resources

Storage & Content Delivery

- CloudFront
  Global Content Delivery Network
- Glacier
  Archive Storage in the Cloud
- S3
  Scalable Storage in the Cloud
- Storage Gateway
  Integrates On-Premises IT Environments with Cloud Storage

Database

- DynamoDB
  Predictable and Scalable NoSQL Data Store
- ElastiCache
  In-Memory Cache
- RDS
  Managed Relational Database Service
- Redshift
  Managed Petabyte-Scale Data Warehouse Service

Deployment & Management

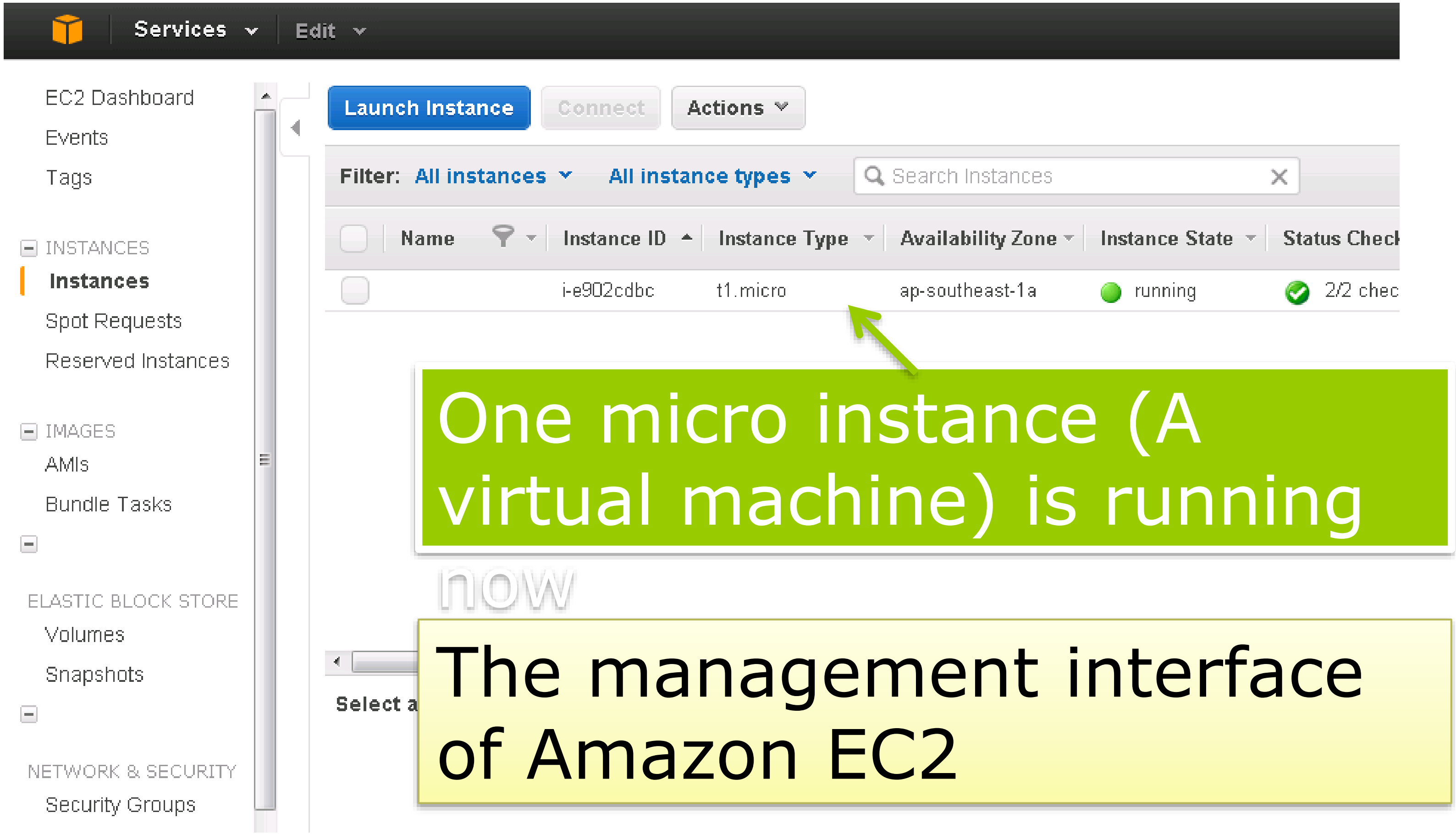
- CloudFormation
  Templated AWS Resource Creation
- CloudWatch
  Resource and Application Monitoring
- Data Pipeline
  Orchestration for Data-Driven Workflows
- Elastic Beanstalk
  AWS Application Container
- IAM
  Secure AWS Access Control
- OpsWorks
  DevOps Application Management Service

App Services

- CloudSearch
  Managed Search Service
- Elastic Transcoder
  Easy-to-use Scalable Media Transcoding
- SES
  Email Sending Service
- SNS
  Push Notification Service
- SQS
  Message Queue Service
- SWF
  Workflow Service for Coordinating Application Components

The management interface of Amazon Web Services

# Amazon EC2



| □ | Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Check |
|---|---|---|---|---|---|---|
| □ | | i-e902cdbc | t1.micro | ap-southeast-1a | ● running | ✓ 2/2 chec |

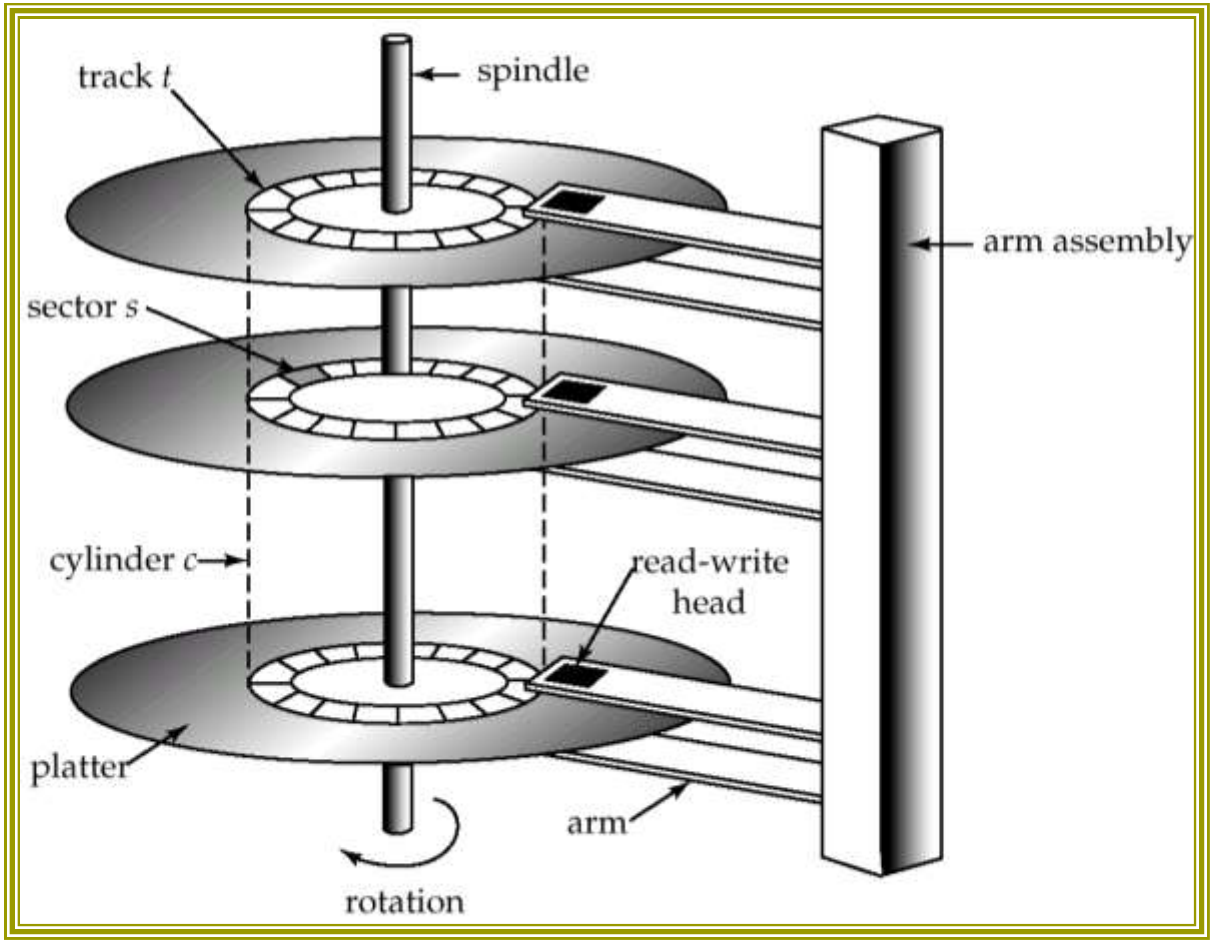
One micro instance (A virtual machine) is running now

The management interface of Amazon EC2

# Magnetic Disks

- **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
  - Over 16,000 tracks per platter on typical hard disks
- Each track is divided into **sectors.**
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 200 (on inner tracks) to 400 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (typically 2 to 4)
  - one head per platter, mounted on a common arm.
- **Cylinder** *i* consists of *i*th track of all the platters

# Magnetic Hard Disk Mechanism



**NOTE: Diagram is schematic, and simplifies the structure of actual disk drives**

# Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins.  Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - Average latency is 1/2 of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 4 to 8 MB per second is typical
  - Multiple disks may share a controller, so rate that controller can handle is also important
    - E.g. ATA-5: 66 MB/second,  SCSI-3: 40 MB/s
    - Fiber Channel: 256 MB/s

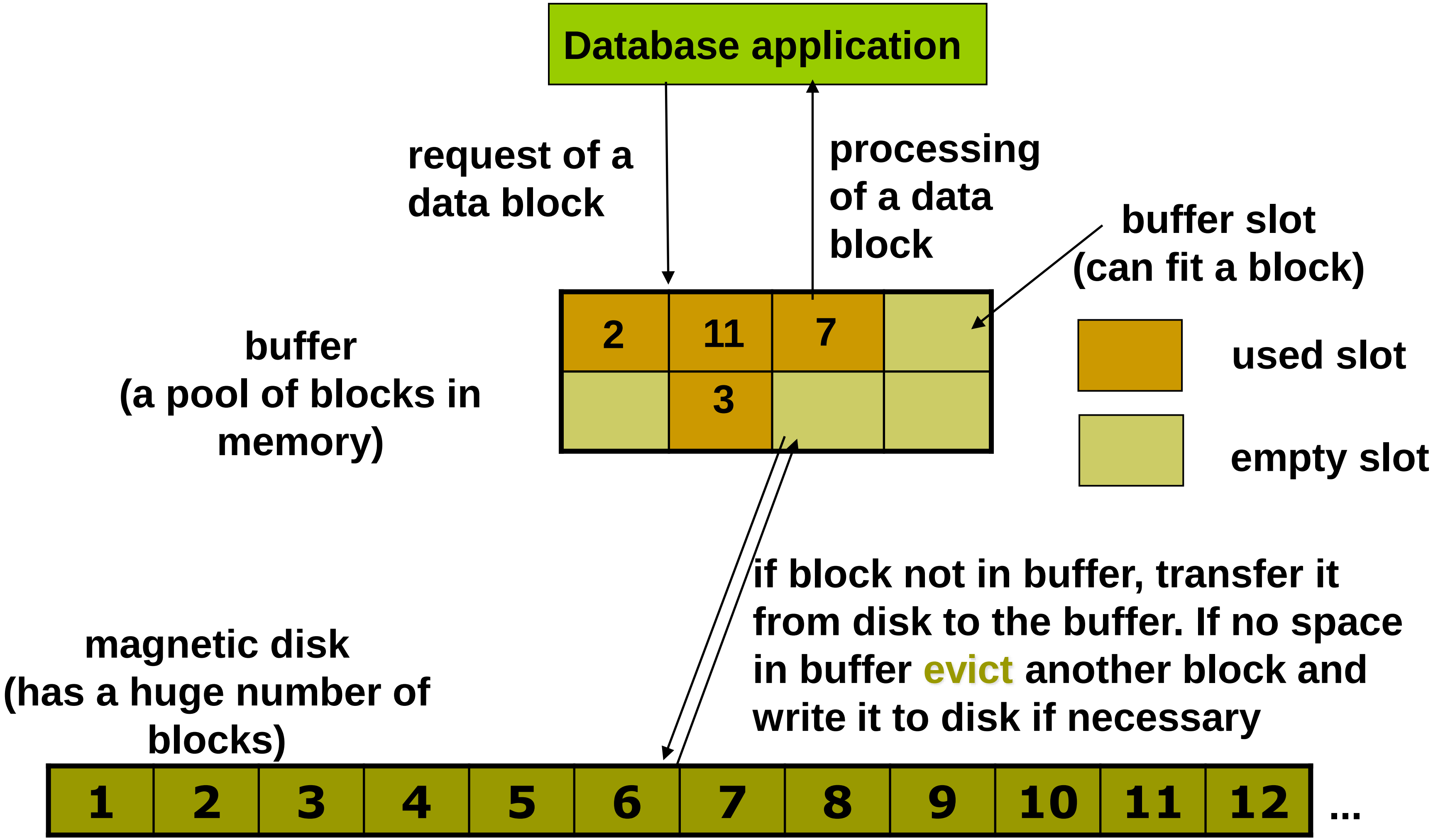
# Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
  - data is transferred between disk and main memory in blocks
  - sizes range from 512 bytes to several kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks:  more space wasted due to partially filled blocks
    - Typical block sizes today range from 4 to 16 kilobytes
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized

# Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer. Typical size of a block ranges between 4Kb-16Kb.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

# Storage Access (example)



Database application

request of a data block

processing of a data block

buffer slot (can fit a block)

buffer
(a pool of blocks in memory)

| | | | |
|---|---|---|---|
| 2 | 11 | 7 | |
| | 3 | | |

used slot

empty slot

if block not in buffer, transfer it from disk to the buffer. If no space in buffer **evict** another block and write it to disk if necessary

magnetic disk
(has a huge number of blocks)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Goal of buffer management: minimize block transfers

# Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  1. If the block is already in the buffer, the requesting program is given the address of the block in main memory
  2. If the block is not in the buffer,
     1. the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block.
     2. The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     3. Once space is allocated in the buffer, the buffer manager reads the block from the disk to the buffer, and passes the address of the block in main memory to requester.

# Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references. *If a block has not been recently used, then it is unlikely that it will be used in the near future.*
- This replacement policy is also used at different applications. A proxy server keeps in the most recently used web pages in a local cache. If a user requests again a page he has seen, it does not need to be downloaded again in the future.
- LRU works well for unpredicted access patterns.

# Buffer-Replacement Policies

- However, queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user’s query to predict future references
- LRU can be a bad strategy for certain access patterns involving repeated scans of data. Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

# File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- Each record has an address in the file, which is called record pointer or record id (simply rid).
- A simple approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

# Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **clustered file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
  - However, not good for queries accessing only a few relations
  - In general, this representation is barely used

# Part B
## Database Indexing

*Acknowledgement: This presentation is based on Prof. Nikos Mamoulis' COMP3323 slides.*

# Indexing

- Why indexing?
- What are the types of indexes that we have in a DBMS?
- How does the B+-tree index a relation based on a search key? How are B+-trees updated?
- How do hash-based indexes operate?

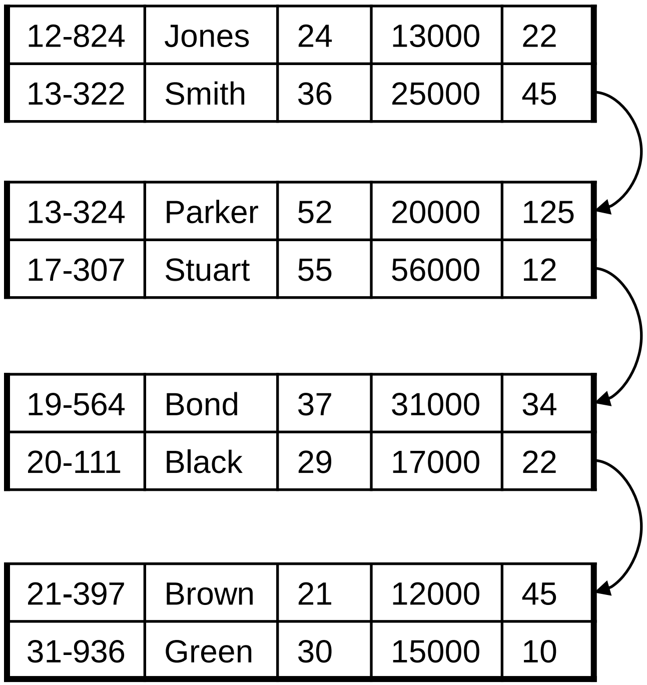
# Indexing: Motivation

- Assume that we have an Employees relation stored in a relation file

Employees

| *ssn* | *name* | *age* | *salary* | *lot* |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

Employees file



| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |

| | | | | |
|---|---|---|---|---|
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |

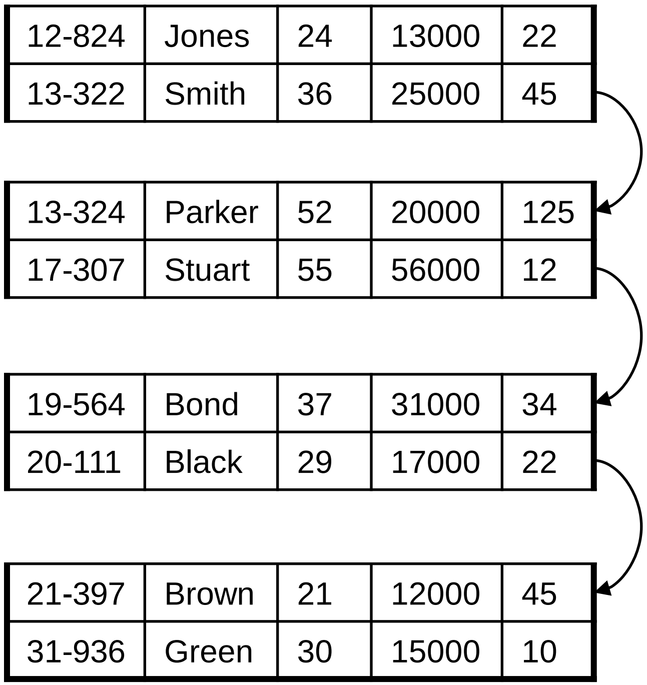
| | | | | |
|---|---|---|---|---|
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |

| | | | | |
|---|---|---|---|---|
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

# Indexing: Motivation (cont’d)

- The size of a record is quite large, so we can store only a few records in each block of the relation file

Employees file



| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |

| | | | | |
|---|---|---|---|---|
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |

| | | | | |
|---|---|---|---|---|
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |

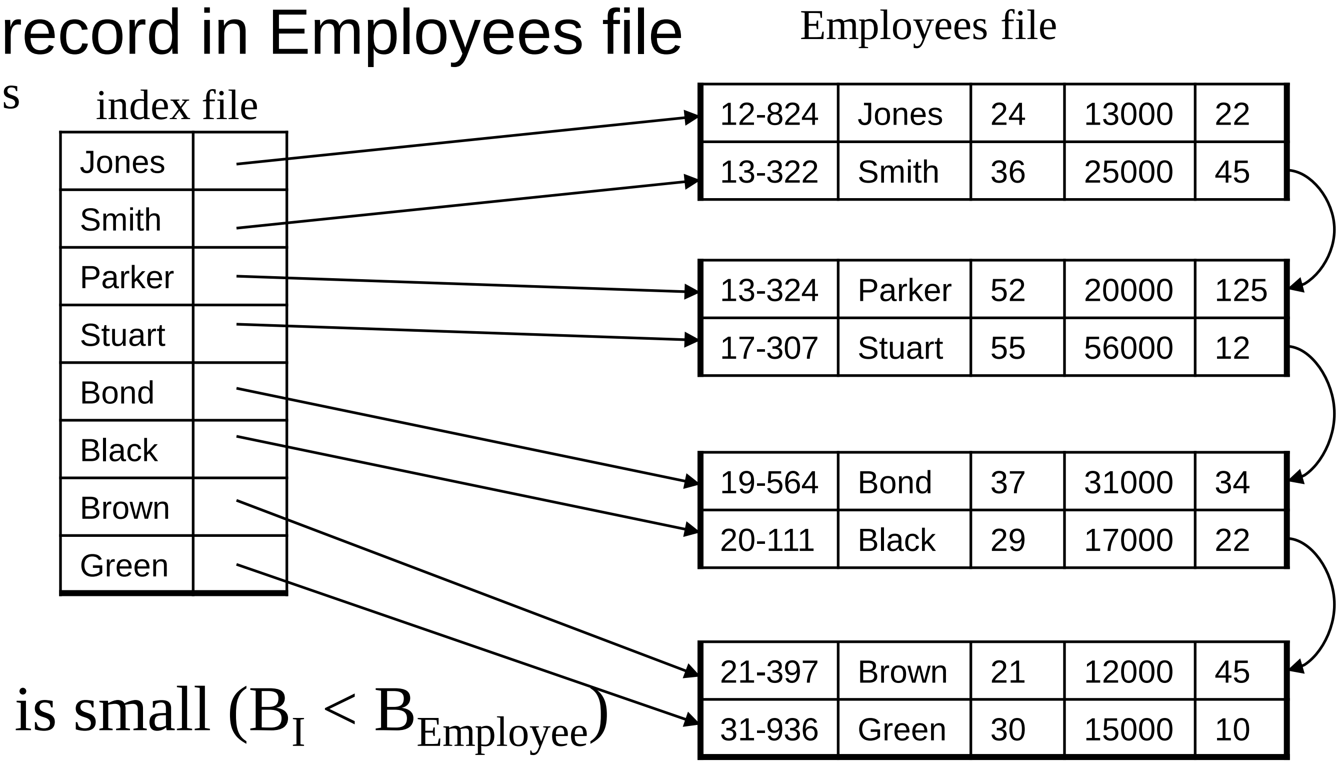
| | | | | |
|---|---|---|---|---|
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

- Consider the query:
  - Return the records of employees with name = “Green”
  - $\sigma_{name=\text{‘Green’}}$ Employees
- Query is evaluated by scanning the relation file and checking for each record if name=“Green”
- Cost = number of blocks in file ($B_{Employees}$)

# Indexing: Motivation (cont’d)

- Suppose that we keep a separate file with records of the form (name, record-id), such that there is one entry for each record in Employees file

$\sigma_{name=\text{‘Green’}}$ Employees



Employees file

index file

| |
|---|
| Jones |
| Smith |
| Parker |
| Stuart |
| Bond |
| Black |
| Brown |
| Green |

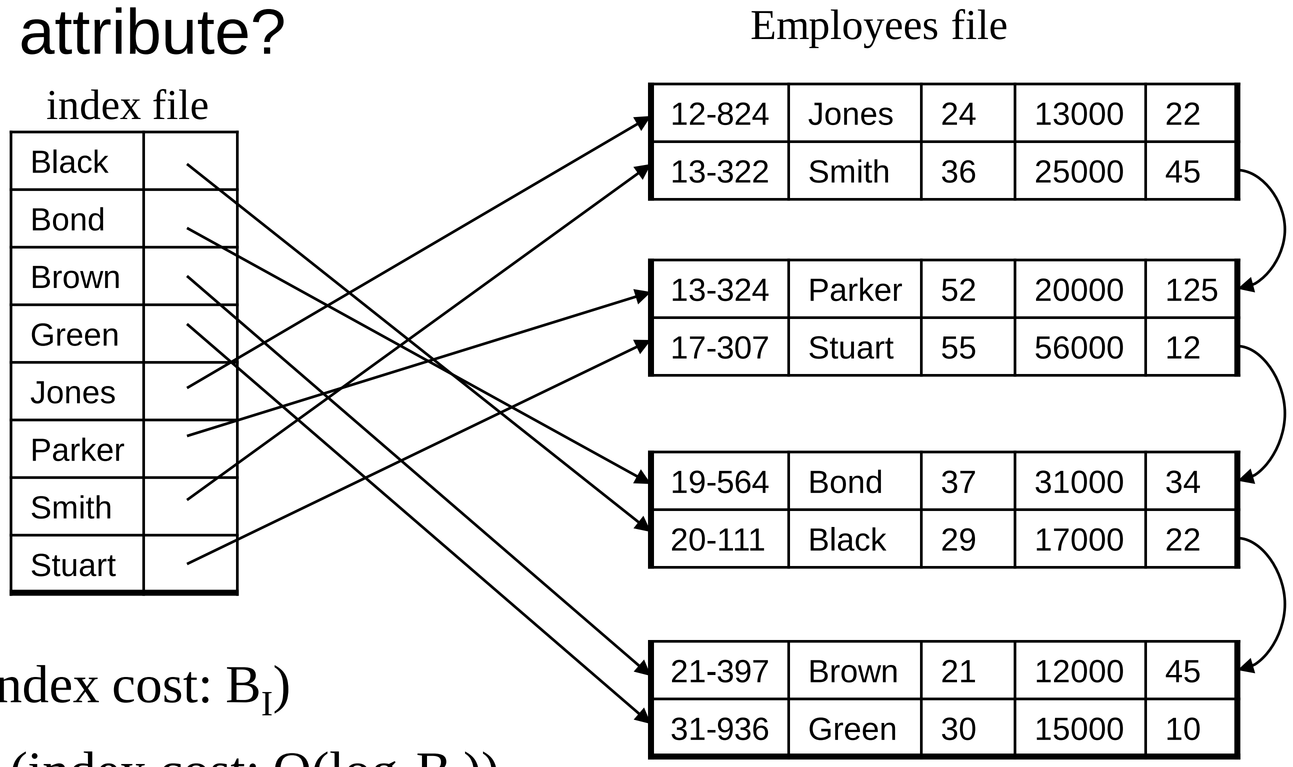
| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

- Index file is small ($B_I < B_{Employee}$)
- Cost = number of blocks in index ($B_I$) + number of blocks containing query results

# Indexing: Motivation (cont’d)

- Instead of having the index file entries in a random order, why don’t we keep them ordered by the value of the search attribute?

