Principles of Database Systems (CS307)

Lecture 13 - 1: Indexing

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Announcements

- Assignment on Trigger, due date: 23:59 on December 15th 2024, Beijing Time
    - Please do not miss the deadline

Outline

Part 1: Relational Language

Basic SQL
- Create Tables and Insert Data
- Basic Queries

Intermediate SQL
- Advanced Join
- Subquery
- Window Function
- *Transactions*

Advanced SQL
- Function
- Procedure
- Trigger
- View

Lab Only

Part 2: DB Design and Application

Relational DB Design
- The Entity-Relationship Model
- E-R Diagram Design

Relational Algebra

Normalization
- Functional Dependency
- Normal Forms
- FD Theory
- Decomposition w/ FD

Application Design
- Use Databases in Programs
- Complex Data Types (JSON, XML, etc.)

Part 3: Internal Mechanism Of Relational DB Systems

Storage Management
- Physical and Logical Storage Structure
- Indexing

Query Management
- Query Processing
- Query Optimization

Transaction Management
- Transaction
- Concurrency

Part 4: Miscellaneous Topics

Big Data
- Big Data Storage
- Query and Process Big Data

Advanced Topics
- Beyond Relational DB
- Data Analytics

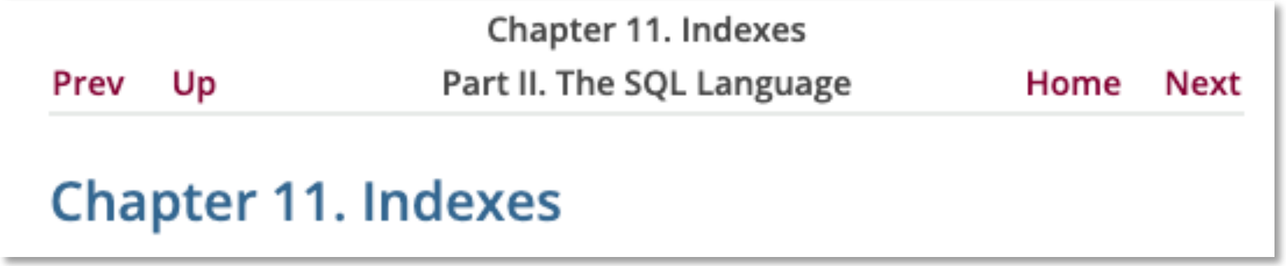
Introduction

Motivation

- Many queries reference only a small proportion of the records in a file
  - E.g., find all instructors in the Physics department
    - Inefficient to read every tuple in the instructor relation to check if the dept name value is “Physics”
    - The system should be able to locate these records directly
- Think about an example in a library:
  - How can we find a book?
    - Books are on the shelves in a sequential order
    - We had drawers where you could look for books by author, title or sometimes subject that were telling you the "***coordinates***" of a book
  - author, title and subject are like indexes (索引)

Terminology

- Plural of index: indices, or indexes?
  - Both are correct in English
    - indices (Latin): Often used in scientific and mathematical context representing the places of an element in an array, vector, matrix, etc.
    - indexes (American English): Used in publishing for the books
  - What about database?
    - A good way: Follow the naming convention of the project or the DBMS



https://www.postgresql.org/docs/current/indexes.html

Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the <u>search key</u>

```
1,12 stulyev,ru,1971,161
2,Al-mummia test,eg,1969,102
3,"Ali Zaoua, prince de la rue",ma,2000,90
4,Apariencias,ar,2000,94
5,Ardh Satya,in,1983,130
6,Armaan,in,2003,159
7,Armaan,pk,1966,
8,Babettes gæstebud,dk,1987,102
9,Banshun,jp,1949,108
10,Bidaya wa Nihaya,eg,1960,
11,Variety,us,2008,106
12,"Bon Cop, Bad Cop",ca,2006,
13,Brilliantovaja ruka,ru,1969,100
14,C'est arrivé près de chez vous,be,1992,95
15,Carlota Joaquina - Princesa do Brasil,br,1995,
16,Cicak-man,my,2006,107
```

Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key
    - E.g., find movies with IDs larger than 100 and smaller than 200

```
1,12 stulyev,ru,1971,161
2,Al-mummia test,eg,1969,102
3,'Ali Zaoua, prince de la rue",ma,2000,90
4,Apariencias,ar,2000,94
5,Ardh Satya,in,1983,130
6,Armaan,in,2003,159
7,Armaan,pk,1966,
8,Babettes gæstebud,dk,1987,102
9,Banshun,jp,1949,108
10,Bidaya wa Nihaya,eg,1960,
11,Variety,us,2008,106
12,"Bon Cop, Bad Cop",ca,2006,
13,Brilliantovaja ruka,ru,1969,100
14,C'est arrivé près de chez vous,be,1992,95
15,Carlota Joaquina - Princesa do Brasil,br,1995,
16,Cicak-man,my,2006,107
```

In the current storage structure, the records are sorted by `movieid`

- So, it will be easy to find a specific `movieid` with binary search

Searching for Record

- Remember searching algorithms in Data Structure?
    - Linear search
        - Scan all records from top to bottom
    - Binary search
        - Divide and conquer
        - Assumption: Records are **sorted** by the search key

- However, how can we find data based on the non-sorted columns?
    - E.g., find all Chinese movies

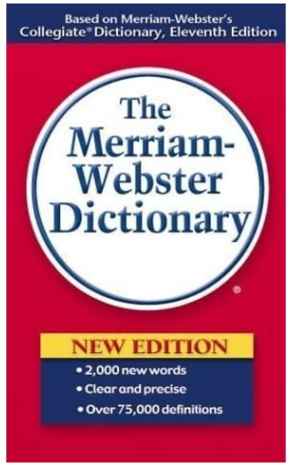
```
1,12 stulyev,ru,1971,161
2,Al-mummia test,eg,1969,102
3,"Ali Zaoua, prince de la rue",ma,2000,90
4,Apariencias,ar,2000,94
5,Ardh Satya,in,1983,130
6,Armaan,in,2003,159
7,Armaan,pk,1966,
8,Babettes gæstebud,dk,1987,102
9,Banshun,jp,1949,108
10,Bidaya wa Nihaya,eg,1960,
11,Variety,us,2008,106
12,"Bon Cop, Bad Cop",ca,2006,
13,Brilliantovaja ruka,ru,1969,100
14,C'est arrivé près de chez vous,be,1992,95
15,Carlota Joaquina - Princesa do Brasil,br,1995,
16,Cicak-man,my,2006,107
```

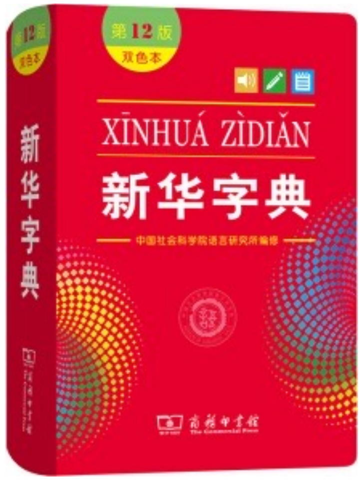
`country`

Find the rows `where country = 'cn'`

- The country codes are not sorted in the current storage structure, so the binary search algorithm cannot be used

Searching for Record





- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are usually sorted in the alphabetical order of Pinyin
    - However, we have other ways of looking up a character
      - Radicals (偏旁部首)
      - Number of strokes (数笔画)
      - Four-corner method (四角号码)

Practical Use

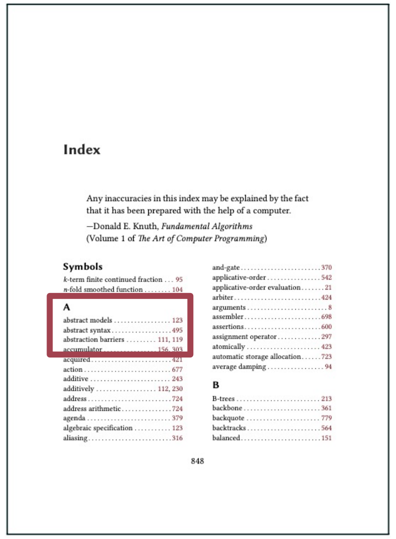
Index in Databases

- Concept
  - An **index** is a data structure which improves the efficiency of retrieving data with specific values from a database

  - Usually, indexes locate a row by a series of location indicators
    - E.g., (filename, block number, offset)

Index in Databases

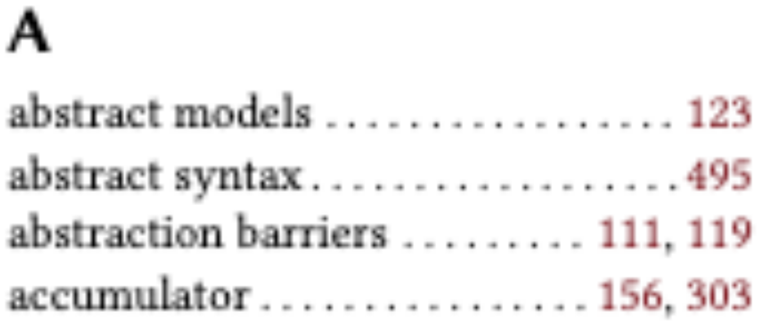
- Concept
  - An **index** is a data structure which <u>improves the efficiency of retrieving data</u> with <u>specific values</u> from a database
  - Usually, indexes locate a row by a series of <u>location indicators</u>
    - E.g., (filename, block number, offset)
  - It is like indexes in books
    - Location indicator: (page, row)



Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.
—Donald E. Knuth, *Fundamental Algorithms*
(Volume 1 of *The Art of Computer Programming*)

848



Index in Databases

- Actually, we have been benefited from indexes off-the-shelf

  - In PostgreSQL, indexes are built <u>automatically</u> on columns with `primary key` or `unique` constraints

Experiment on Using Indexes

- Duplicate a table with no index

```
create table movies_no_index as select * from movies;
```

```
-- auto-generated definition
create table movies_no_index
(
    movieid       integer,
    title         varchar(100),
    country       char(2),
    year_released integer,
    runtime       integer,
    user_name     varchar(20)
);
```

Experiment on Using Indexes

- Check the performance on retrieving data
  - Significant difference between queries on the two tables

```sql
-- Query 1
explain analyze
    select *
    from movies
    where movieid > 100 and movieid < 300;

-- Query 2
explain analyze
    select *
    from movies_no_index
    where movieid > 100 and movieid < 300;
```

Query 1
(on `movies`)

	QUERY PLAN
1	Bitmap Heap Scan on movies (cost=10.32..136.35 rows=199 width=40) (actual time=0.162..0.440 rows=199 loops=1)
2	Recheck Cond: ((movieid > 100) AND (movieid < 300))
3	Heap Blocks: exact=6
4	-> Bitmap Index Scan on movies_pkey (cost=0.00..10.28 rows=199 width=0) (actual time=0.136..0.136 rows=199 loops=1)
5	Index Cond: ((movieid > 100) AND (movieid < 300))
6	Planning Time: 0.413 ms
7	Execution Time: 0.507 ms

Query 2
(on `movies_no_index`)

	QUERY PLAN
1	Seq Scan on movies_no_index (cost=0.00..217.06 rows=199 width=40) (actual time=0.039..5.075 rows=199 loops=1)
2	Filter: ((movieid > 100) AND (movieid < 300))
3	Rows Removed by Filter: 9005
4	Planning Time: 0.444 ms
5	Execution Time: 5.156 ms

Experiment on Using Indexes

- If there is no index on a column (or several columns), we can create one manually

```sql
-- SQL Syntax for creating indexes
create index index_name
on table_name (column_name [, ...]);
```

Theoretical Aspects

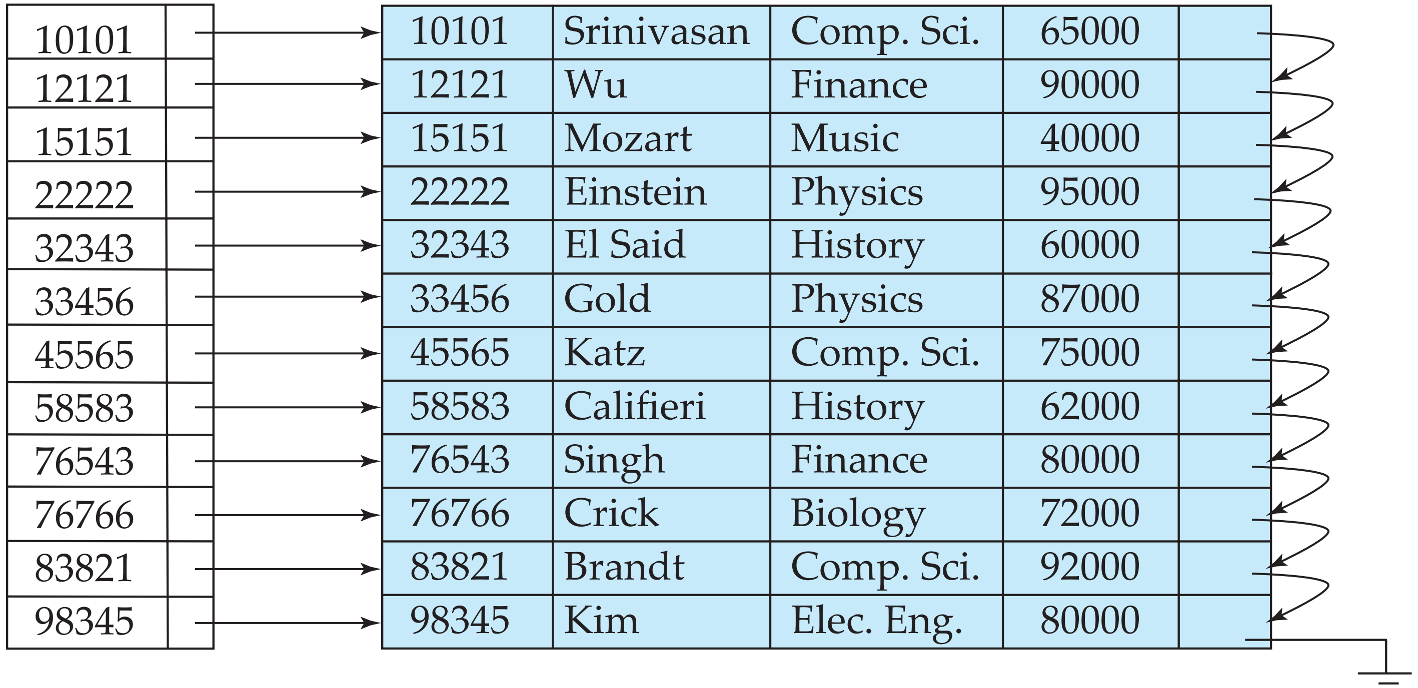
Index Taxonomy

- 1) In terms of storage structure, is the index completely separated with the data records?
  - No ⇒ **Integrated index**
    - PK (primary key) index in a MySQL InnoDB database
    - PK index in a SQL Server database
  - Yes ⇒ **External index**
    - Indexes in a PostgreSQL database
    - Indexes in a MySQL MyISAM database

Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
    - Allows the records of a file to be read in an order corresponding to the physical order in the file
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

Example of clustered index

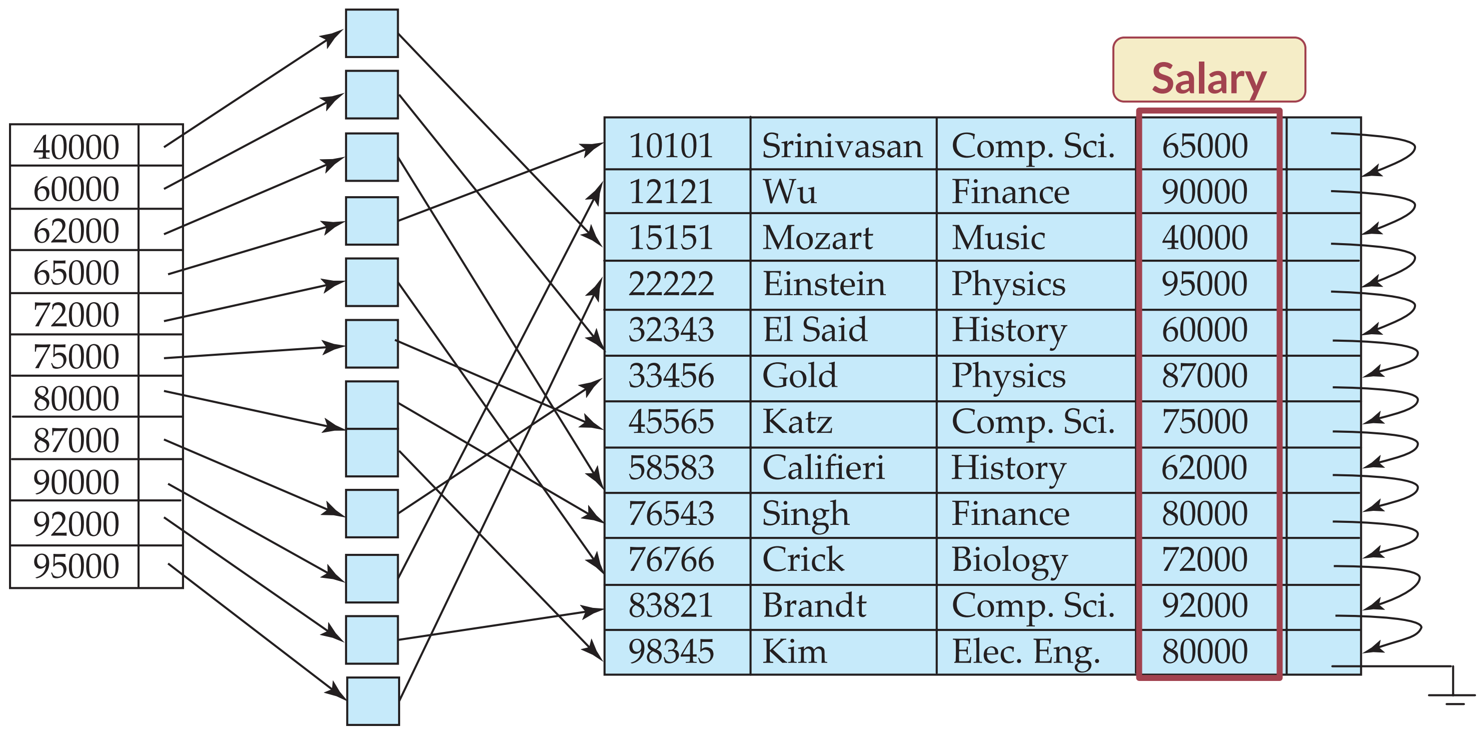


Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
    - Allows the records of a file to be read in an order corresponding to the physical order in the file
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

A secondary index on the column “salary”

- Index record points to a bucket that contains <u>pointers</u> to all the actual records with that <u>particular search-key value</u>
- Secondary indices have to be dense

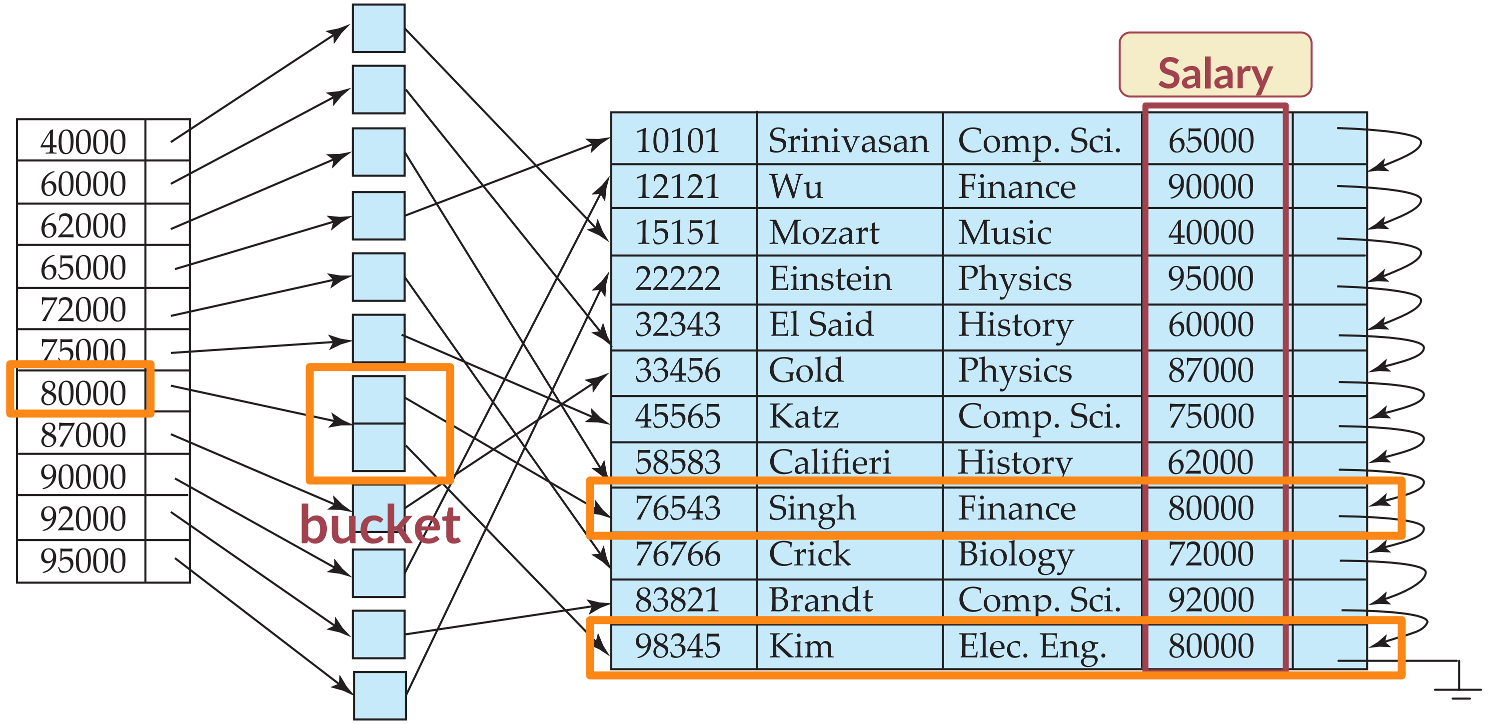


Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
    - Allows the records of a file to be read in an order corresponding to the physical order in the file
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

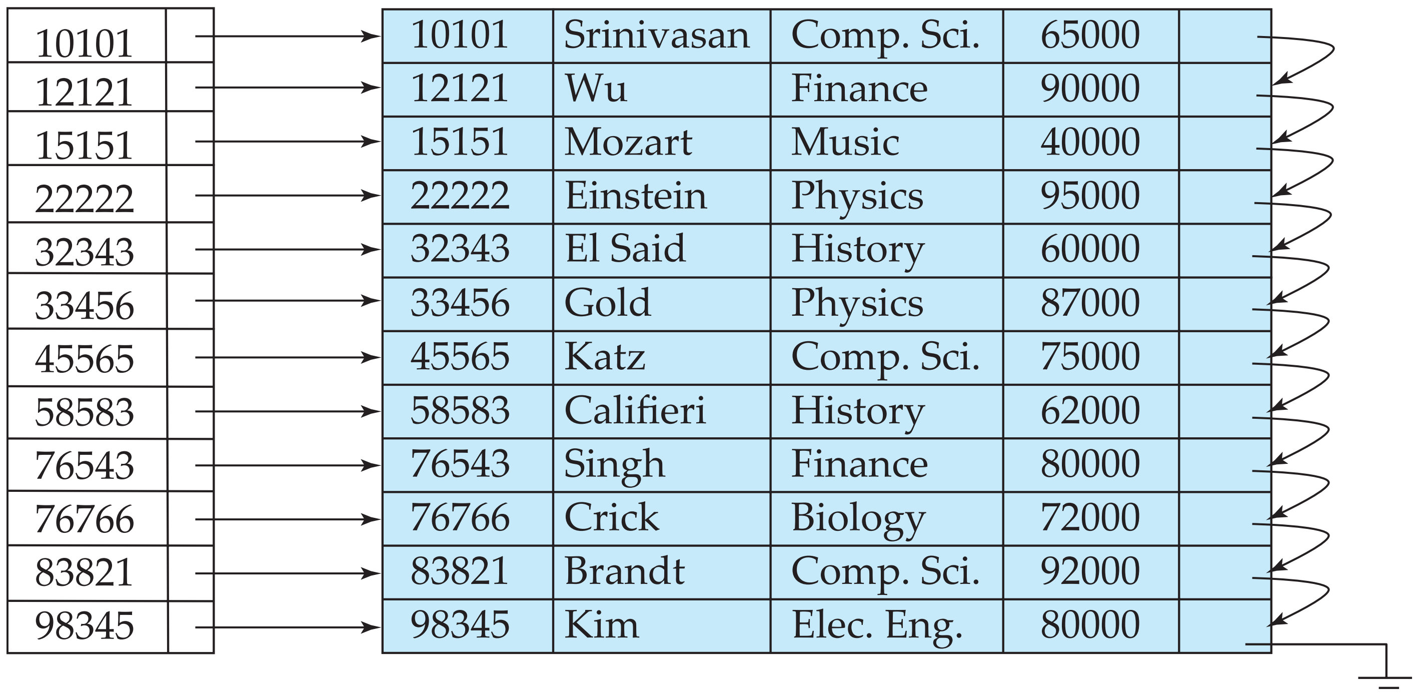
A secondary index on the column “salary”

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value
- Secondary indices have to be dense

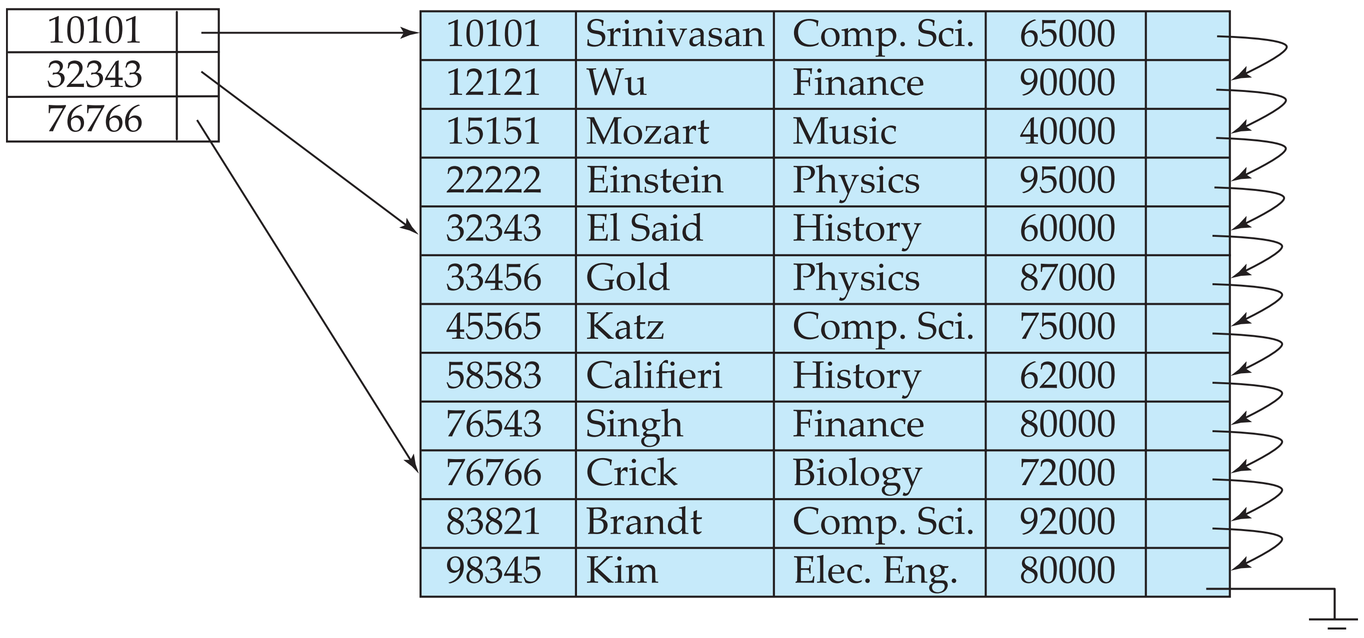


Index Taxonomy

- 3) Does every search key in the data file correspond to an index entry?
  - Yes ⇒ **Dense Index**
  - No ⇒ **Sparse Index**



Dense Index



Sparse Index

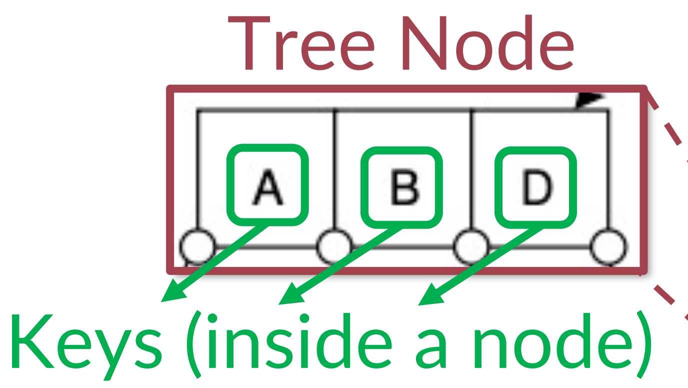
Index Taxonomy

- 4) Does the search key contain more than one attribute?
  - Yes ⇒ **Multi-key index** (Multi-column index)
  - No ⇒ **Single-key index** (Single-column index)
    - *We mainly focus on single-key index for now*