$\sigma_{name= \text{‘Green’}}$ Employees



index file

| | |
|---|---|
| Black | |
| Bond | |
| Brown | |
| Green | |
| Jones | |
| Parker | |
| Smith | |
| Stuart | |

Employees file

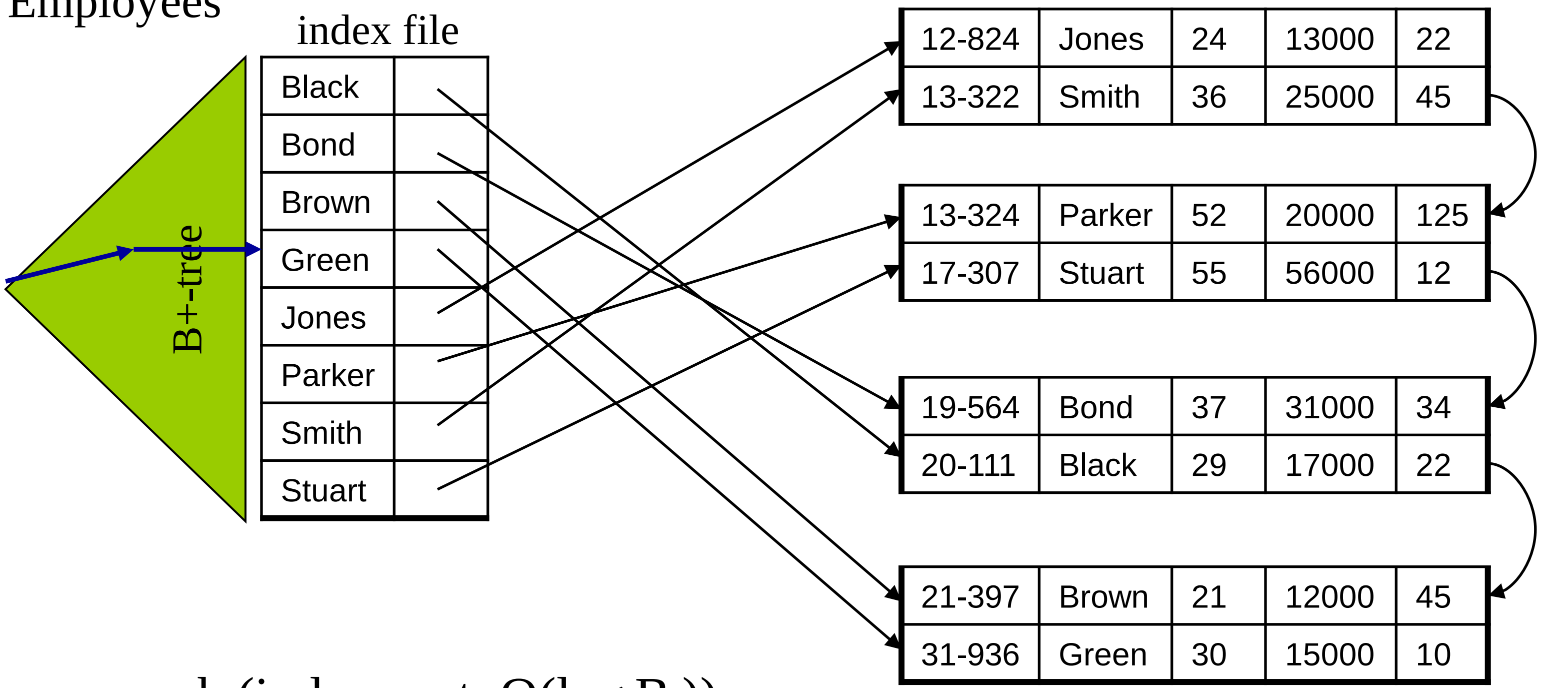
| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

- Linear scan (index cost: $B_I$)
- Binary search (index cost: $O(\log_2 B_I)$)
- Cost = $O(\log_2 B_I)$ + number of blocks containing query results

# Indexing: Motivation (cont’d)

- Why don’t we use a data structure that can maintain the index entries sorted after insertions/deletions and also assist faster search?

$\sigma_{name=\text{'Green'}}$ Employees



Employees file

| | |
|---|---|
| Black | |
| Bond | |
| Brown | |
| Green | |
| Jones | |
| Parker | |
| Smith | |
| Stuart | |

| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |

| | | | | |
|---|---|---|---|---|
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |

| | | | | |
|---|---|---|---|---|
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |

| | | | | |
|---|---|---|---|---|
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

- *f*-way search (index cost: $O(\log_f B_I)$)
- Cost = $O(\log_f B_I)$ + number of blocks containing query results

# Indexing: Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
    - E.g., author catalog in library, phone directory index
- **Search Key** – an attribute or a set of attributes used to look up records in a file. (Phone book directory.)
- An **index file** consists of records (called **index entries**) of the form

| | |
|---|---|
| search-key | pointer |

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
    - **Ordered indices:** search keys are stored in sorted order
        - Example: The $B^+$-tree
    - **Hash indices:** search keys are distributed across “buckets” using a “hash function”.

# What is a Good Index?

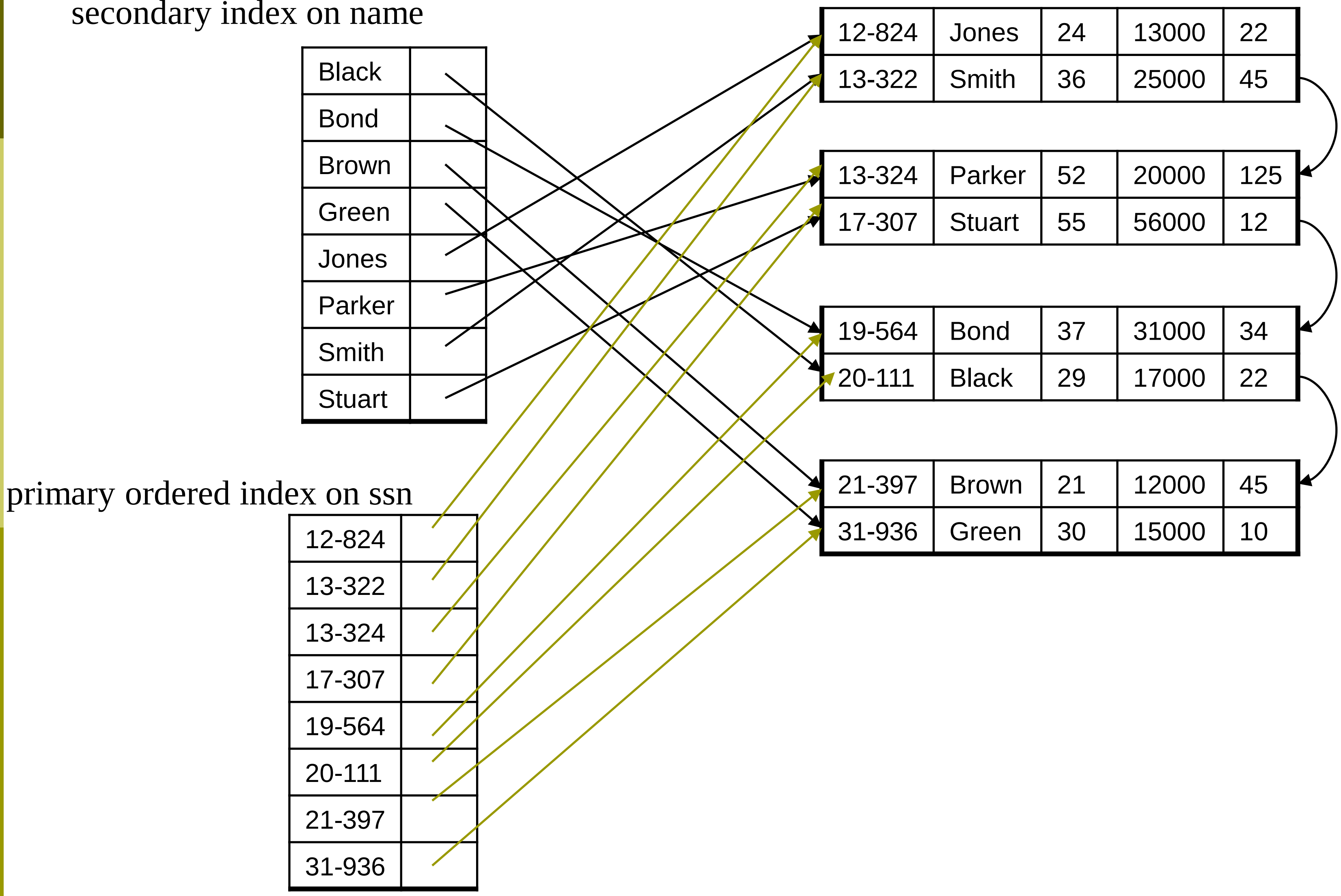
- Index quality is evaluated by several factors
  - Access types supported by the index efficiently. E.g.,
    - records with a specified value in the attribute (equality query)
      - Find employee with name="Green"
    - or records with an attribute value falling in a specified range of values (range query).
      - Find employees with salary between 20000 and 30000.
  - Access time – query response time
  - Insertion time – data record insertion time
  - Deletion time – data record deletion time
  - Space overhead – size of the index file

# Classification of Indexes

- In an **ordered index**, index entries are stored sorted on the search key value.  E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustered index**
  - The search key of a primary indexed file is usually but not necessarily the primary key.
- Index-sequential file: ordered sequential file with a primary index.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustered index.