Index Implementation

- Data Structures for Indexes
  - B-tree, B+-tree
    - Very famous data structures for building indexes
  - Hash table

B-tree



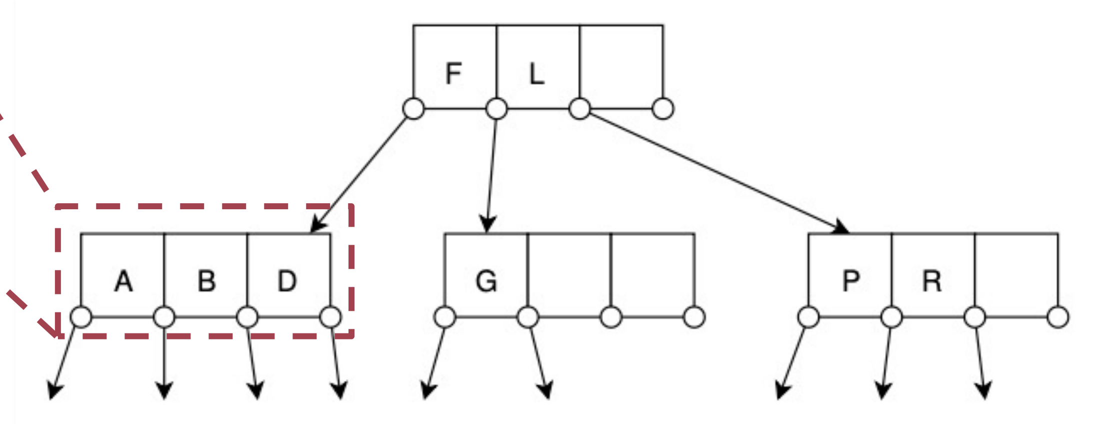


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

- A B-tree of order ***m*** satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing n keys ($K_1 < K_2 < K_3 < \cdots < K_n$) with n+1 children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, $[m/2] \leq$ # of children $\leq m$
    - … except that <u>a root node</u> may have less than $[m/2]$ children
  - (**Always balanced**) All leaves appear on the same level

B-tree

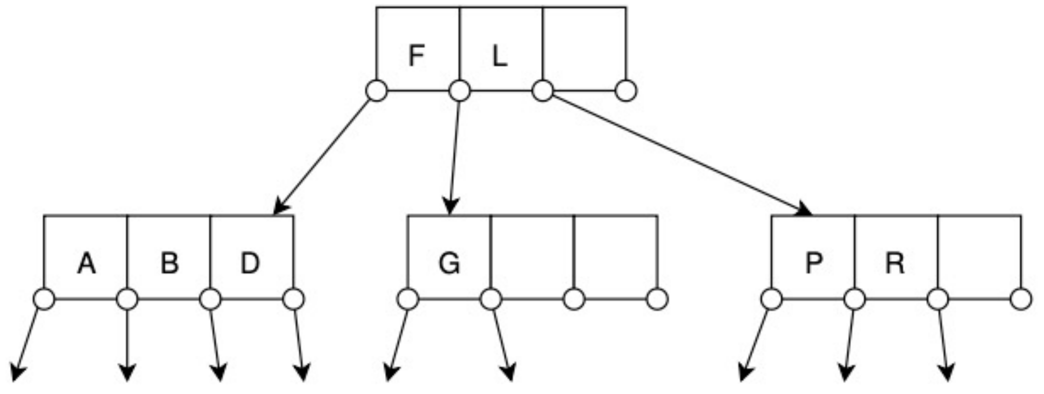


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

- A B-tree of order *m* satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing n keys $(K_1 < K_2 < K_3 < \cdots < K_n)$ with $n+1$ children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, $[m/2] \leq$ # of children $\leq m$
    - ... except that <u>a root node</u> may have less than $[m/2]$ children
  - (**Always balanced**) All leaves appear on the same level

B-tree

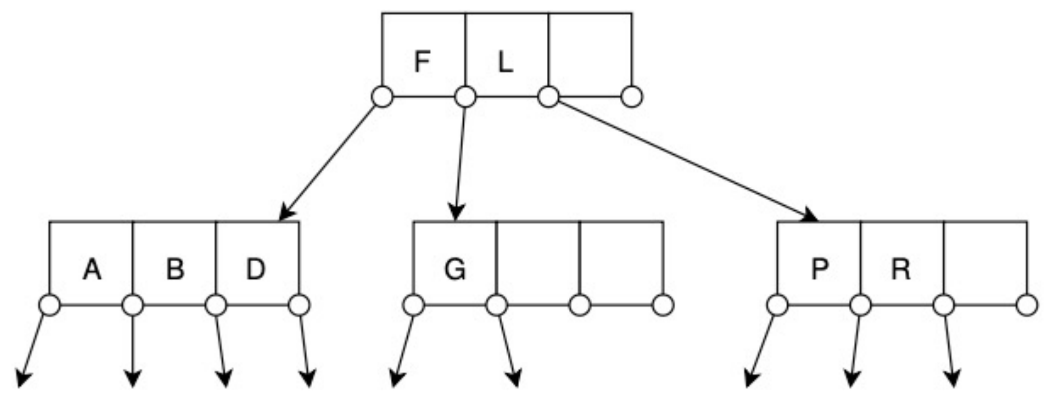


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

- A B-tree of order ***m*** satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing n keys ($K_1 < K_2 < K_3 < \cdots < K_n$) with n+1 children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, [m/2] ≤ # of children ≤ m
    - … except that a root node may have less than [m/2] children
  - (**Always balanced**) All leaves appear on the same level

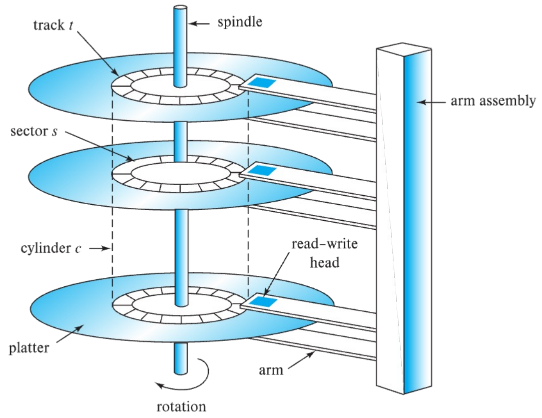
- [m/2] is the called the minimum branching factor (a.k.a. minimum degree) of the tree
- A B-tree of order m is usually called a “[m/2]-m tree”, like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, . . .
  - In practice, the order m is much larger (~100)

B-tree

- Height of a *B*-tree: $h \leq 1 + \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right)$
  - If we take an 50-100 tree with 1M records:
    - $h \leq 1+\log_{100/2}(1000000/2) = 4.354$ (i.e., 4 levels)

B-tree

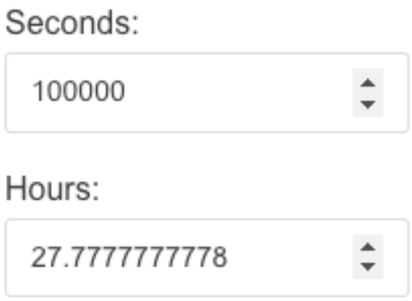
- Height of a B-tree: $h \leq 1 + \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right)$
  - If we take an 50-100 tree with 1M records:
    - $h \leq 1+\log_{100/2}(1000000/2) = 4.354$ (i.e., 4 levels)
  - Why do we use B-trees?
    - We can set the size of a B-tree node as the disk page size
      - i.e., m can be chosen with consideration on the page size
    - The height of the tree -> Number of disk I/Os
      - The number of disk I/Os can be relatively small



Access time: 5-20ms

1ns = 10^{-6}ms

Access time: 50-70ns



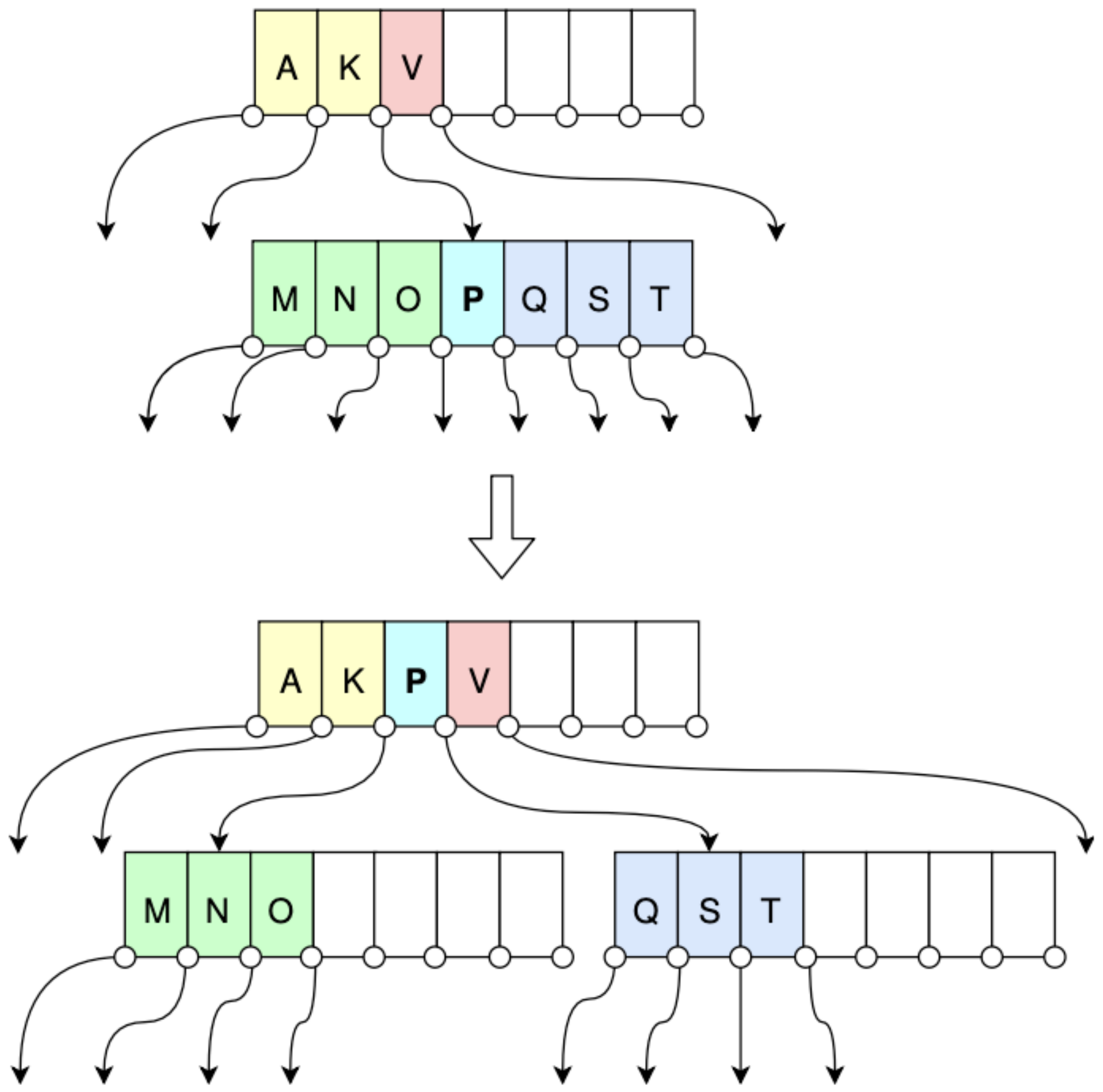
B-tree

- Tree operations:
  - Search, Insert, Delete
  - Update (Delete + Insert)