# Classification of Indexes: examples

Employees file



secondary index on name

| | |
|---|---|
| Black | |
| Bond | |
| Brown | |
| Green | |
| Jones | |
| Parker | |
| Smith | |
| Stuart | |

primary ordered index on ssn

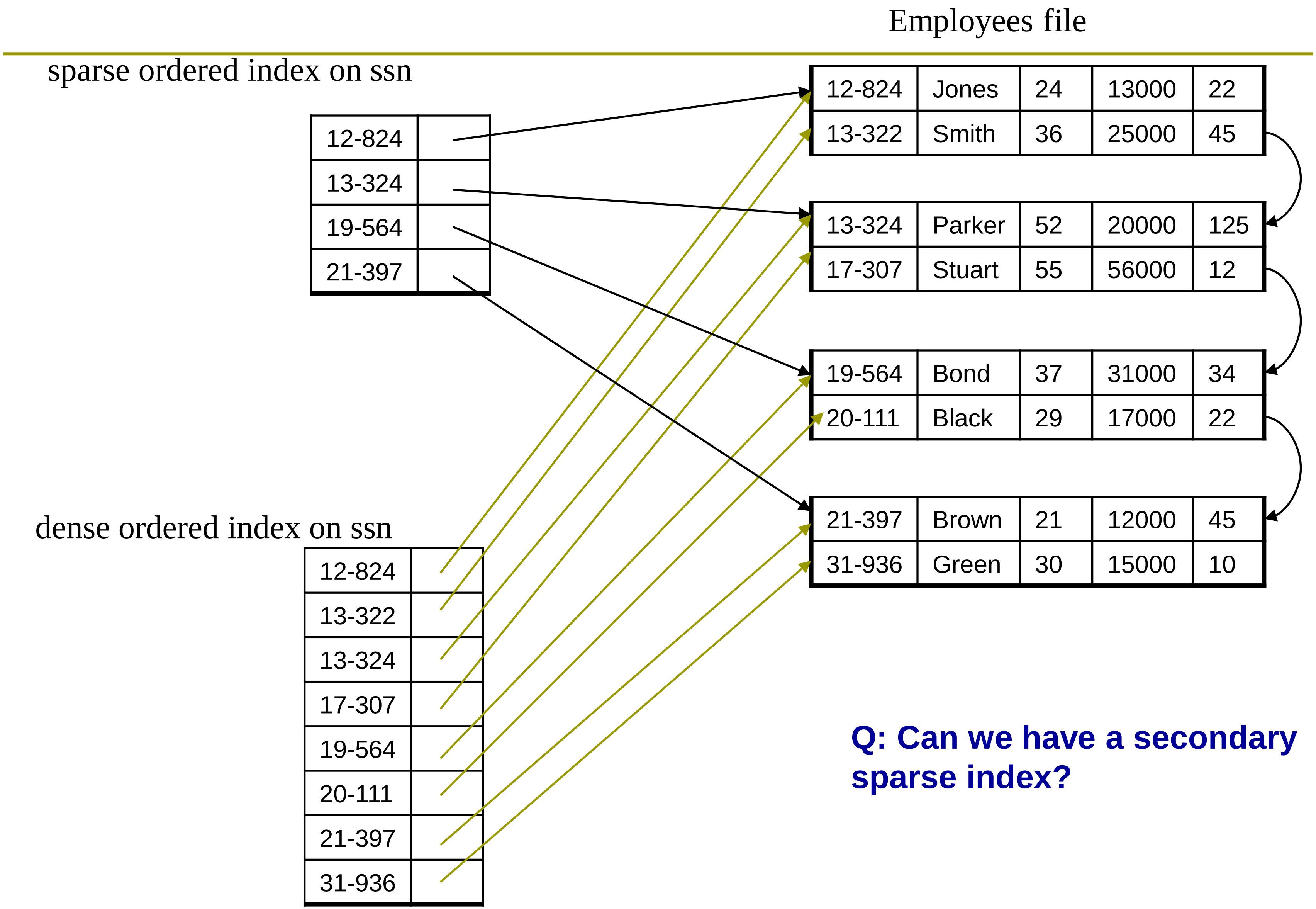
| | |
|---|---|
| 12-824 | |
| 13-322 | |
| 13-324 | |
| 17-307 | |
| 19-564 | |
| 20-111 | |
| 21-397 | |
| 31-936 | |

| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

# Classification of Indexes (cont’d)

- Dense index — Index record appears for every search-key value in the file.
- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

# Classification of Indexes: examples



Employees file

| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |

| | | | | |
|---|---|---|---|---|
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |

| | | | | |
|---|---|---|---|---|
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |

| | | | | |
|---|---|---|---|---|
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

sparse ordered index on ssn

| | |
|---|---|
| 12-824 | |
| 13-324 | |
| 19-564 | |
| 21-397 | |

dense ordered index on ssn

| | |
|---|---|
| 12-824 | |
| 13-322 | |
| 13-324 | |
| 17-307 | |
| 19-564 | |
| 20-111 | |
| 21-397 | |
| 31-936 | |

**Q: Can we have a secondary sparse index?**

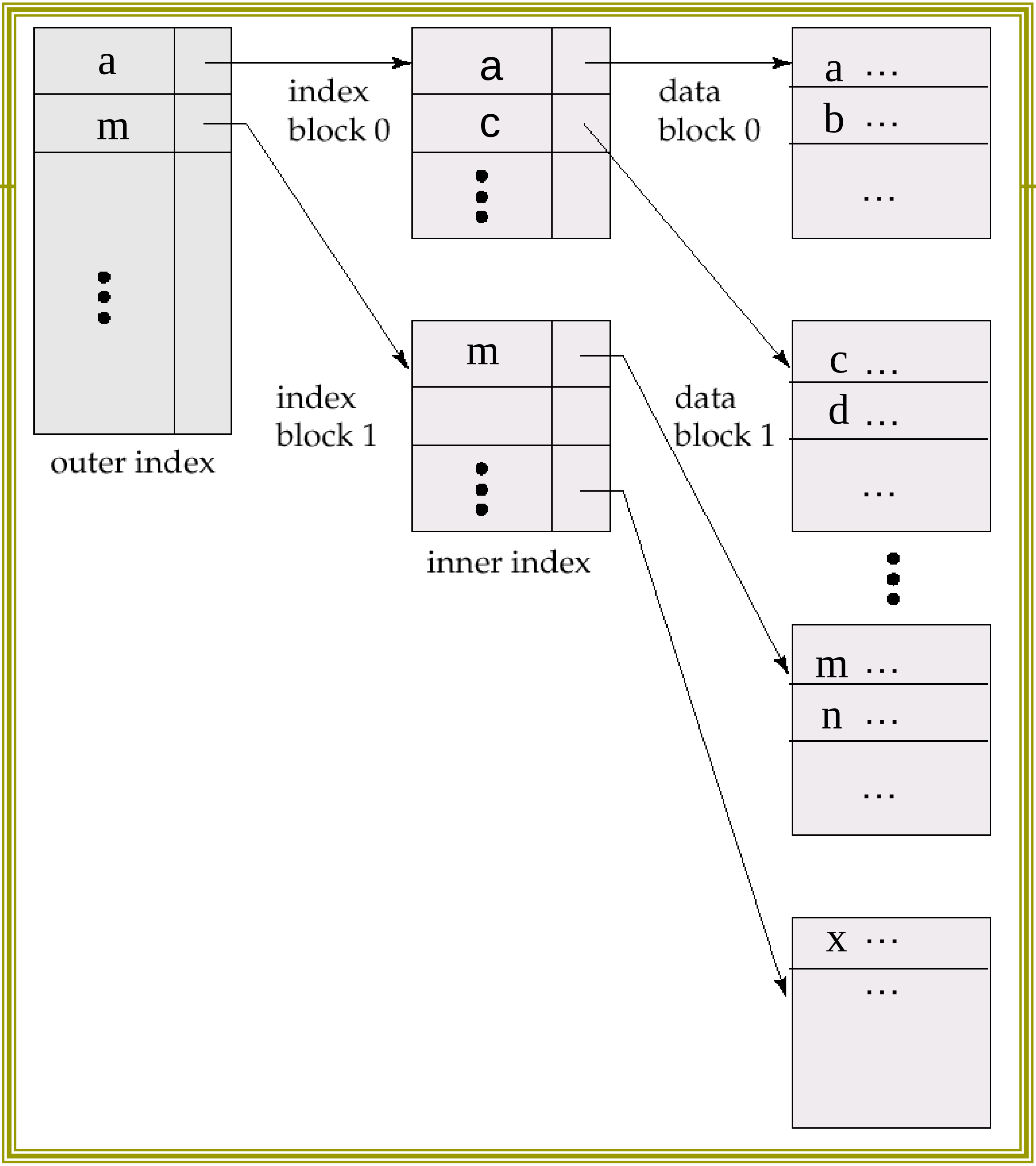
# Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
  - Index is much smaller than relation file (cheap scan)
  - Index can be ordered (fast search)
- When a file is modified, every index on the file must be updated
  - Updating indices imposes overhead on database modification
  - Indexes should be used with care
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

# Multilevel Index

- If index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat 1st level of index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index on 1st-level index file
  - inner index – the 1st-level index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

# Multilevel Index (Cont.)



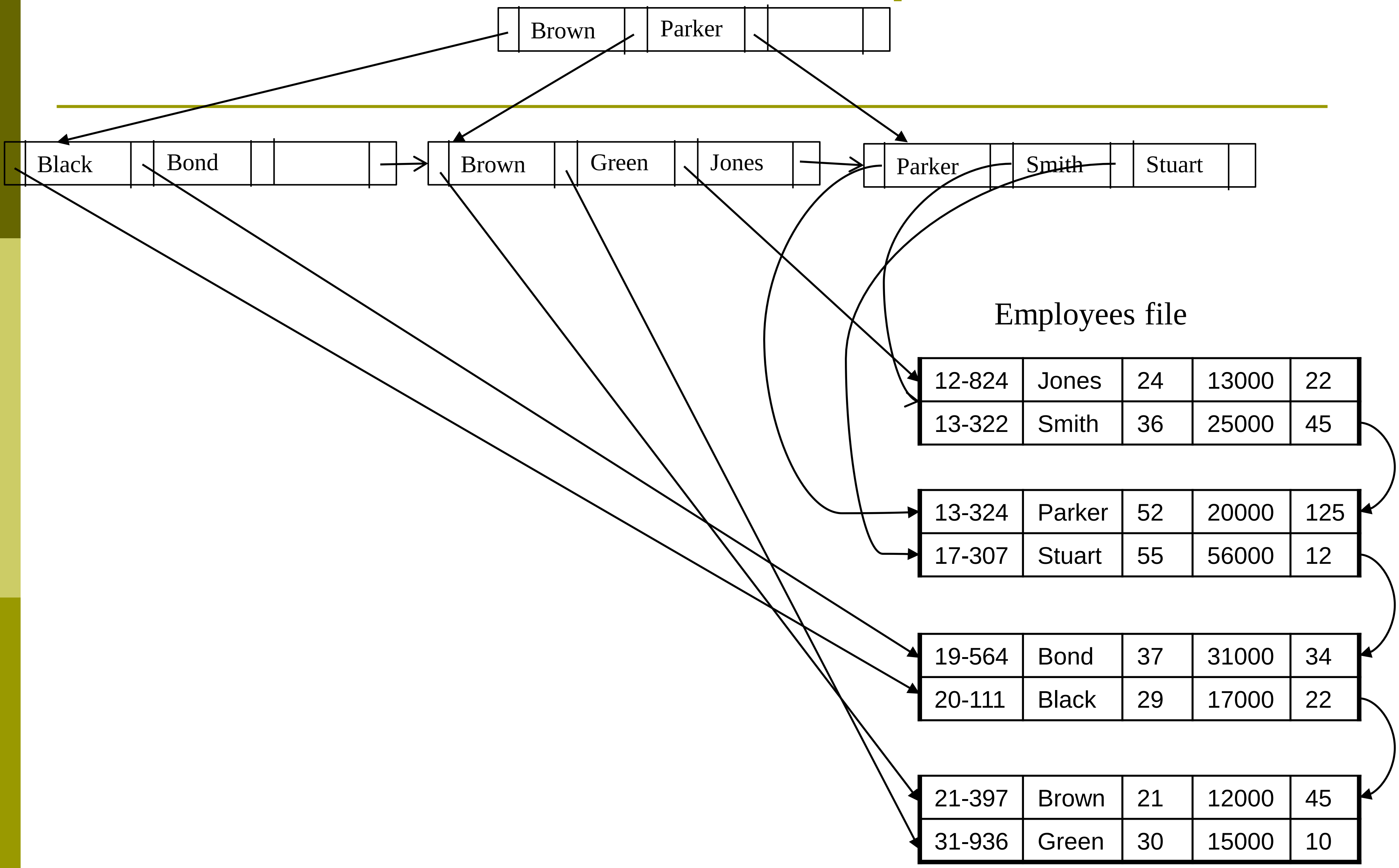
# B$^+$-Tree Index Files

- A dynamic, multi-level index
- Advantage: automatically reorganizes itself with small local changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B$^+$-trees: extra insertion and deletion overhead, space overhead.
- Advantages of B$^+$-trees outweigh disadvantages, and they are used extensively.

# $B^+$-Tree Index: Basic Properties

- Disk-based tree structure
  - every node of the tree is a block and has an address (block-id) on the disk
- Multi-way tree
  - each node has multiple children (between $\lceil n/2 \rceil$ and n, where $\lceil n/2 \rceil$ is the **order** or **degree** of the tree)
  - Therefore, at least 50% of the space in a node is guaranteed to be occupied (this rule may not apply to tree root)
- Balanced tree
  - all paths from the root to a leaf have the same length
  - guarantees good search performance (to be seen later)
- Disjoint partition of attribute domain into ranges
  - each sub-tree indexes a range in the attribute domain
  - the entries of a directory node define the separators between domain intervals
  - leaf nodes store index entries and pointers to the relation file

# B$^+$-Tree Index: Example



Brown | Parker

Black | Bond

Brown | Green | Jones

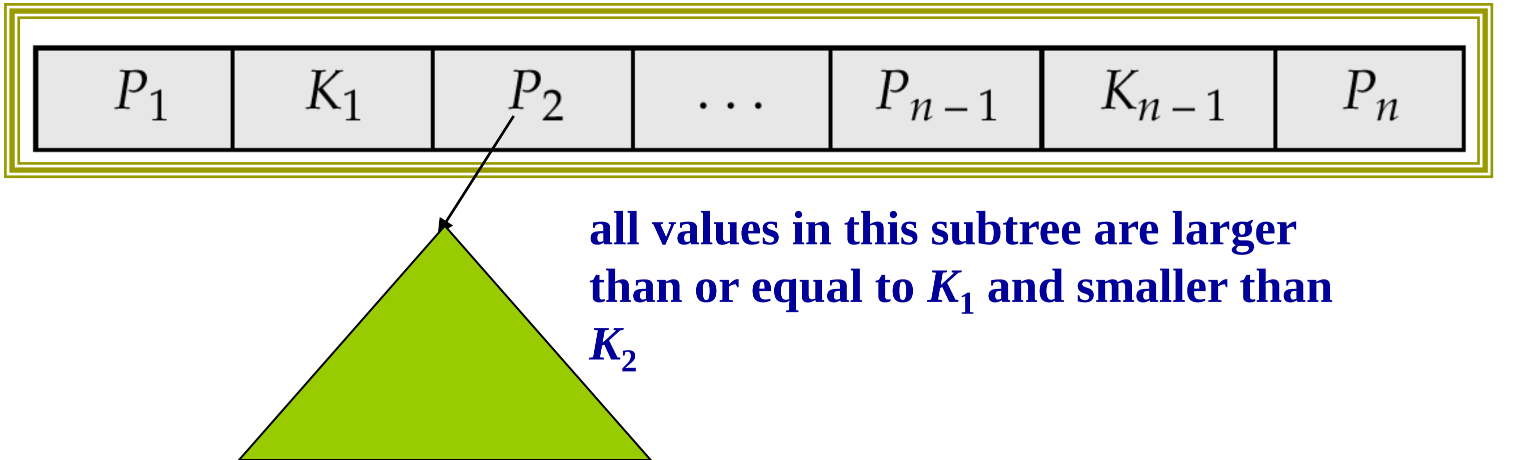
Parker | Smith | Stuart

Employees file

| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

# Non-Leaf Nodes in B$^+$-Trees

- Each non-leaf node contains up to n-1 search key values and up to n pointers
- All non-leaf nodes (except root) contain at least $\lceil n/2 \rceil$ pointers ($\lceil n/2 \rceil$ is sometimes called the minimum **fan-out** or **degree**)
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which $P_1$ points are less than $K_1$
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which $P_i$ points have values greater than or equal to $K_{i-1}$ and smaller than $K_{m-1}$



| | | | | | | |
|---|---|---|---|---|---|---|
| $P_1$ | $K_1$ | $P_2$ | $\ldots$ | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

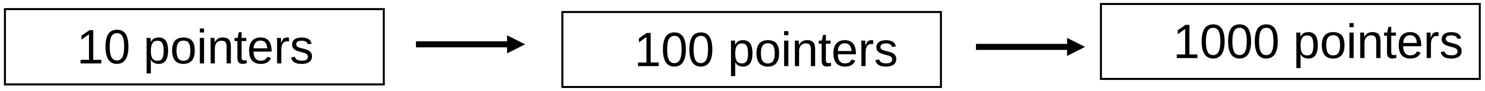
**all values in this subtree are larger than or equal to $K_1$ and smaller than $K_2$**

# Leaf Node in a B$^+$-Tree

- Contains between $\lceil (n-1)/2 \rceil$ and n-1 entries
- Each index entry is a search key value + a record-id
- If $L_i$, $L_j$ are leaf nodes and $i < j$, $L_i$' s search-key values are all smaller than $L_j$' s search-key values
- Each leaf node is linked with a pointer to the next node

# Observations about B$^+$-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
  - Nodes of the tree are dynamically created/deleted, so we cannot guarantee physical closeness
- The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices.
- The B$^+$-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.



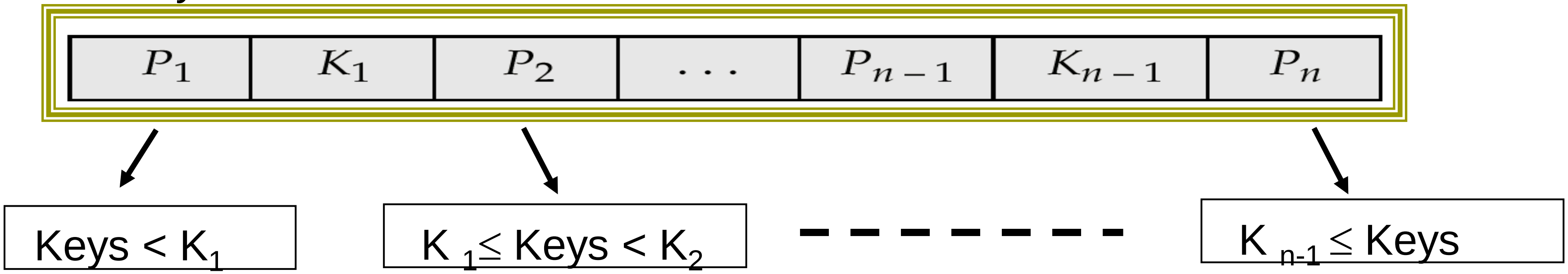
10 pointers → 100 pointers → 1000 pointers

$\mathrm{Log}_{10}$ 1000 = 3, 3 levels can handle a file of 1000 records for n/2 = 10

- Insertions and deletions to the main file can be handled efficiently (in logarithmic time).

# Queries on B$^+$-Trees

- Find all records with a search-key value of *k.*
   1. Start with the root node
      1. Examine the node for the smallest search-key value > *k.*
      2. If such a value exists, assume it is $K_i$. Then follow $P_i$ to the child node. (E.g. $P_2$ is for keys in $K_1 \leq$ Keys $< K_2$ ).
      3. Otherwise $k \geq K_{n-1}$, where there are *n* pointers in the node. Then follow $P_n$ to the child node.
   2. If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
   3. Eventually reach a leaf node. If for some *i*, key $K_i$ = k follow pointer $P_i$ to the desired record. Else no record with search-key value *k* exists.



| $P_1$ | $K_1$ | $P_2$ | . . . | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|---|---|---|---|---|---|---|

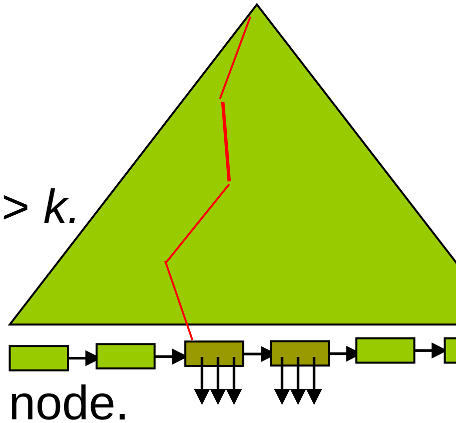
Keys $< K_1$

$K_1 \leq$ Keys $< K_2$

$K_{n-1} \leq$ Keys

# Queries on $B^+$-Trees – Range Queries

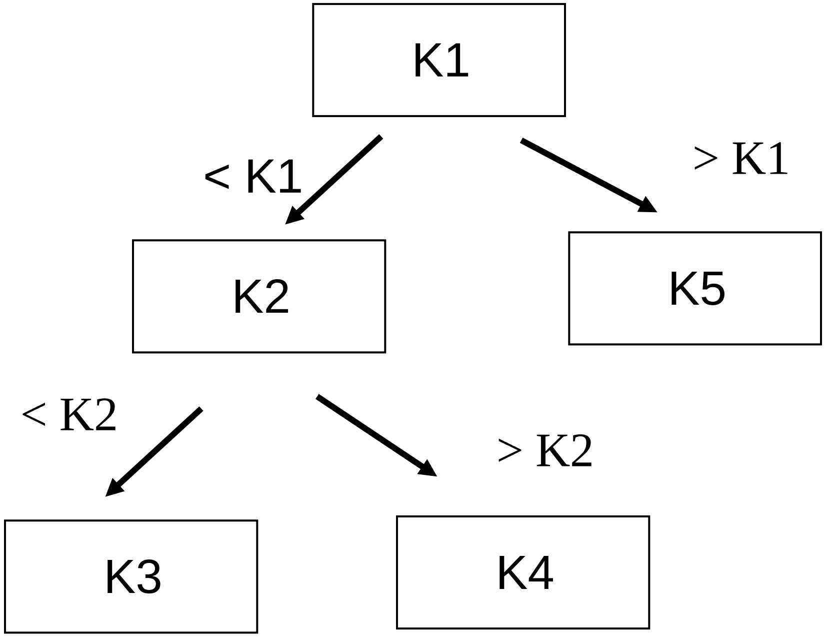
- Find all records with a search-key value between *k* and *m* (*k*<*m*).
  1. Start with the root node
     1. Examine the node for the smallest search-key value > *k.*
     2. If such a value exists, assume it is $K_j$. Then follow $P_i$ to the child node
     3. Otherwise $k \geq K_{n-1}$, where there are *n* pointers in the node. Then follow $P_n$ to the child node.
  2. If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  3. Eventually reach a leaf node. If for some *i*, $k \leq K_i \leq m$ follow pointer $P_i$ to the desired record. Continue with next entry $K_{i+1}$, while $K_{i+1} \leq m$. If at end of leaf node follow pointer to next node, until $K_i > m$ or end of index.



# Queries on $B^{+}$-Trees (Cont.)

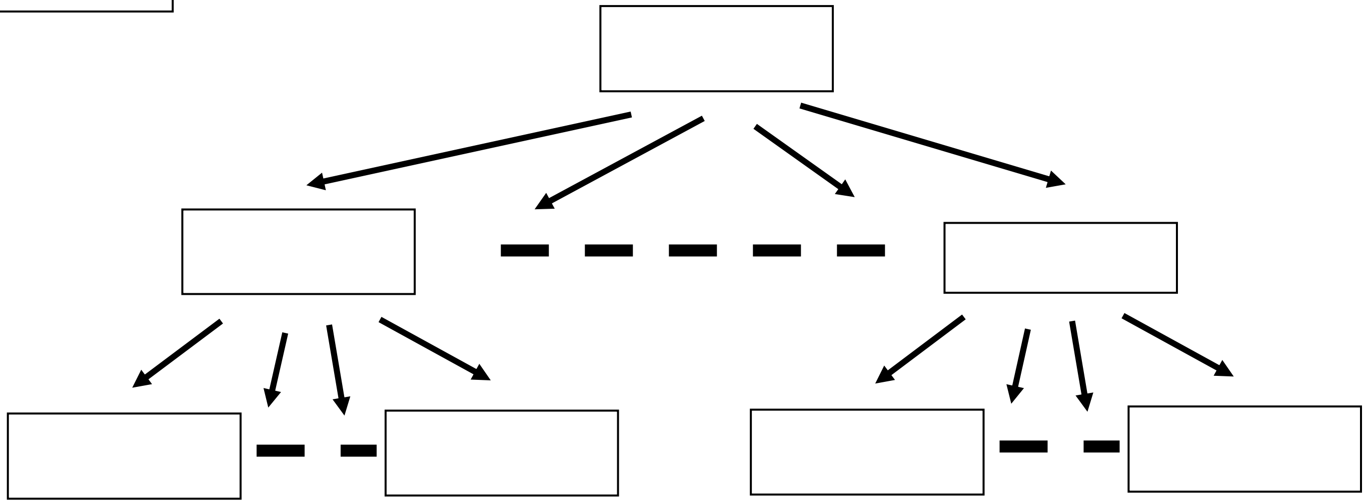
- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are *K* search-key values in the file, the path is not longer than $\lceil \log_{(n/2)}(K) \rceil$. (The degree of a node is no less than n/2).
- A node has generally the same size of a disk block, typically 4 kilobytes, and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n/2 = 50, at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - ( $log_2(1{,}000{,}000) \approx 20$)
  - above difference is significant since every node access may need a disk I/O, costing around 10 milliseconds!
- Similar result for a binary search of an ordered sequential file

# Comparison with a Binary Tree



Problems of binary tree :

1. **Small fan-out:** a node only has two children. Tree will be very tall. Large number of node accesses. Many I/Os introduced.
2. **Bad space utilization:** Each node is a disk page of only two entries. Index file could be larger than relation file! This is why we require that nodes are at least 50% full.