- What is special in B-tree
  - Split and merge nodes

B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

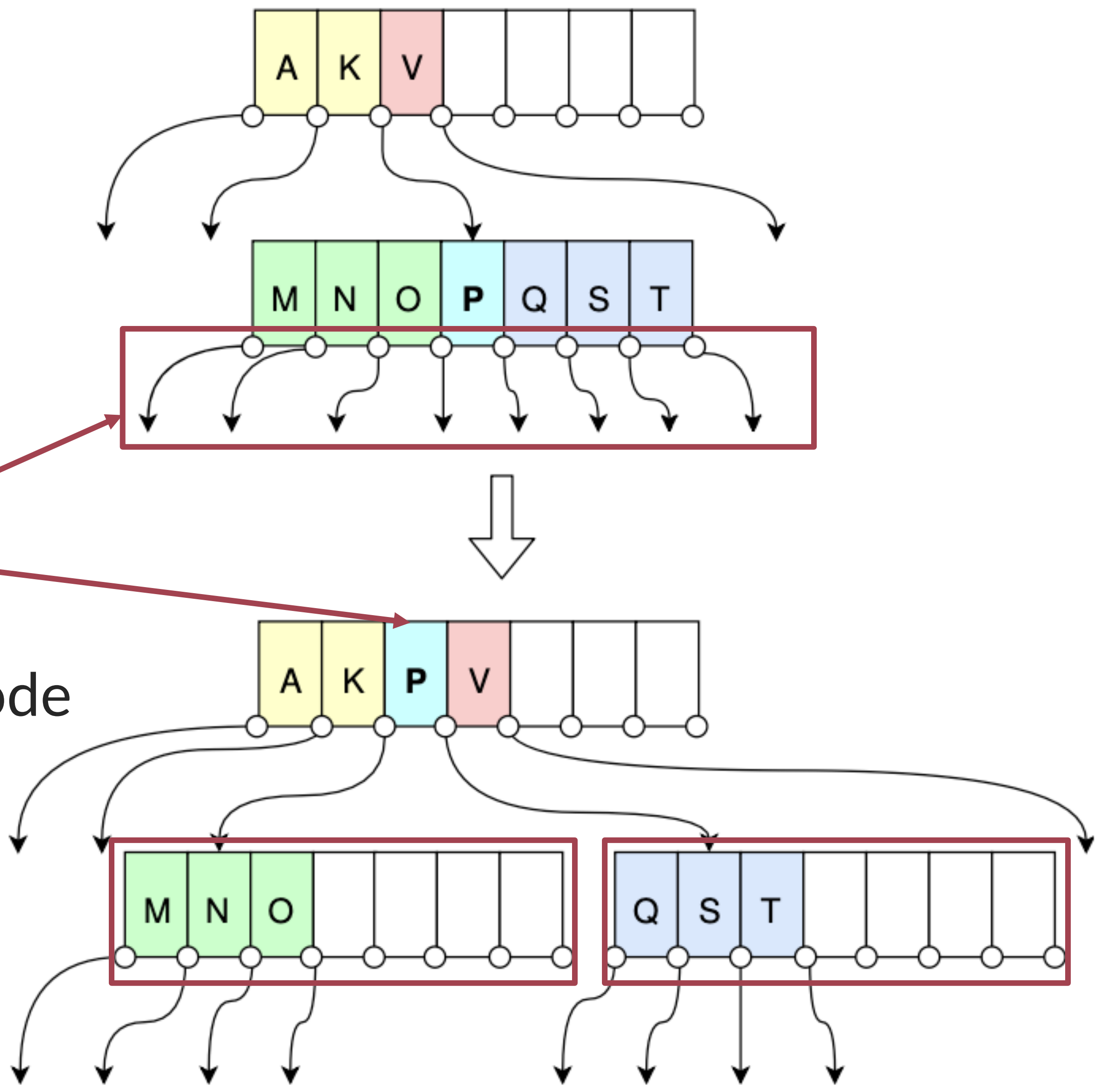


B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

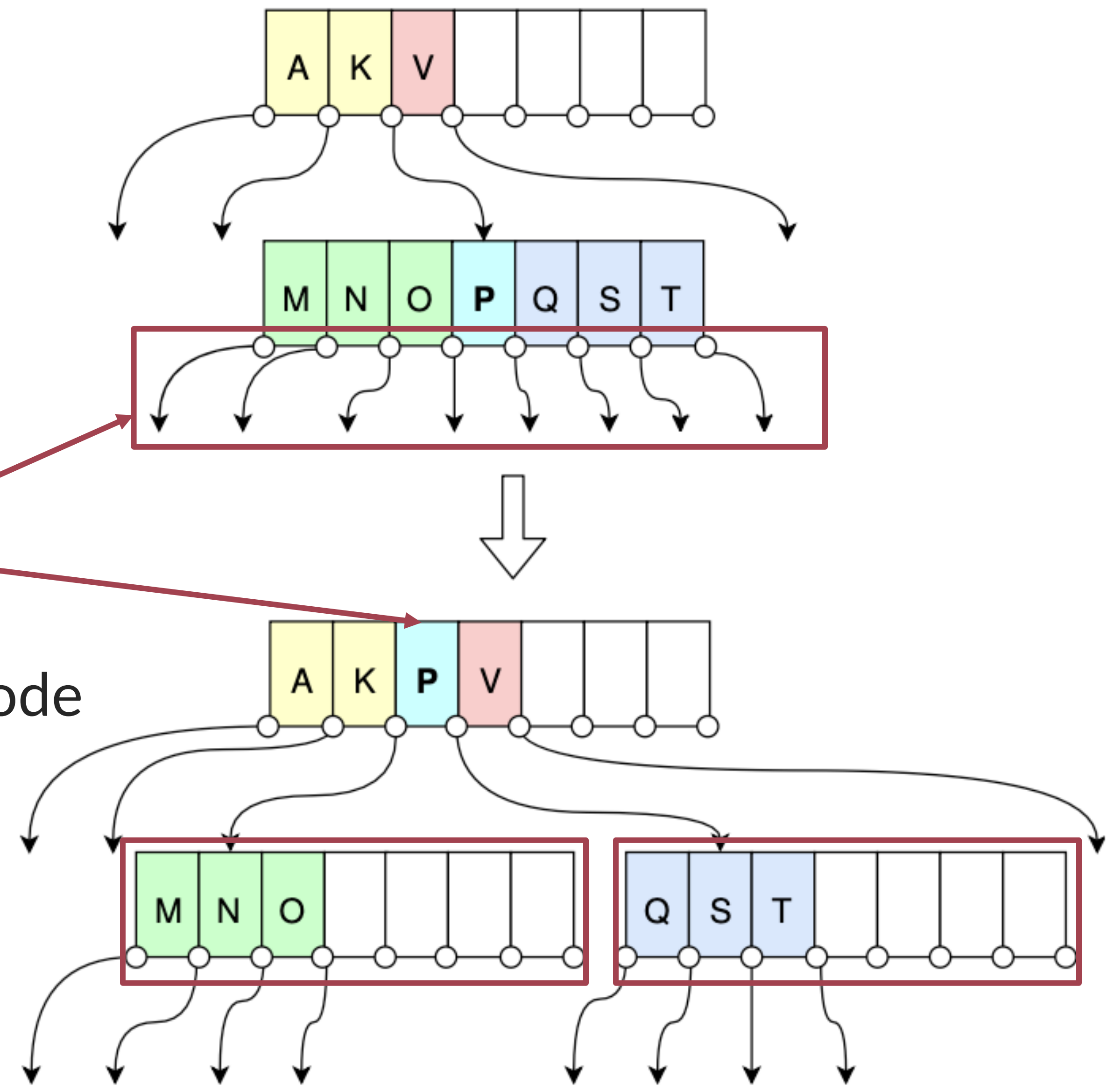
The number of children is larger than m (m=7)

- This node will be split into two nodes
- The pivot key will be elevated into the parent node



B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"



The number of children is larger than m (m=7)

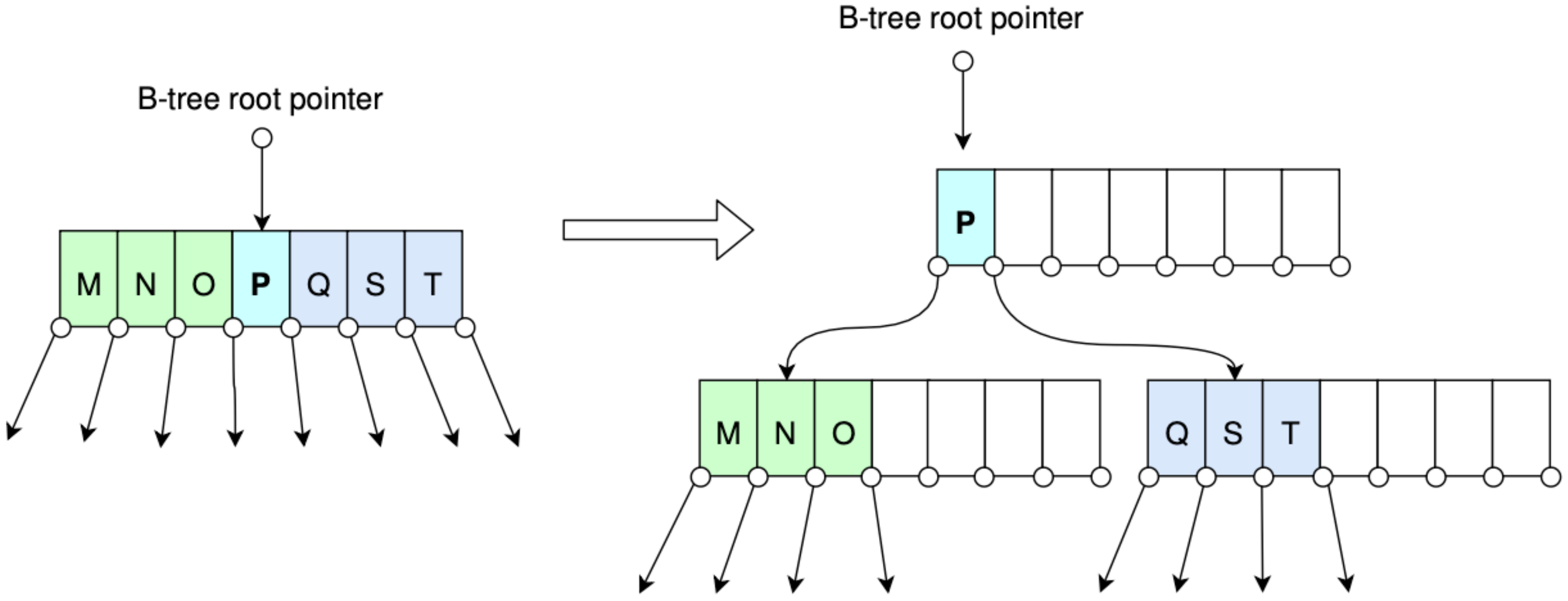
- This node will be split into two nodes
- The pivot key will be elevated into the parent node

  - What if the parent (or even the root) node is also full?

B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

- Split the root node of the B-tree

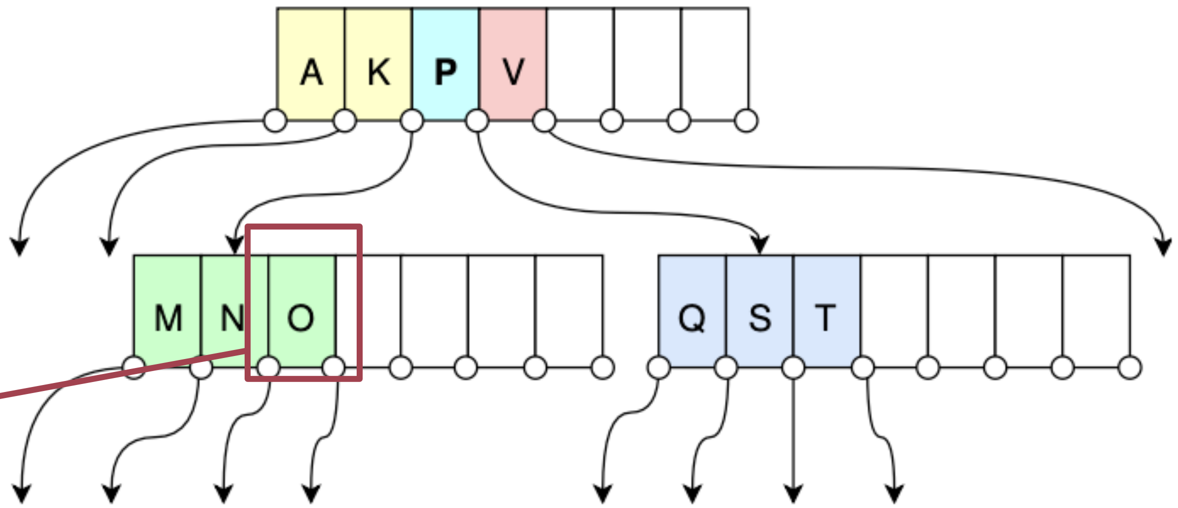


Note that the height of the B-tree is increased by 1 in this case

- This is the only way that a B-tree increases its height

B+-tree

- A Problem in B-tree: **Table traversal** when <u>only the B-tree is provided</u>
  - In B-tree, data are stored on all nodes
  - What if we want to traverse all records in the table?
    - `select * from letters`

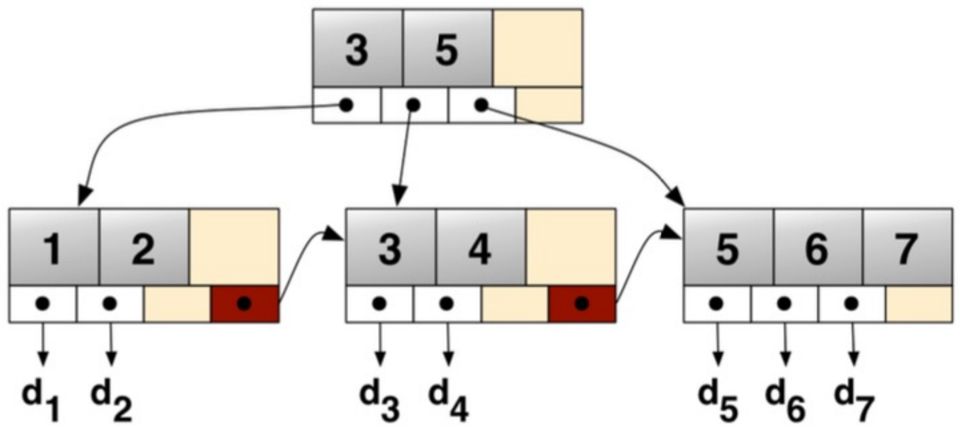


For example, we have accessed the node for letter "O"

- How can we find the next row?
  - We must go back to the parent node to access "P" (extra time cost)

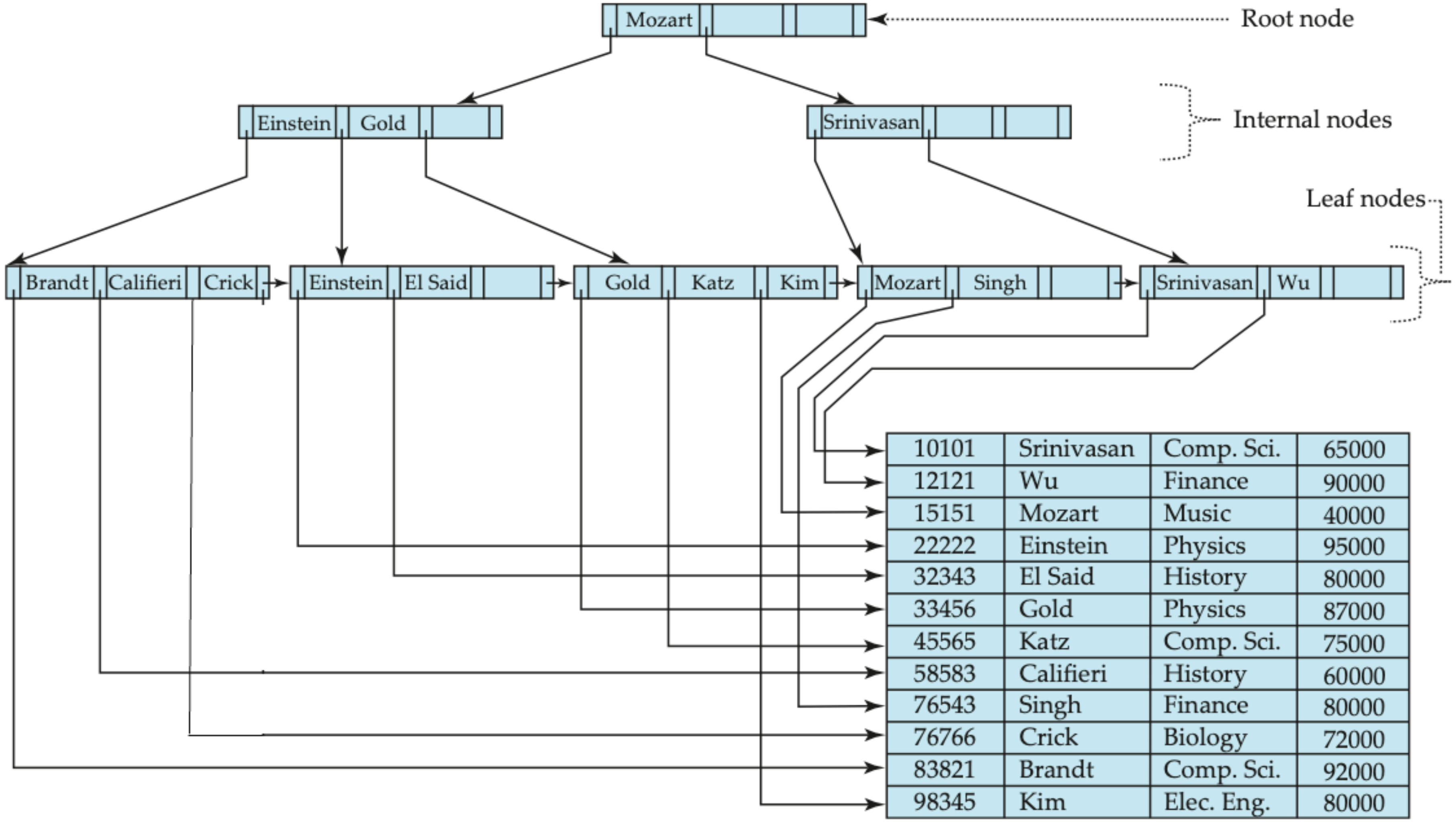
B+-tree

- Features of a B+-tree (compared with B-trees)
  - Data stored only in leaves
  - Leaves are linked sequentially