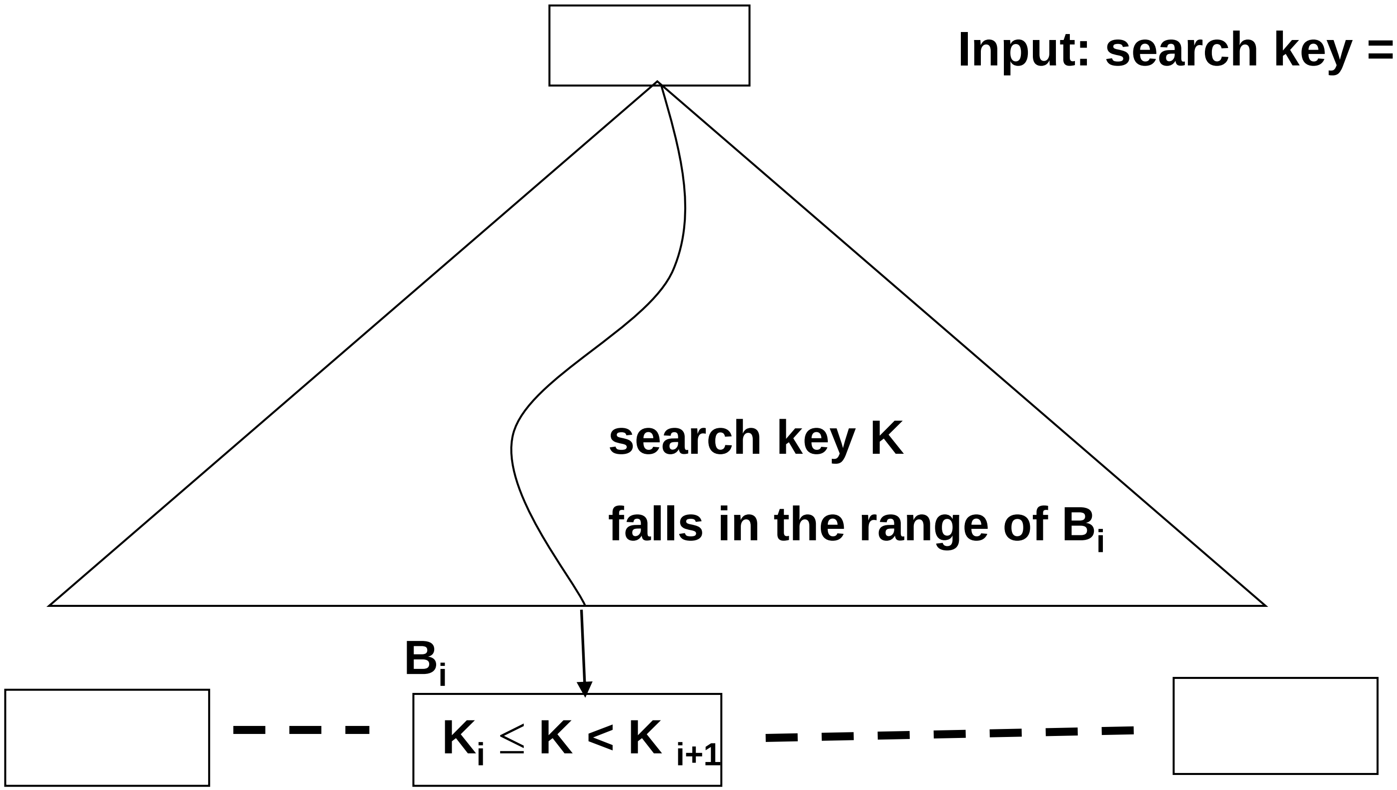


# Updates on $B^+$-Trees: Insertion

- Find the leaf node in which the search-key value to be inserted would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary one more pointer is associated with the search key value.
- If the search-key value is not there, then add the record to the main file. Then:
  - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  - Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slides.

# Updates on $B^+$-Trees: Insertion (Cont.)

**Input: search key = K**



**search key K**

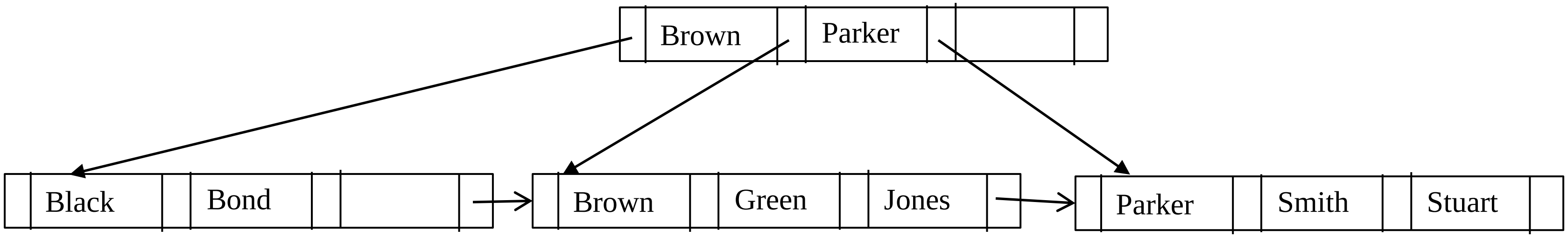
**falls in the range of $B_i$**

**$B_i$**

**$K_i \leq K < K_{i+1}$**

1. **Key K is found in the leave node $B_i$, so data can be added to the data file.**
2. **Key K is not found in $B_i$. K is inserted into $B_i$ , and then to the data file.**
3. **If $B_i$ is already full, a split of $B_i$ is performed.**

# $B^+$-Tree Index: Splitting Example



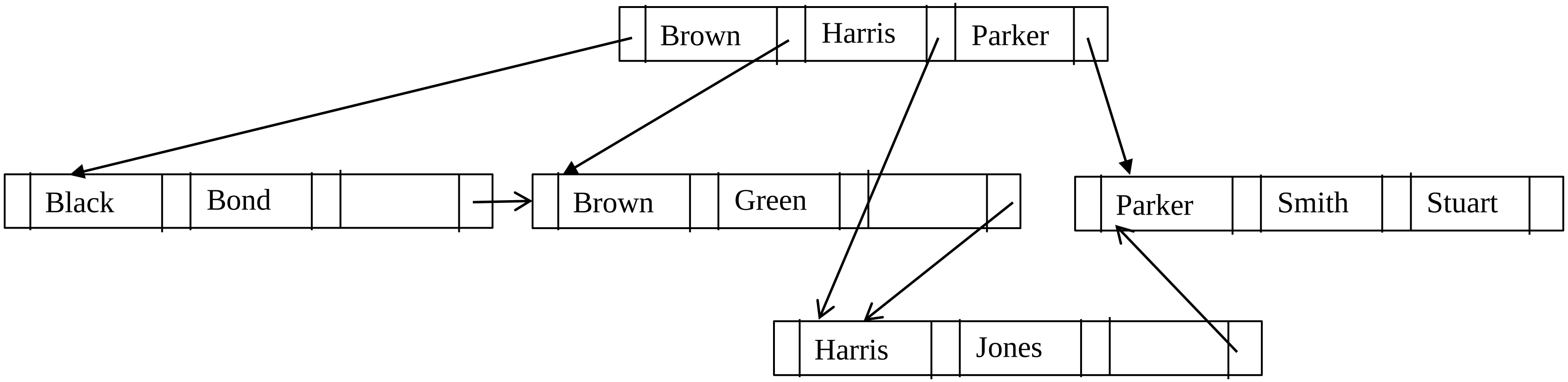
Brown Parker

Black Bond

Brown Green Jones

Parker Smith Stuart

After inserting entry “Harris”:



Brown Harris Parker

Black Bond

Brown Green

Parker Smith Stuart

Harris Jones

# Updates on B$^+$-Trees: Insertion (Cont.)

- Splitting a node:
  - take the (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split. If the parent is full, split it and propagate the split further up.
- The splitting of nodes proceeds upwards till a node that is not full is found. In the worst case the root node may be split increasing the height of the tree by 1.
- Non-leaf node splitting:
  - Overflown node has n+1 pointers and n values
  - Leave first $\lceil n/2 \rceil$ key values and $\lceil n/2 \rceil$+1 pointers to original node
  - Move last $\lceil n/2 \rceil$ key values and $\lceil n/2 \rceil$+1 pointers to new node
  - insert (middle key value, pointer to new node) to parent node

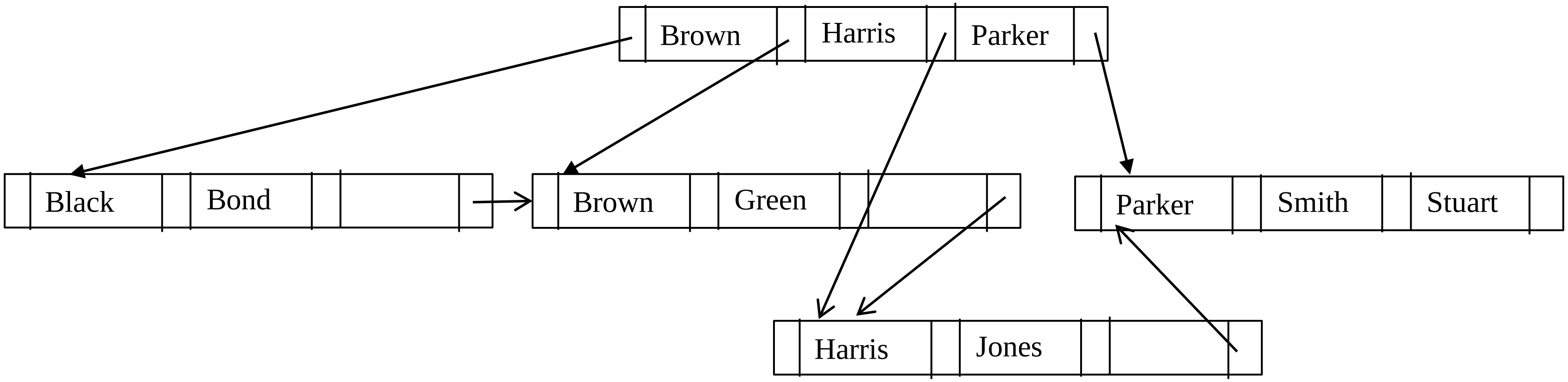
# Updates on $B^+$-Trees: Deletion

- Find the record to be deleted, and remove it from the relation file
- Remove (search-key value, record-id) of deleted record from the leaf node of the B+-tree
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node. (Deletion triggers a merge.)
  - Delete the pair ($K_{i-1}$, $P_i$), where $P_i$ is the pointer to the deleted node, from its parent, recursively using the above procedure.

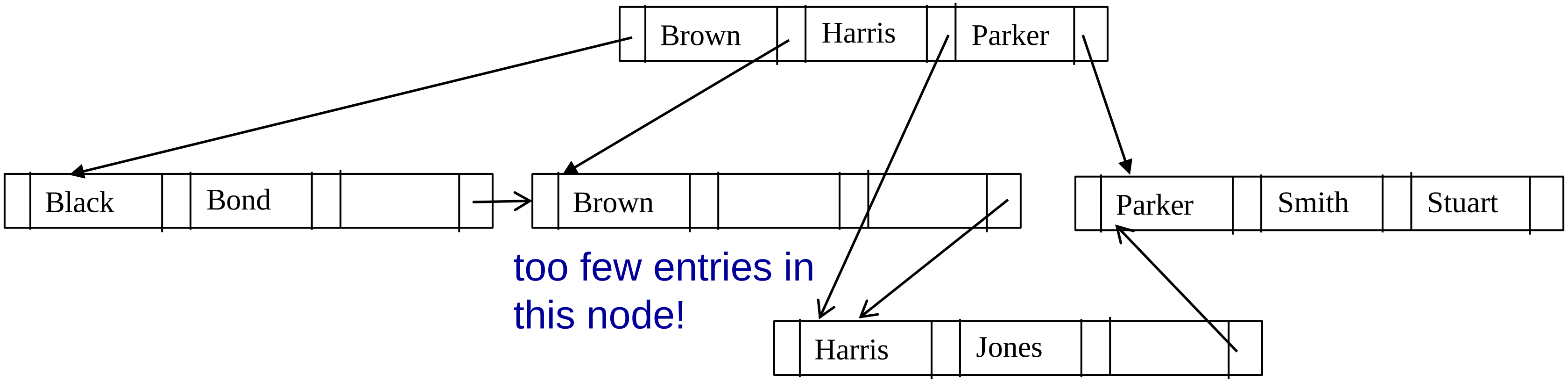
# Updates on B$^+$-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling does not fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries. (Deletion and rebalancing.)
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards until a node which has n/2 or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

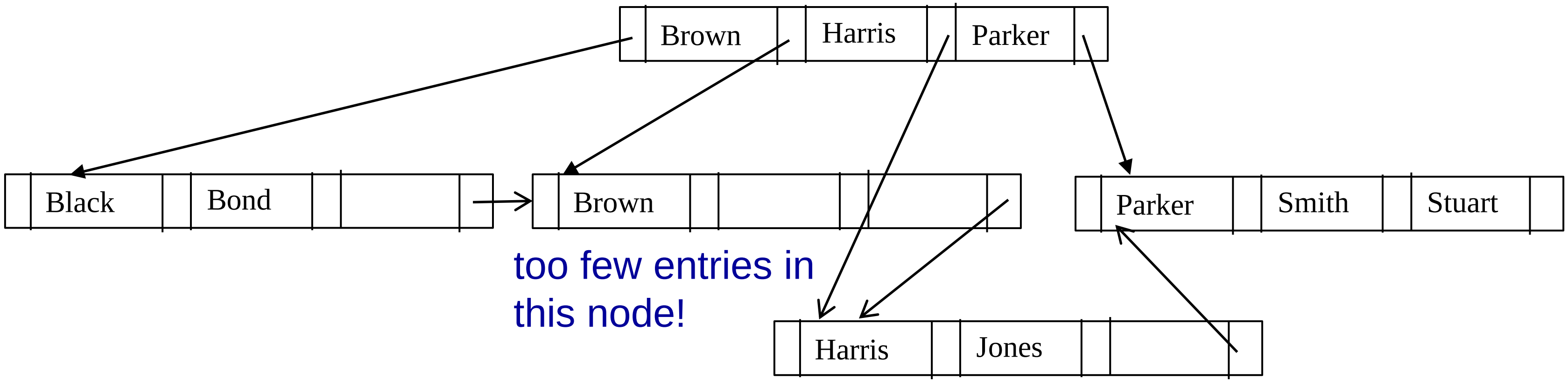
# $B^+$-Tree Index: Deletion Example 1



After deleting entry “Green”:



# B$^+$-Tree Index: Deletion Example 2



Brown Harris Parker

Black Bond

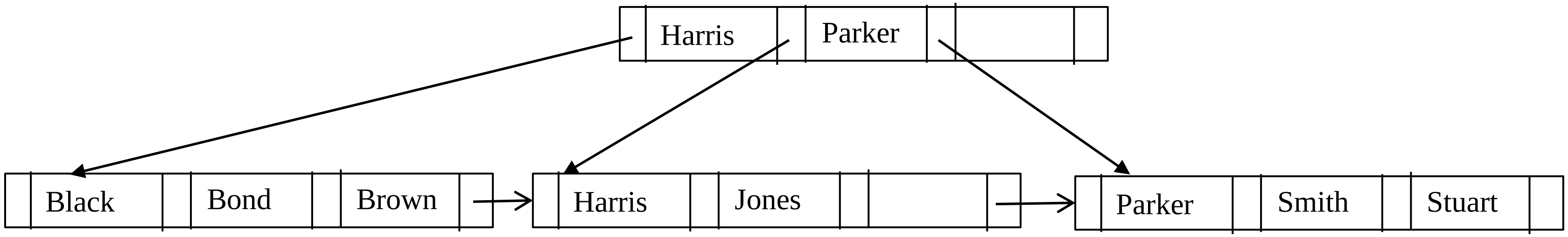
Brown

Parker Smith Stuart

Harris Jones

too few entries in this node!

Move “Brown” to prev. node and delete its node



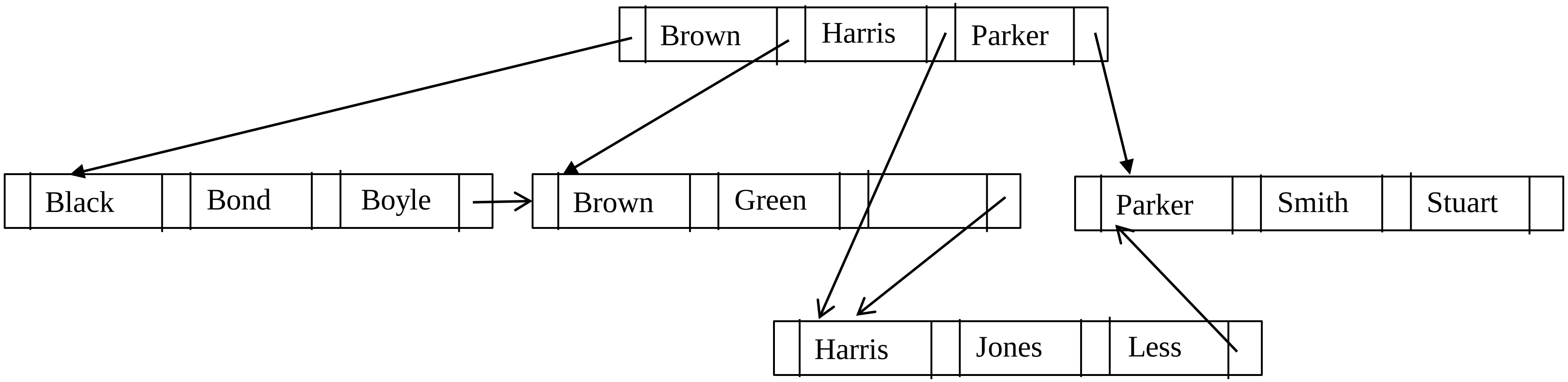
Harris Parker

Black Bond Brown

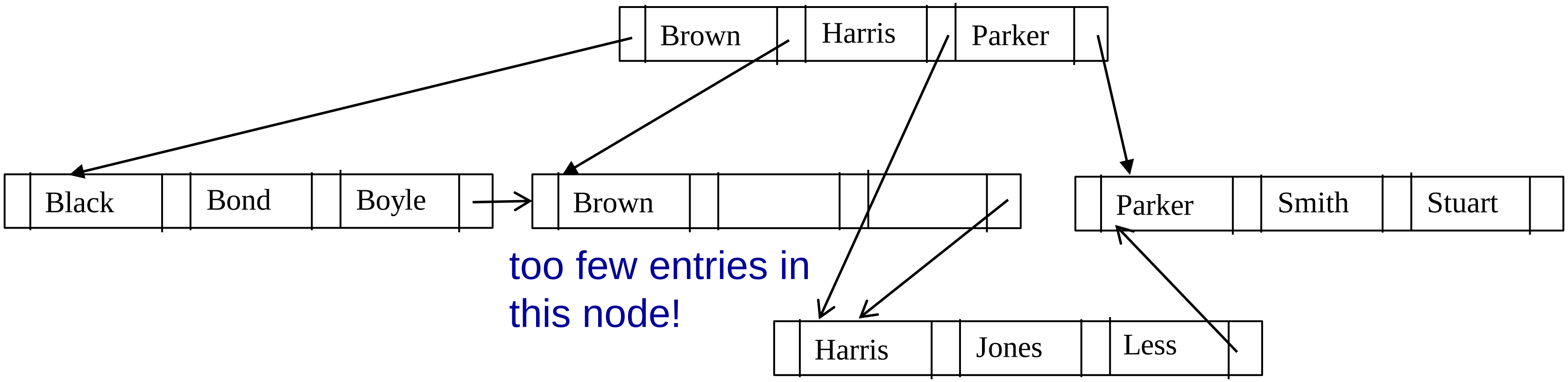
Harris Jones

Parker Smith Stuart

# B$^+$-Tree Index: Deletion Example 3

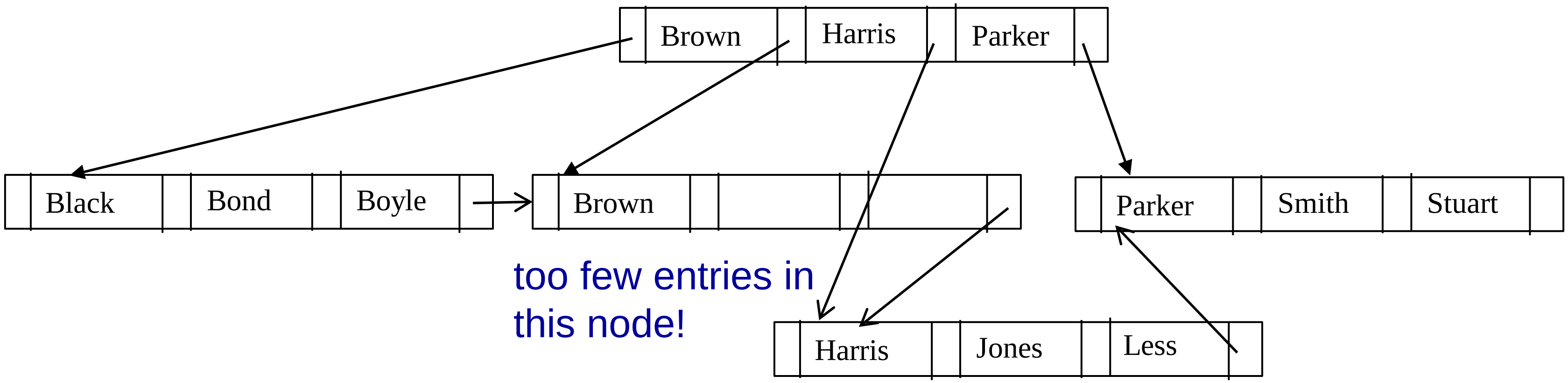


After deleting entry “Green”:



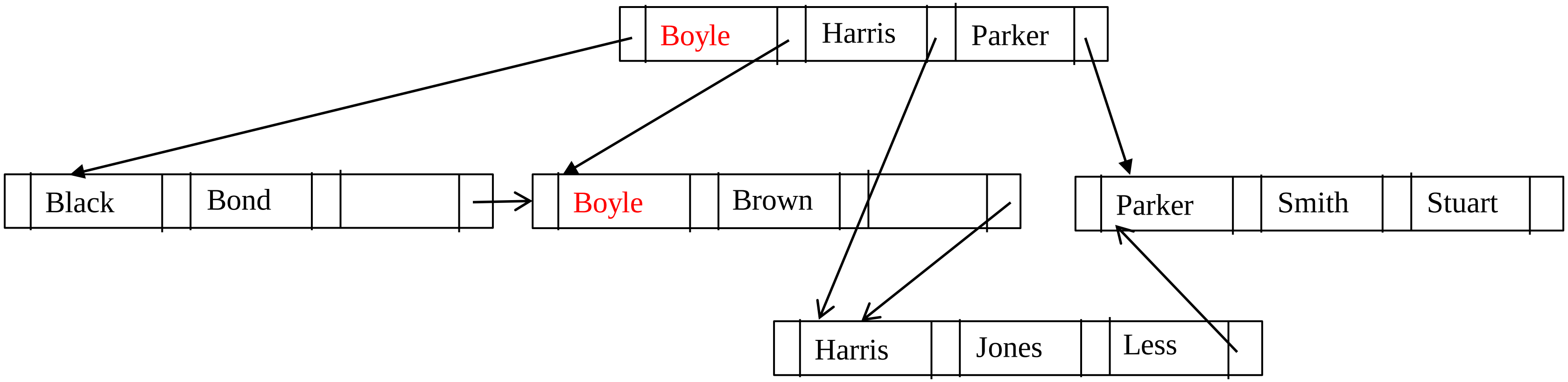
too few entries in this node!

# B+-Tree Index: Deletion Example 4



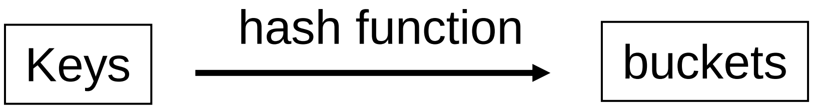
too few entries in this node!

Redistribute entries from sibling nodes:
1) Boyle is moved to underflown node
2) Parent entry changes



# Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*


Keys

hash function

buckets

h (Smith) = i ;  Smith will be in bucket i

- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record. (Collision).

# Example of Hash File Organization

Hash file organization of *Employees* file, using *name* as key
(See figure in next slide.)