B+-tree

- A complete example of a B+-tree
  - Data stored only in leaves
    - No need to squeeze data into non-leaf nodes
  - Leaves are linked sequentially
    - Faster table traversal from top to bottom
    - Better support for range queries



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Index It or Not: Where Indexing May Help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria

  - `=`
  - `<, <=, >, >=, between`
  - `in`

  - `exists`
  - `like` (prefix matching)

- Be cautious when the indexed columns need frequent writing operations
  - Overhead to update indexes in `insert`, `update`, and `delete` operations
- Functions

```sql
SELECT attr1, attr2
FROM table
WHERE function(column) = search_key
```

```sql
-- Create an index on the return values of the function
-- instead of the original values
create index idx_name ON table1(function(col1));
```

Note: The expression should be deterministic. For detailed usage, please refer to:
`https://www.postgresql.org/docs/14/indexes-expressional.html`

Index It or Not: Where Indexing May Help

- Be cautious when using indexes on a small table
  - Full scan ≠ Bad scheme
  - Index retrieval ≠ Good scheme

Hashing

- Hashing is a widely used technique for building indexes

- A **bucket** is a unit of storage containing one or more entries
  - A bucket is typically a disk block
  - We obtain the bucket of an entry 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
    - Hash function is used to locate entries for access, insertion as well as deletion.

- Entries with different search-key values may be mapped to the same bucket
  - … thus, the entire bucket must be searched sequentially to locate an entry.

Hashing Index & Hashing File Organization

- In a hash index, buckets store entries with pointers to records

bucket1

```
15151: block 1, offset 0
```

bucket2

```
32343: block 3, offset 0
58583: block 3, offset 24
```

- In a hash file-organization, buckets store records

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Example: How `join` Works (with the help of indexes)

- Some widely used `join` algorithms
  - Nested-loop join
  - Hash join
  - Sort-merge join

Example: How `join` Works (with the help of indexes)

- Nested (loop) join
  - Straight-forward linking between records from two tables in a nested-loop manner

```
for each row in t1 match C1(t1)
    for each row in t2 match P(t1, t2)
        if C2(t2)
            add t1|t2 to the result
```

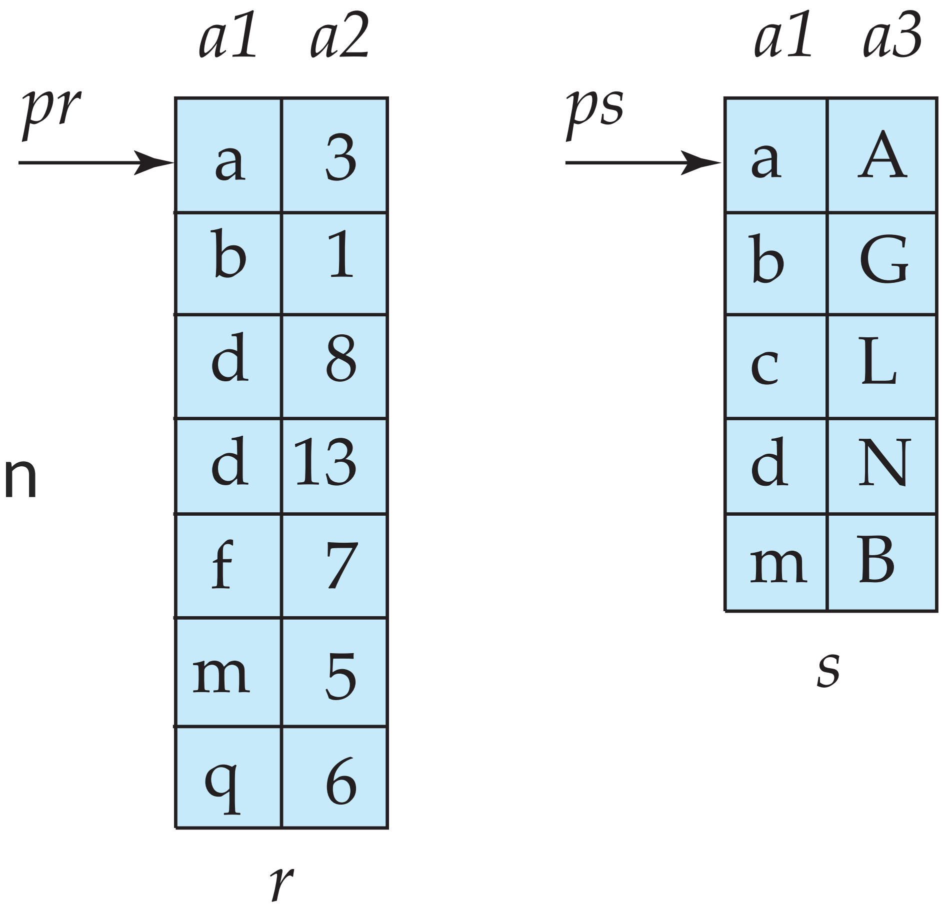
Example: How `join` Works (with the help of indexes)

- Hash join
  - Build a set of buckets for a smaller table to speed up the data lookup

- Procedure:
  - 1. Create a hash table for the smaller table t1 in the memory
  - 2. Scan the larger table t2. For each record r,
    - 2.1 Compute the hash value of r.`join_attribute`
    - 2.2 Map to corresponding rows in t1 using the hash table

Example: How `join` Works (with the help of indexes)

- Sort-merge join (a.k.a. merge join)
  - Zipper-like joining
- Procedure:
  - 1. Sort tables t1 and t2 respectively according to the join attributes
  - 2. Perform an interleaved scan of t1 and t2. When encountering a matched value, join the related rows together.



When there are clustered indexes on the join attributes, step 1, the most expensive operation, can be skipped because t1 and t2 are already sorted in this scenario.

Principles of Database Systems (CS307)

Lecture 13 - 2: Query Processing

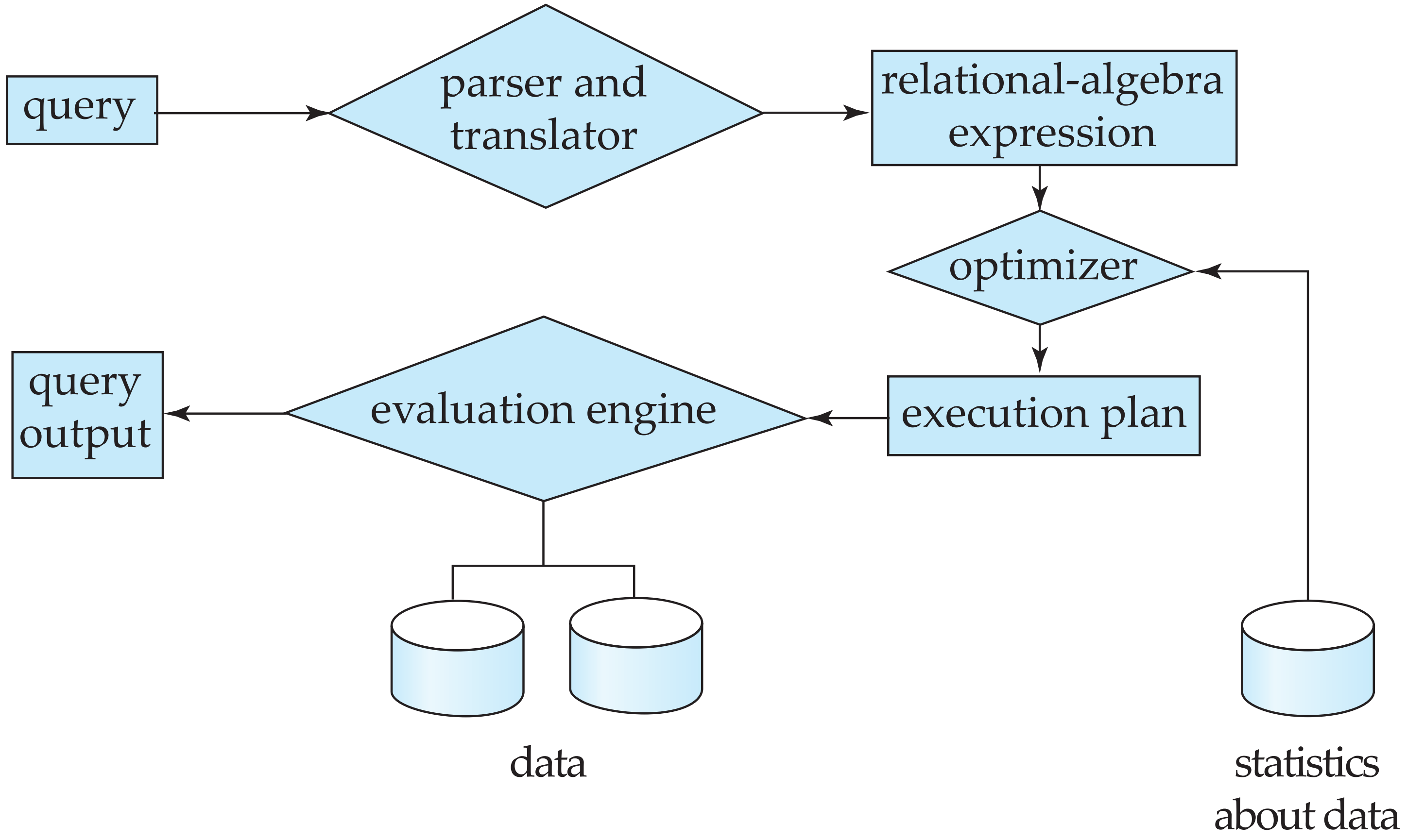
Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

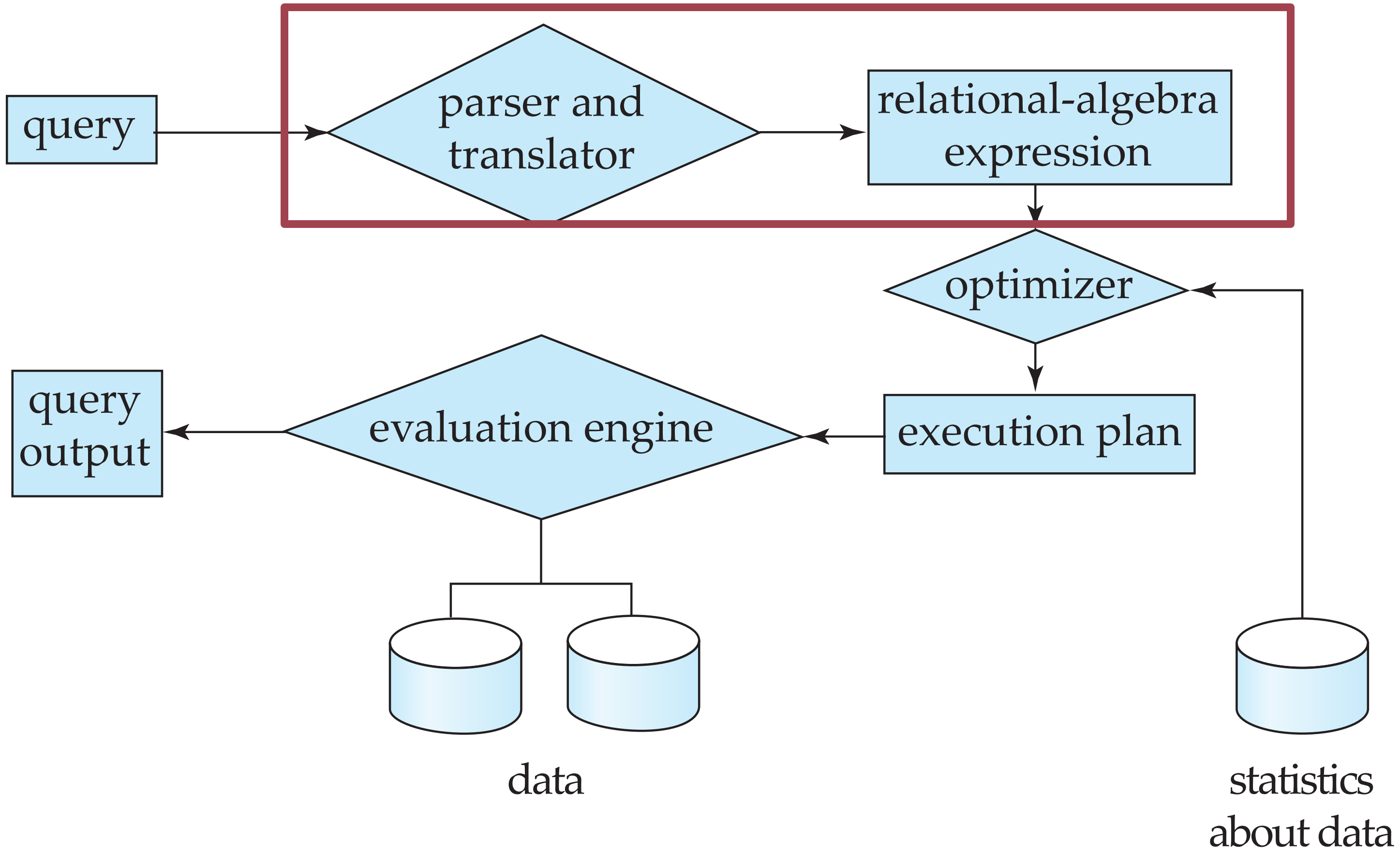
Basic Steps in Query Processing

- Parsing and Translation
- Optimization
- Evaluation



Basic Steps in Query Processing

- Parsing and Translation
  - Translate the query into its internal form
    - The internal form is then translated into relational algebra
  - Parser checks syntax and verifies relations



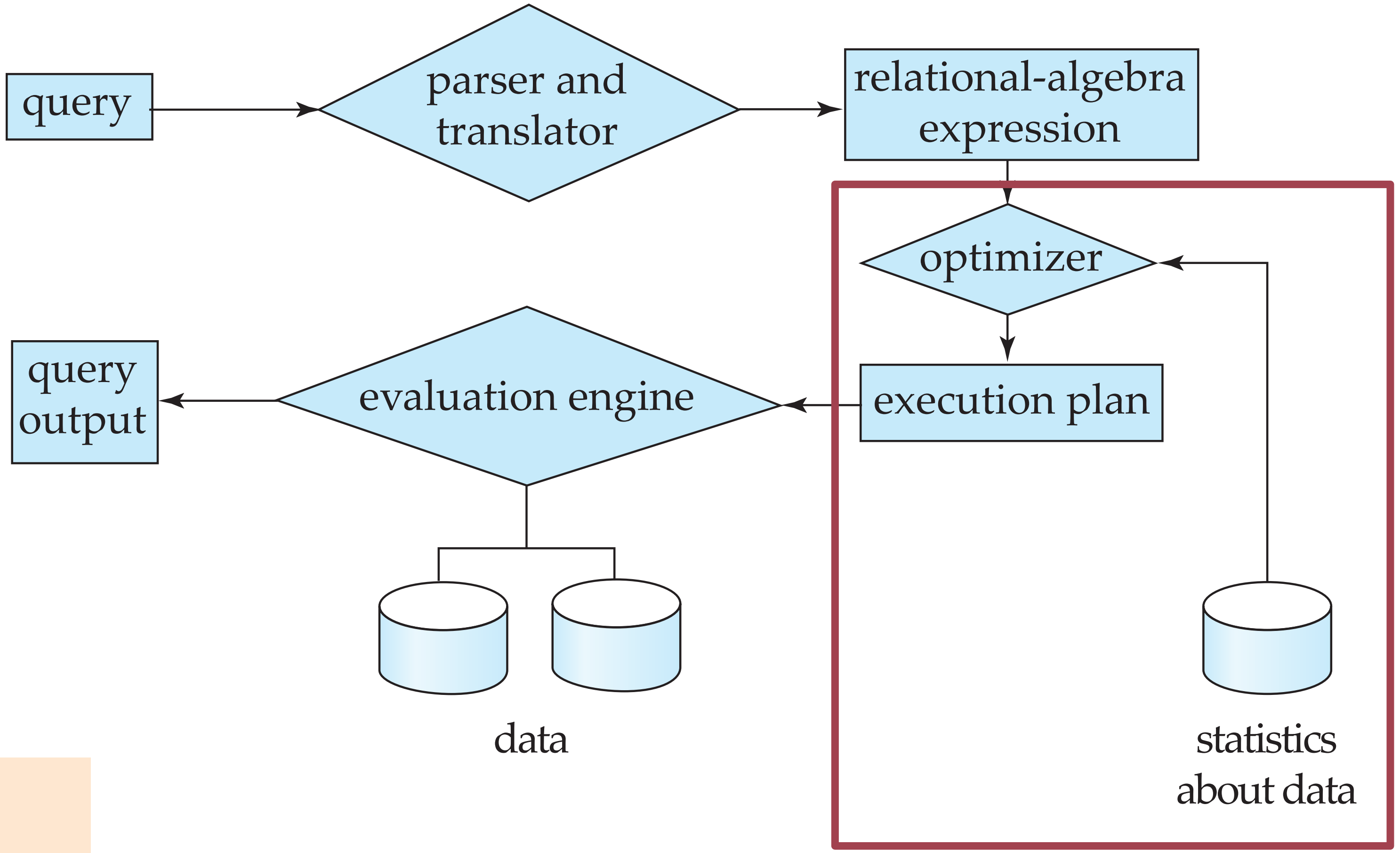
Basic Steps in Query Processing

- Optimization
  - A relational algebra expression may have many equivalent expressions
  - E.g.,

$$\sigma_{salary<75000}(\Pi_{salary}(\text{instructor}))$$

is equivalent to

$$\Pi_{\text{salary}}(\sigma_{\text{salary}<75000}(\text{instructor}))$$



But the number of rows involved in the projection operation may be (significantly) smaller in the second expression

Basic Steps in Query Processing

- Optimization
  - A relational algebra expression may have many equivalent expressions
    - … and each relational algebra operation can be evaluated using one of several different algorithms

  - *Correspondingly, a relational-algebra expression can be evaluated in many ways*

Basic Steps in Query Processing

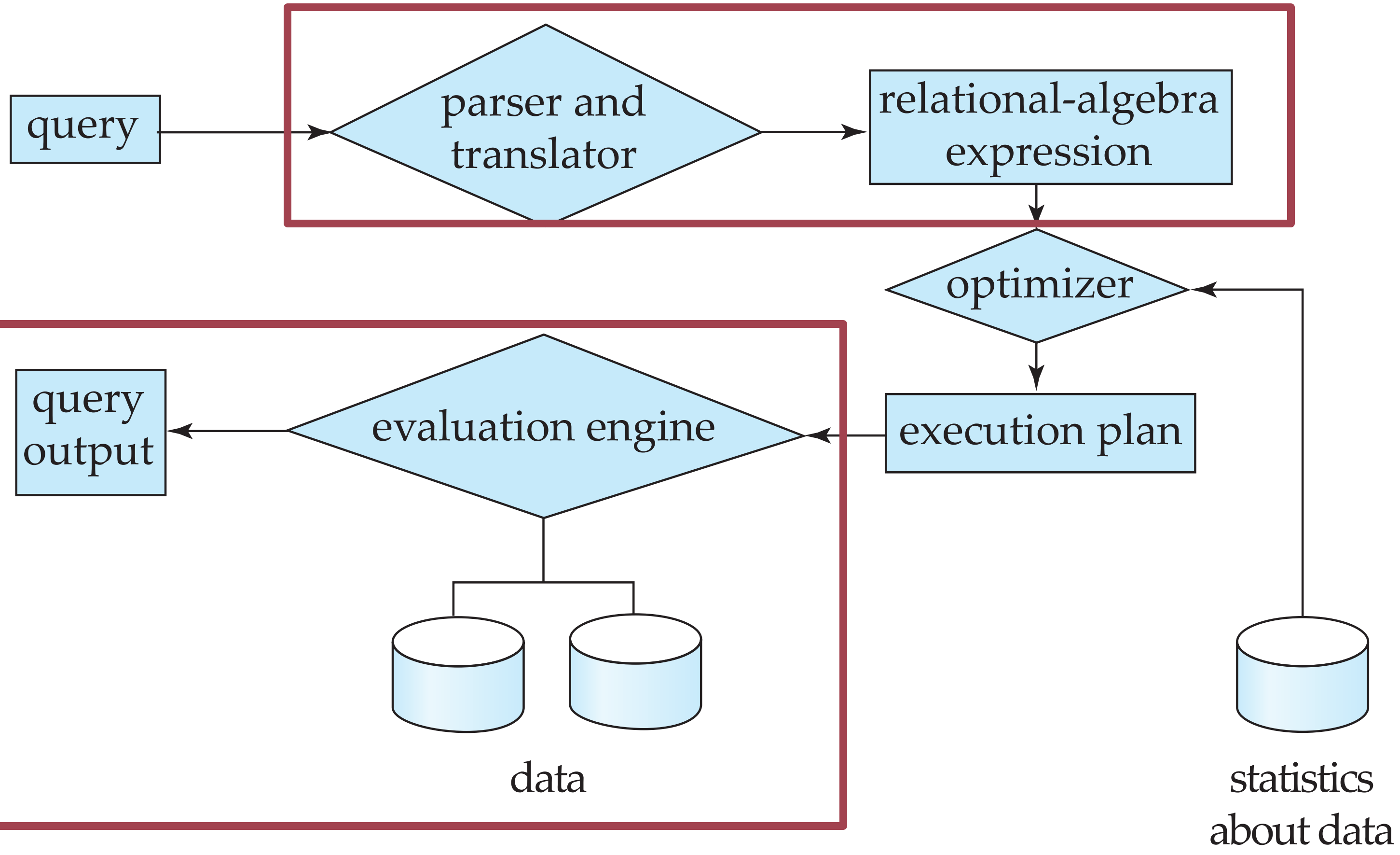
- Optimization
  - **Evaluation Plan**: Annotated expression specifying detailed evaluation strategy
  - E.g.,:
    - Use an index on `salary` to find instructors with `salary<75000`
    - Or perform complete relation scan and discard instructors with `salary<75000`

Query Optimization: Choose the one with the lowest cost among all equivalent evaluation plans

- Cost can be estimated using statistical information from the database catalog
  - E.g., Number of tuples in each relation, size of tuples, etc.

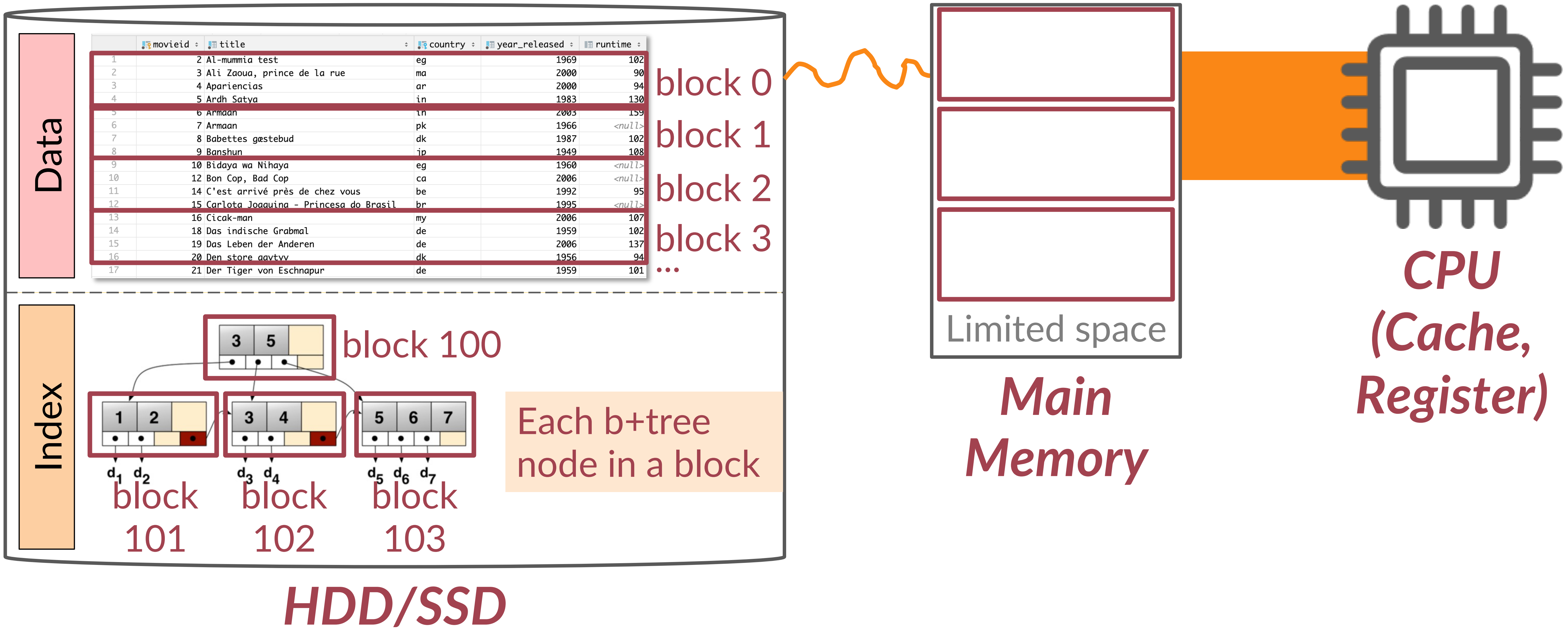
Basic Steps in Query Processing

- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query



Prerequisite

- Storage model

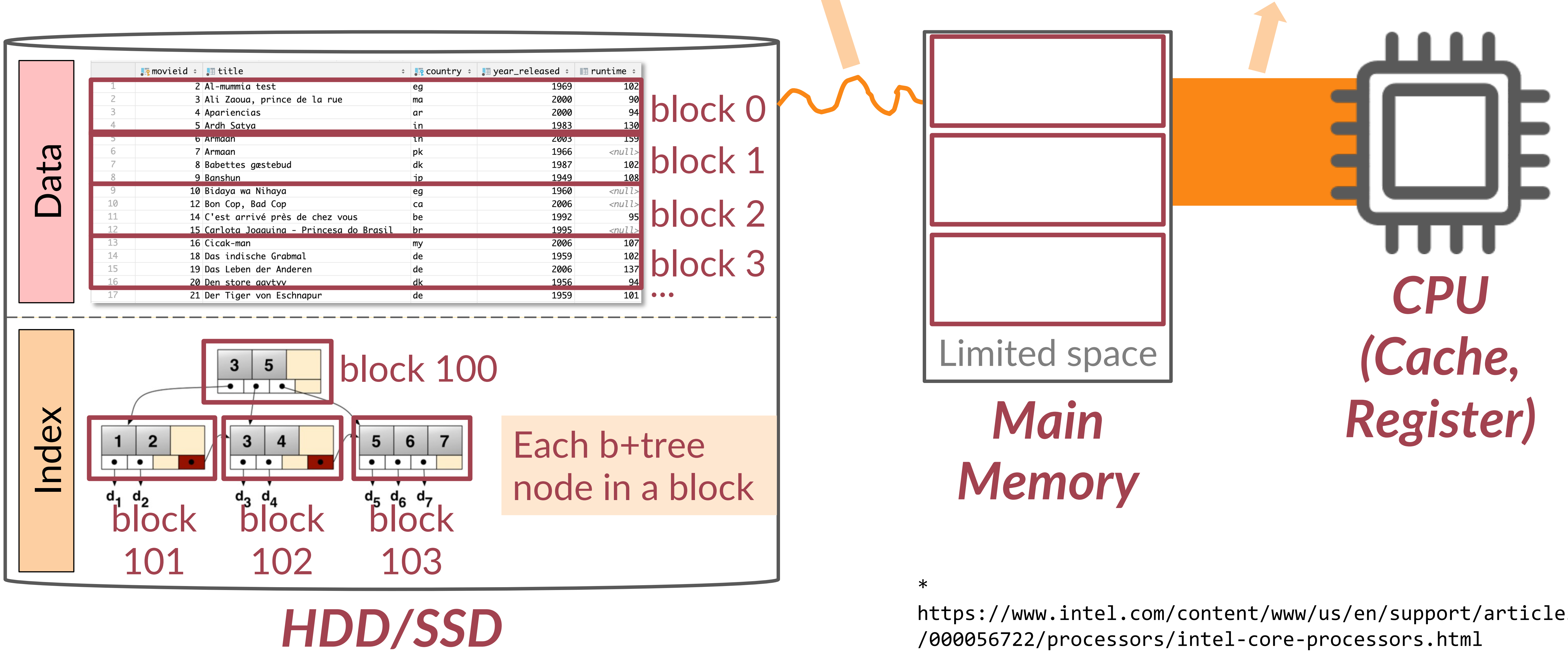


Prerequisite

- Storage model

- Relatively small bandwidth
  - 100MB/s ~ <10GB/s
- High latency
  - Millisecond-level