- There are 3 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 3
  - E.g. h(Smith) = 0 h(Jones) = 0 h(Bond) = 2

$\mathrm{mod}_3(19+13+9+20+8) =$

$\mathrm{mod}_3\ 69 = 0$

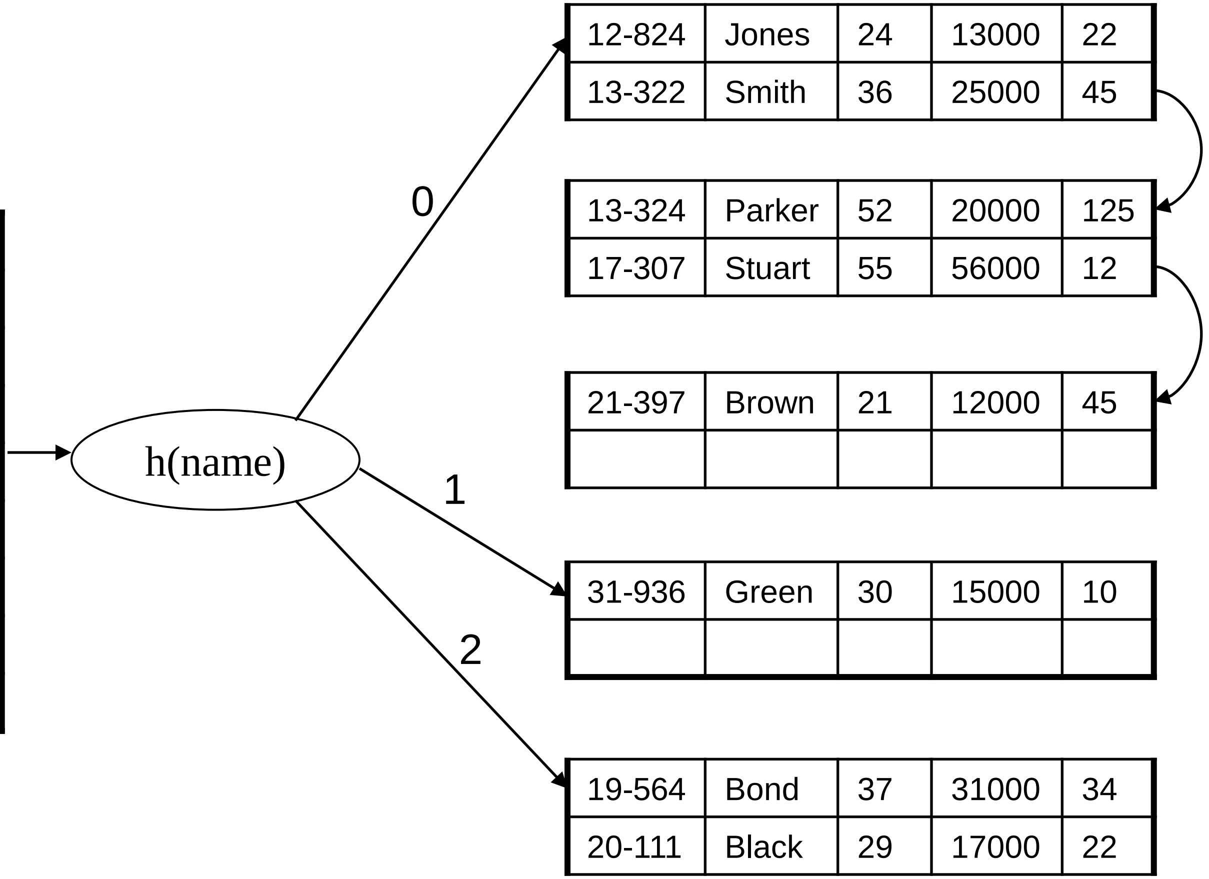
It is always possible to locate uniquely a bucket for a key with a given hash function.

# Example of Hash File Organization

Hash file organization of *Employees* file, using *name* as key

Employees

| ***ssn*** | ***name*** | ***age*** | ***salary*** | ***lot*** |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |



| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |

| | | | | |
|---|---|---|---|---|
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |

| | | | | |
|---|---|---|---|---|
| 21-397 | Brown | 21 | 12000 | 45 |
| | | | | |

| | | | | |
|---|---|---|---|---|
| 31-936 | Green | 30 | 15000 | 10 |
| | | | | |

| | | | | |
|---|---|---|---|---|
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |

**Q: What makes a good hash function ?**

# Hash Functions

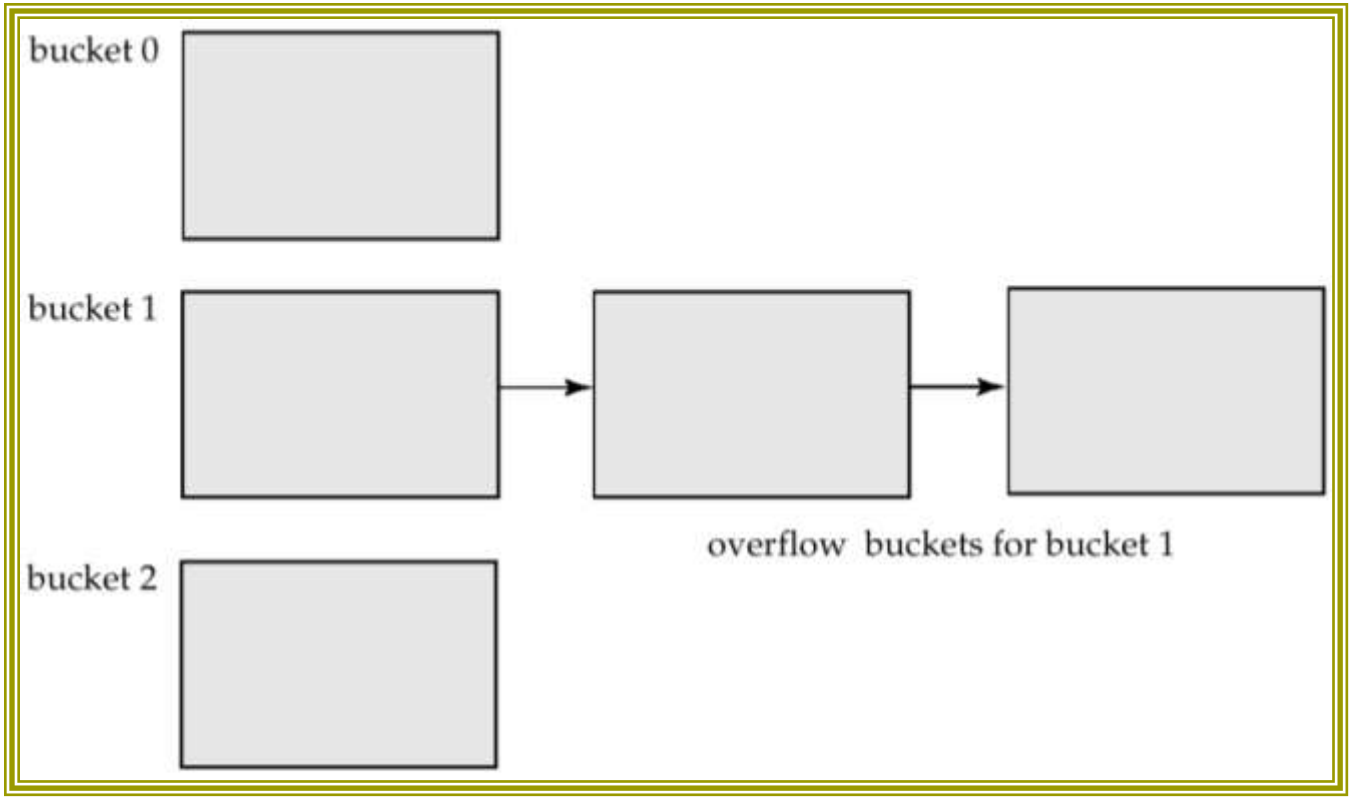
- Worst case has function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.

# Handling of Bucket Overflows

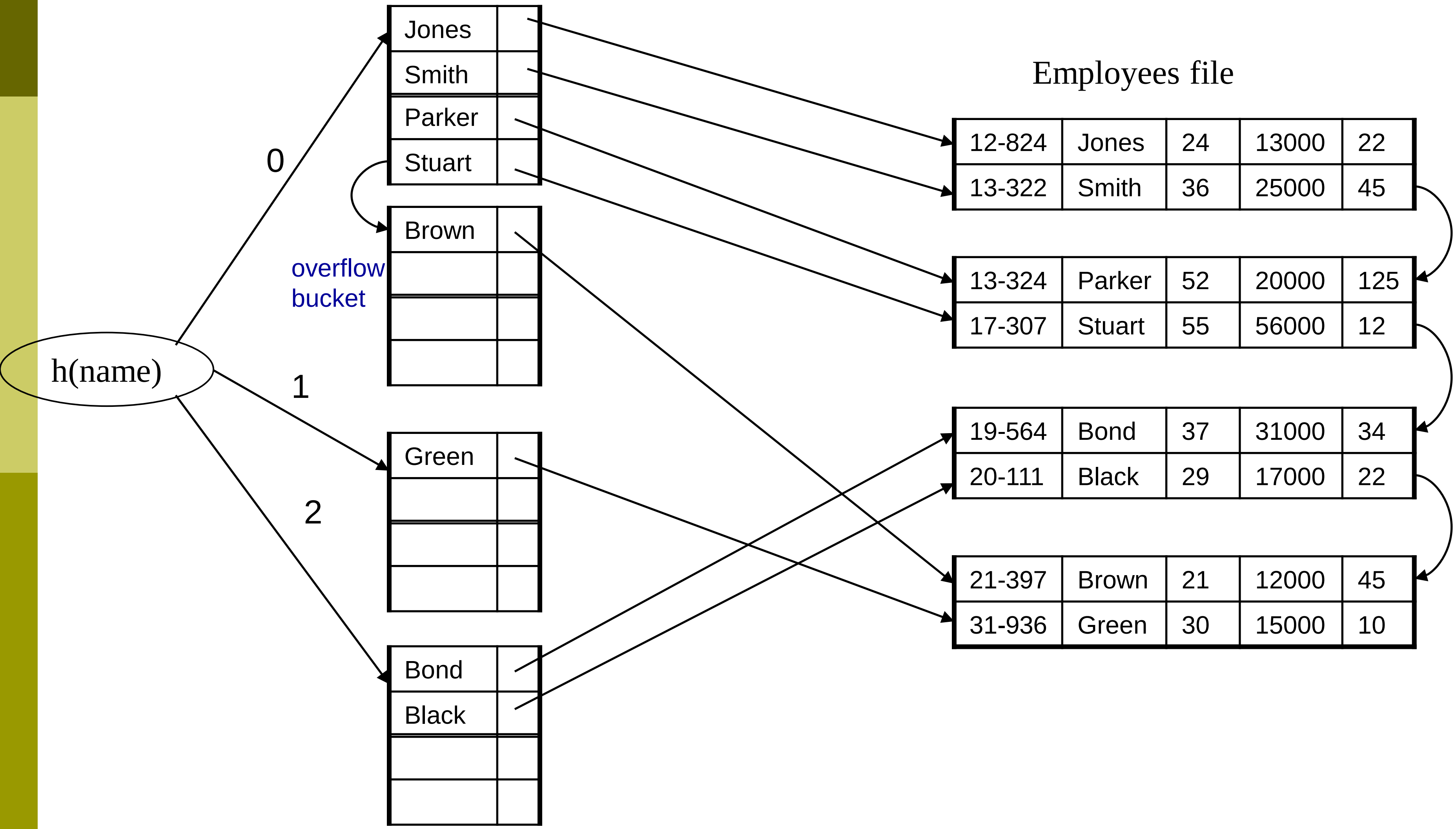
- Overflow chaining / closed hashing – the overflow buckets of a given bucket are chained together in a linked list.



# Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.

# Example of Hash Index



h(name)

0

1

2

| | |
|---|---|
| Jones | |
| Smith | |
| Parker | |
| Stuart | |

overflow bucket

| | |
|---|---|
| Brown | |
| | |
| | |
| | |

| | |
|---|---|
| Green | |
| | |
| | |
| | |

| | |
|---|---|
| Bond | |
| Black | |
| | |
| | |

Employees file

| | | | | |
|---|---|---|---|---|
| 12-824 | Jones | 24 | 13000 | 22 |
| 13-322 | Smith | 36 | 25000 | 45 |
| 13-324 | Parker | 52 | 20000 | 125 |
| 17-307 | Stuart | 55 | 56000 | 12 |
| 19-564 | Bond | 37 | 31000 | 34 |
| 20-111 | Black | 29 | 17000 | 22 |
| 21-397 | Brown | 21 | 12000 | 45 |
| 31-936 | Green | 30 | 15000 | 10 |

# Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically (**dynamic hashing**).

# Summary

- We have discussed 3 key topics of a modern relational database (RDB): database design, query evaluation, and indexing
- They form the foundation of our next topic -- spatial databases