- Very large bandwidth
  - 94GB/s (for DDR4 2933*)
- Very low latency
  - Nanosecond-level



*
https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html

Prerequisite

- Measuring query cost
  - Disk cost can be estimated as:
    - Number of seeks * average-seek-cost
    - Number of blocks read * average-block-read-cost
    - Number of blocks written * average-block-write-cost
  - For simplicity, we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
    - t_T – time to transfer one block
      - Assuming for simplicity that write cost is same as read cost
    - t_S – time for one seek
  - E.g., cost for b block transfers plus S seeks
    $b * t_T + S * t_S$

Prerequisite

- Measuring query cost
  - t_S and t_T depend on where data is stored. With 4 KB blocks:
    - High end magnetic disk: t_S = 4 msec and t_T =0.1 msec
    - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB
  - Required data may be buffer resident already, avoiding disk I/O
    - But hard to take into account for cost estimation
  - Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
    - But more optimistic estimates are used in practice
  - We ignore CPU costs for simplicity
    - Real systems do take CPU cost into account
    - Network costs must be considered for parallel systems

Overview

- Selection
- Joining

Selection Operation

- Let's start from this simple query:

```
select * from movies where [CONDITION];
```

  - If you are the designer of the database engine, what do you think is the best way to fulfill this requirement?
  - Two factors to consider:
    - What comparison is it in the CONDITION (equality / comparison)?
    - Does the column involved in the CONDITION have an index?

Basic Linear Scan

- Linear Search (displayed as `Seq Scan` in PostgreSQL)
  - Scan each file block and test all records to see whether they satisfy the selection condition
  - Cost estimate = b_r block transfers + 1 seek
    - Assuming blocks of the file are stored contiguously
    - b_r denotes number of blocks containing records from relation r
- Although slower than other algorithms for implementing selection, linear search can be applied regardless of
  - Selection condition
  - Ordering of records in the file
  - Availability of indexes

Basic Linear Scan

- However, a full-table linear scan on extremely-large tables can be a disaster
  - E.g., billions of records in database

  - That’s why we need other optimized ways

Index Scan

- **Index scan** – Search algorithms that use an index
  - Selection condition must be on search-key of index

```sql
select * from movies where movieid = 125;
```

We have a B+ tree index on `movieid`

- Plan: `Index Scan`

```sql
select * from movies where runtime = 100;
```

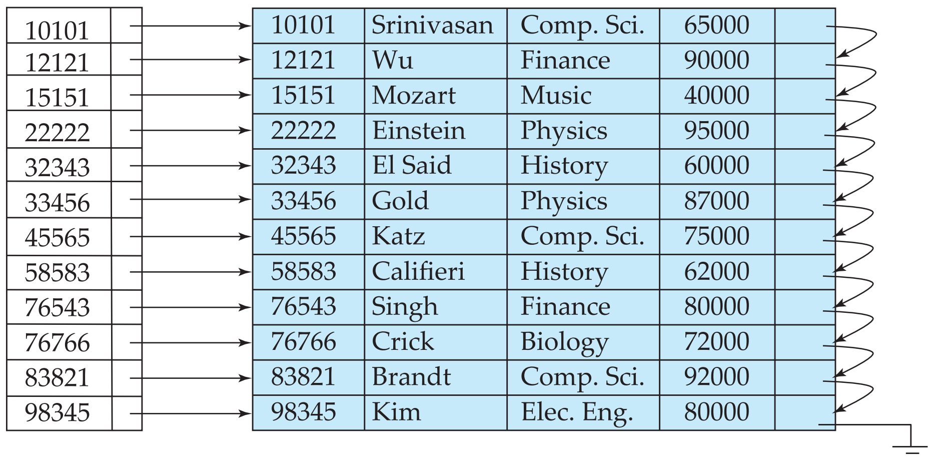
We don't have any index on `runtime`

- Plan: `Seq Scan`

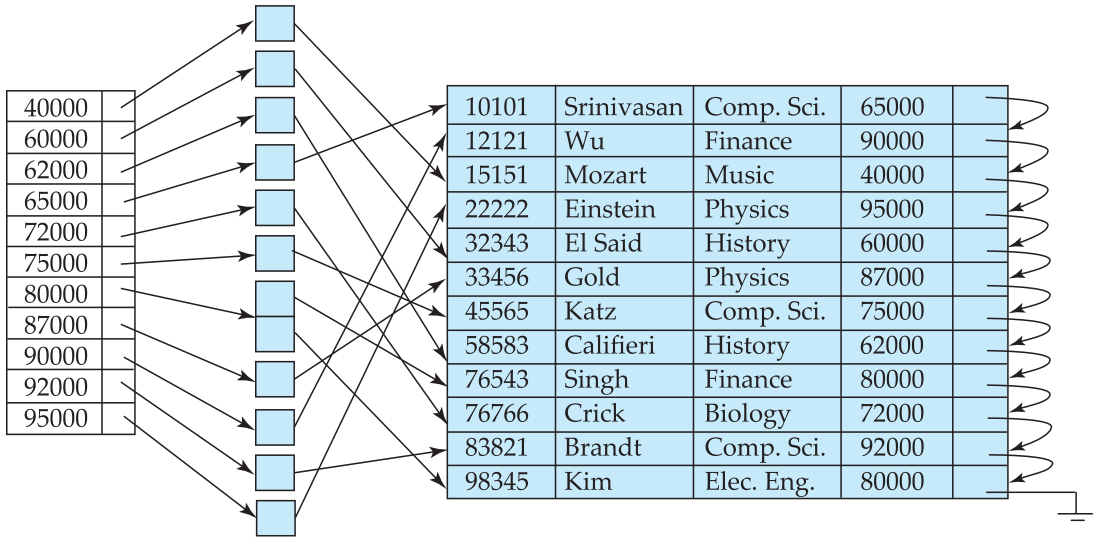
Index Scan

- **Index scan** – Search algorithms that use an index
  - Selection condition must be on search-key of index

- Unlike linear scan, we need to talk about different types of indexes and CONDITIONs
  - Clustered / Non-clustered index (Primary / Secondary index)
  - Equality / Comparison test



Clustered index



Non-clustered index

Index Scan

h_i: height of the B+-tree

Clustered index, equality on key

- Retrieve a single record that satisfies the corresponding equality condition
  - *key => no duplicated values*
  - *Cost* = $(h_i + 1) * (t_T + t_S)$
    - Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.

Clustered index, equality on non-key

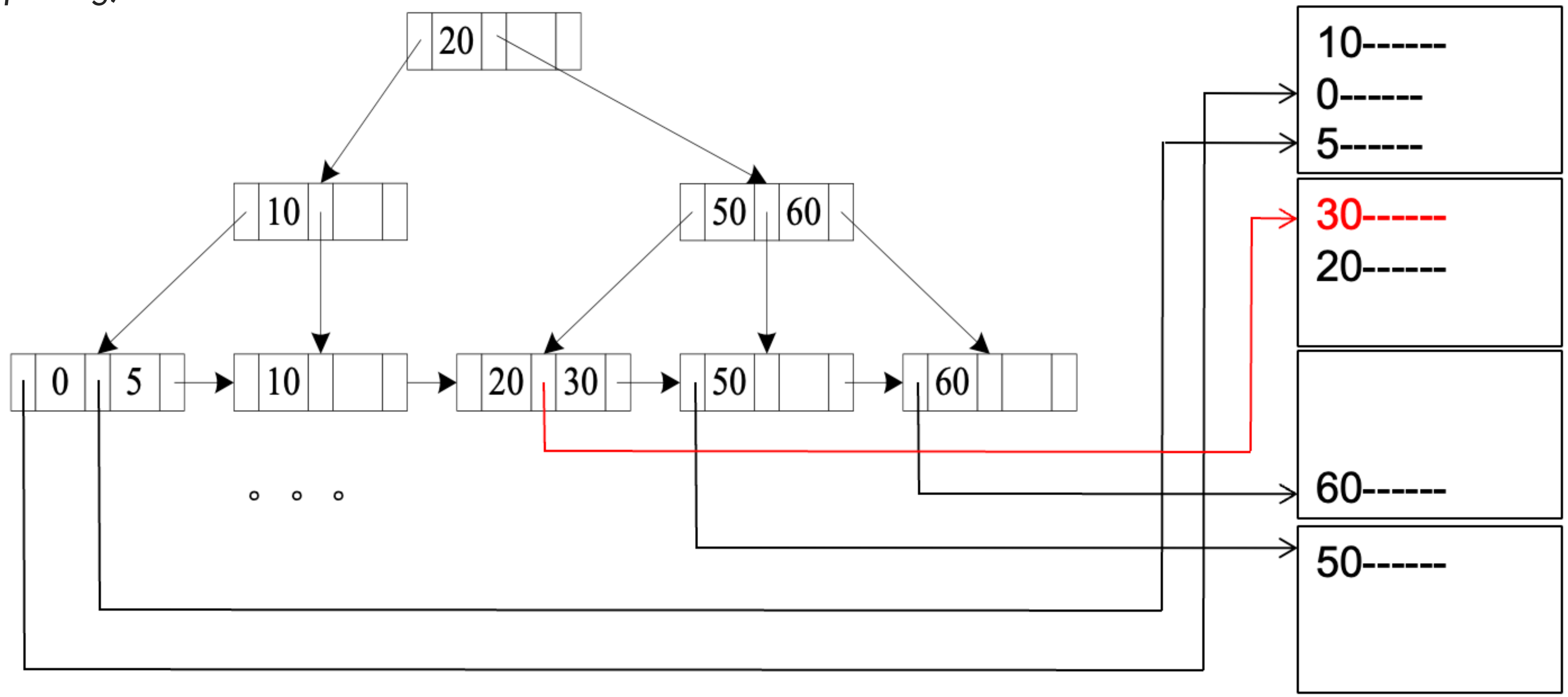
- Retrieve multiple records
  - *non key attributes => possible to have duplicated values*
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
    - One seek for each level of the tree, one seek for the first block
    - Let b = number of blocks containing matching records, which will be on consecutive blocks (since it is a clustering index) and don't require additional seeks

Index Scan

h_i: height of the B+-tree

Secondary index, equality on key/non-key

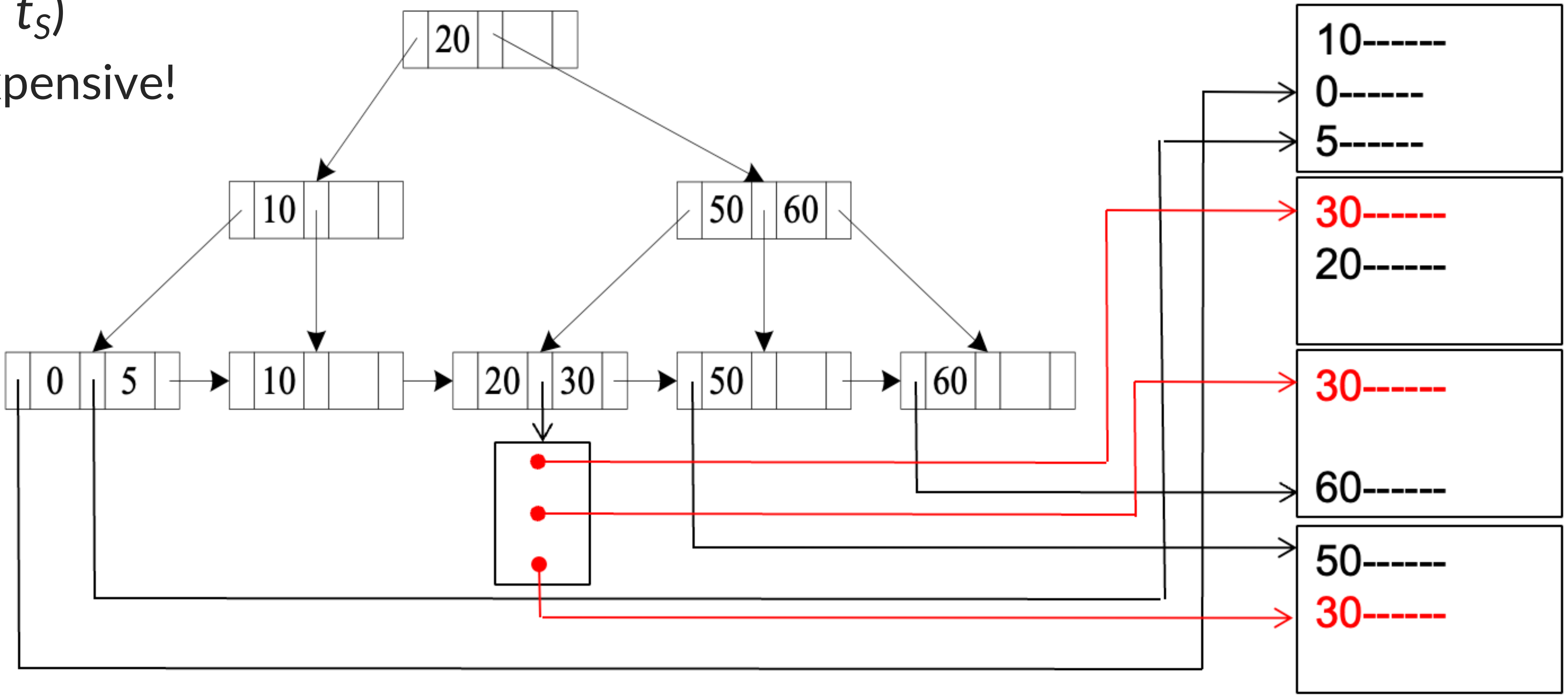
- Retrieve a single record if the search-key is a candidate key
  - $Cost = (h_i + 1) * (t_T + t_S)$



Index Scan

Secondary index, equality on key/non-key

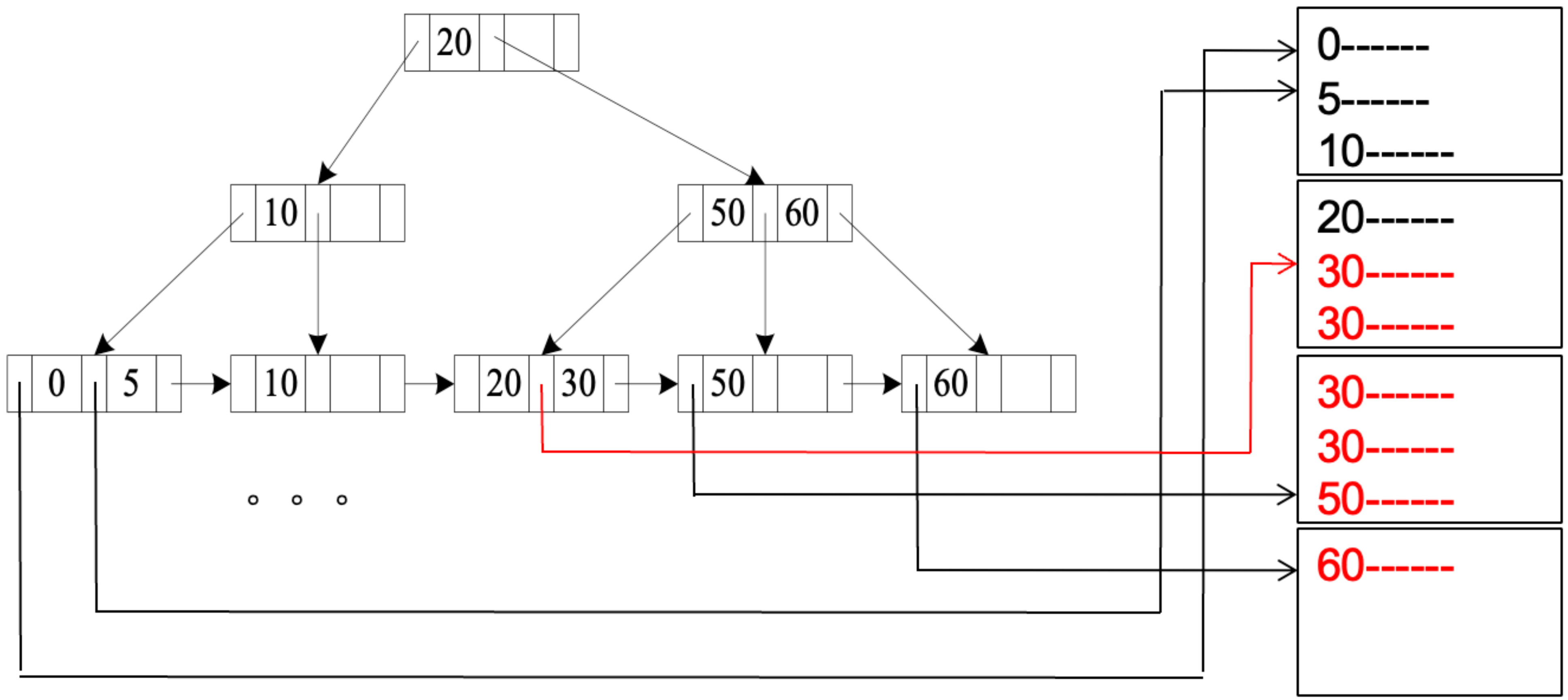
- Retrieve multiple records if search-key is not a candidate key
  - Each of *n* matching records may be on a different block
  - Cost = $(h_i + n) * (t_T + t_S)$
    - Can be very expensive!



Index Scan

Tip: Comparison tests can always be fulfilled with *linear scans*, which is the fallback solution

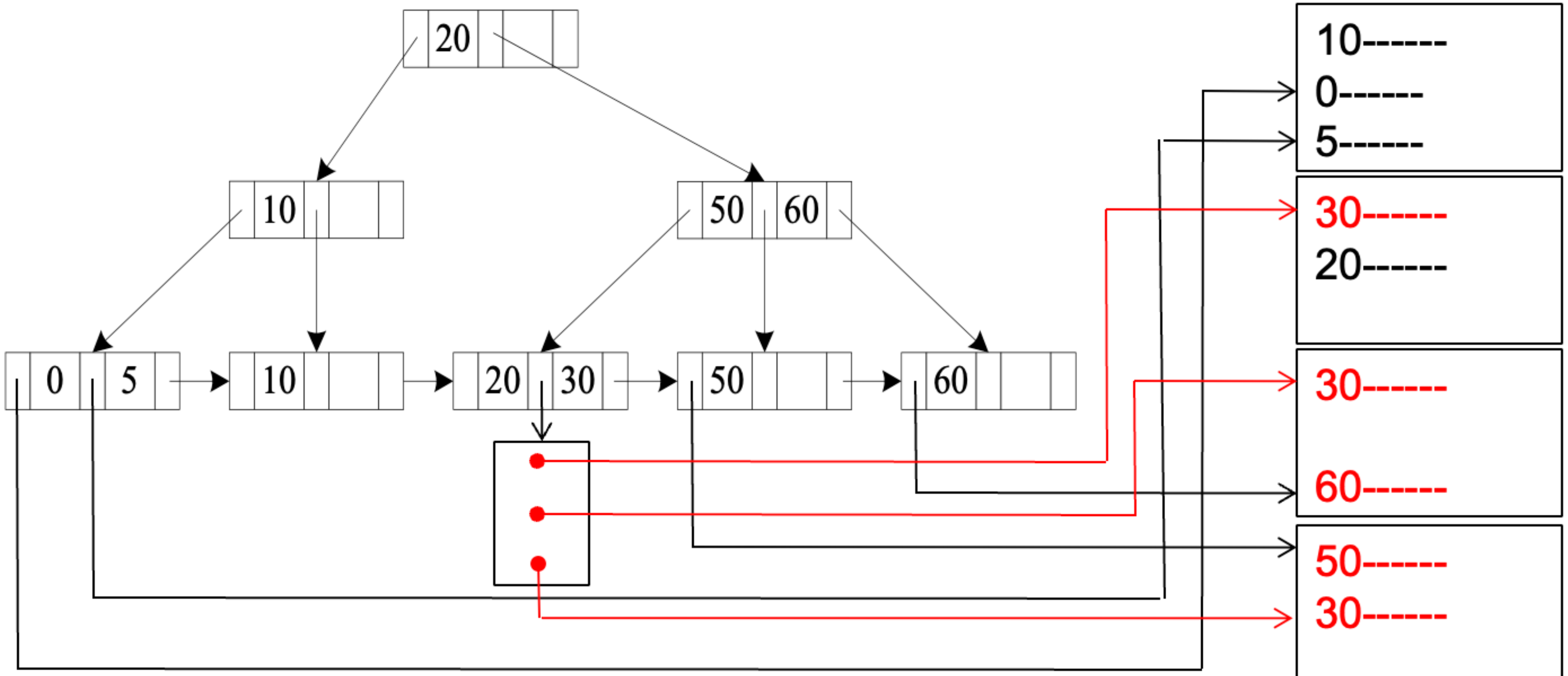
- **Clustered index, comparison** (i.e., Relation is sorted on *A*)
  - For $\sigma_{A \geq V}(r)$, use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$, just scan relation sequentially till first tuple > *v*; do not use index



Index Scan

- **Non-clustered index, comparison**
  - For $\sigma_{A \geq V}(r)$, use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$, just scan leaf pages of index finding pointers to records, till first entry > v

- In either case, retrieving records that are pointed to requires an I/O per record
- Linear scan may be cheaper!



Complex Selections

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

- <u>Conjunctive selection using single-key index(es)</u>
  - Select a θ_i and algorithms mentioned above that results in the least cost for $\sigma_{\theta i}(r)$
  - Test other conditions on tuple after fetching it into memory buffer
- <u>Conjunctive selection using multi-key index</u>
  - Use appropriate composite (multiple-key) index if available
- <u>Conjunctive selection by intersection of identifiers</u>
  - Requires <u>indices</u> with record pointers/identifiers
    - Here, "indices" means the order number of records in the files, like array indices. They are not indexes
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - Then fetch records from file (and test remaining conditions)

Complex Selections

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$

Disjunctive selection by union of identifiers

(Similar to the third way on the previous page)

- Applicable if *all* conditions have available indexes
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers
    - Then fetch records from file
- Otherwise, just use linear scan
  - The disjunctive condition tested on each tuple during the scan

How `join` Works (with the help of indexes)

- Some widely-used `join` algorithms
  - Nested-loop join
  - Indexed nested-loop join
  - Merge join

Nested-loop Join

- To compute the *theta* join $r \bowtie_{\theta} s$

for each tuple t_r **in** r **do begin**
    for each tuple t_s **in** s **do begin**
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add $t_r \bullet t_s$ to the result.
    end
end

- r is called the **outer relation** and s the **inner relation** of the join
  - Think about the "outer loop" and the "inner loop" in programming

- Requires no indices and can be used with any kind of join condition
  - <u>Expensive</u> since it examines every pair of tuples in the two relations

Nested-loop Join via File Scan

- In the worst case, if the memory can only hold one block of each relation, the estimated cost is:
  - $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks
    - n_r - number of records in relation r
    - b_s, b_r - number of blocks in relation s and r
    - $n_r * b_s + b_r$: for each record in r, need to read all blocks in s; and need to read all b_r blocks in r
    - $n_r + b_r$: for each record in r, need to do seek once for s; and need to seek r for b_r times
      - Assuming r and s are stored contiguously on disks

- If the smaller relation fits entirely in memory, use that as the inner relation
  - Reduces cost to $b_r + b_s$ block transfers and 1 + 1 seeks

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join ($r \bowtie_{r.A=s.B} s$) or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join

- For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r
  - Essentially a selection operation given the values of the joining attribute in t_r

Indexed Nested-Loop Join

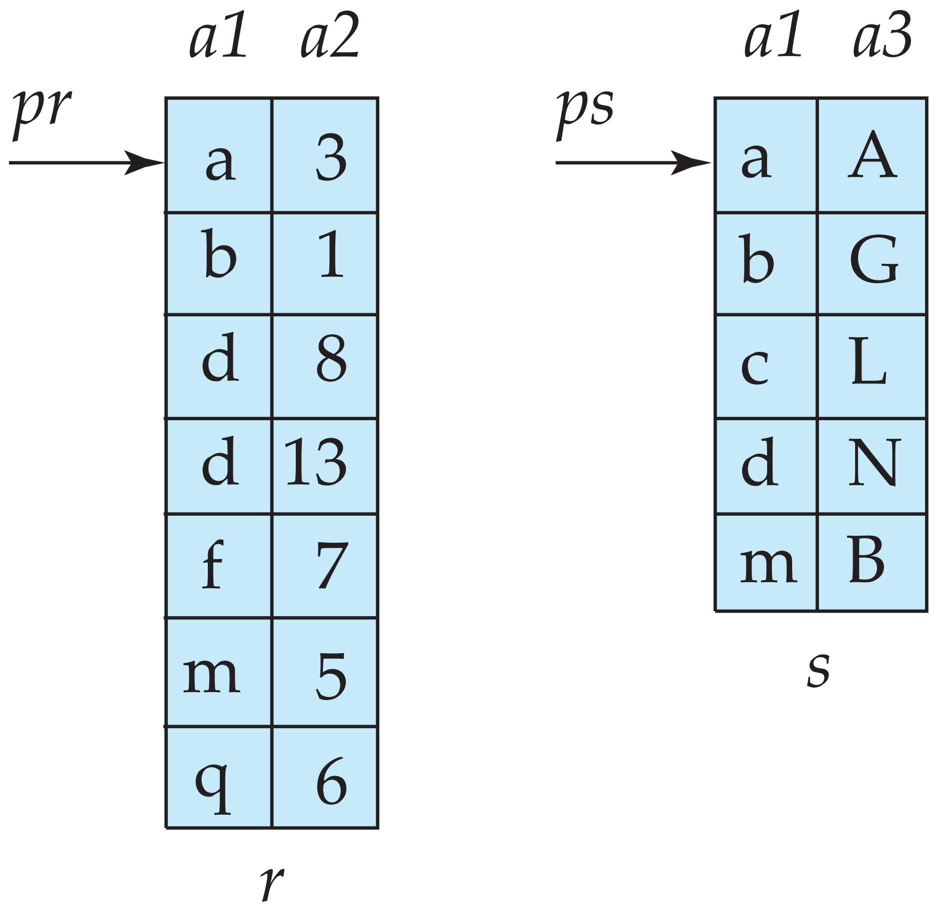
- Worst case: Buffer in memory has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s*

- Cost of the join: $b_r * (t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple of *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition
  - We need b_r seeks as the disk head may have moved between each I/O

- If indices are available on join attributes of both *r* and *s*, use the relation with fewer tuples as the outer relation

Merge Join

- a.k.a., sort-merge join
  - Zipper-like joining
- Steps
  - Sort both relations on their join attribute (if not already sorted on the join attributes)
  - Merge the sorted relations to join them
    - if r.a1[pr] < s.a1[ps], pr++
    - elif r.a1[pr] > s.a1[ps], ps++
    - else, join and move pr and ps
- Join step is similar to the merge stage of the sort-merge algorithm
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched



Merge Join

- Can be used only for equi-joins ($r \bowtie_{r.A=s.B} s$) and natural joins
- Once the relations are in sorted order, each block needs to be read only once
  - Assuming all tuples for any given value of the join attributes fit in memory
- The cost of merge join is:

$$b_r + b_s \text{ block transfers } + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ seeks}$$

+ the cost of sorting if relations are unsorted

b_b: memory buffer size, counted in number of blocks, for each relation

Hash Join

- Hash join
  - Build a set of buckets for a smaller table to speed up the data lookup

- Procedure:
  - 1. Create a hash table for the smaller table t1 in the memory
  - 2. Scan the larger table t2. For each record r,
    - 2.1 Compute the hash value of r.`join_attribute`
    - 2.2 Map to corresponding rows in t1 using the hash table