Principles of Database Systems (CS307)

Lecture 12: Indexing

Yuxin Ma

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.

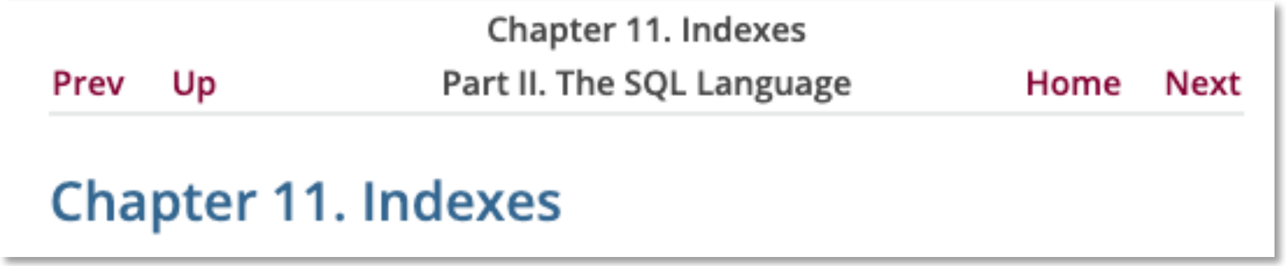
Intro

Motivation

- 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 what were the "***coordinates***" of a book.

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



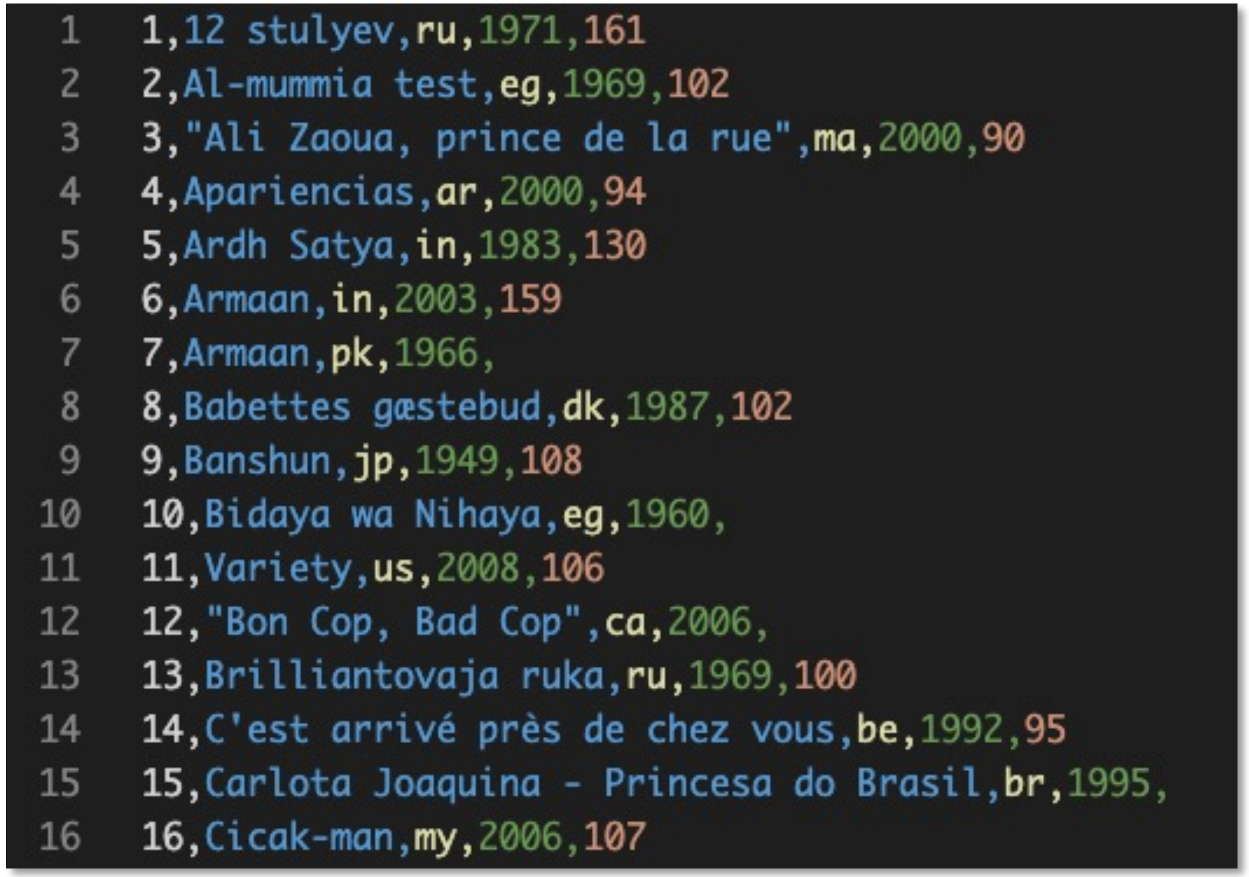
Chapter 11. Indexes

Chapter 11. Indexes

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 search key

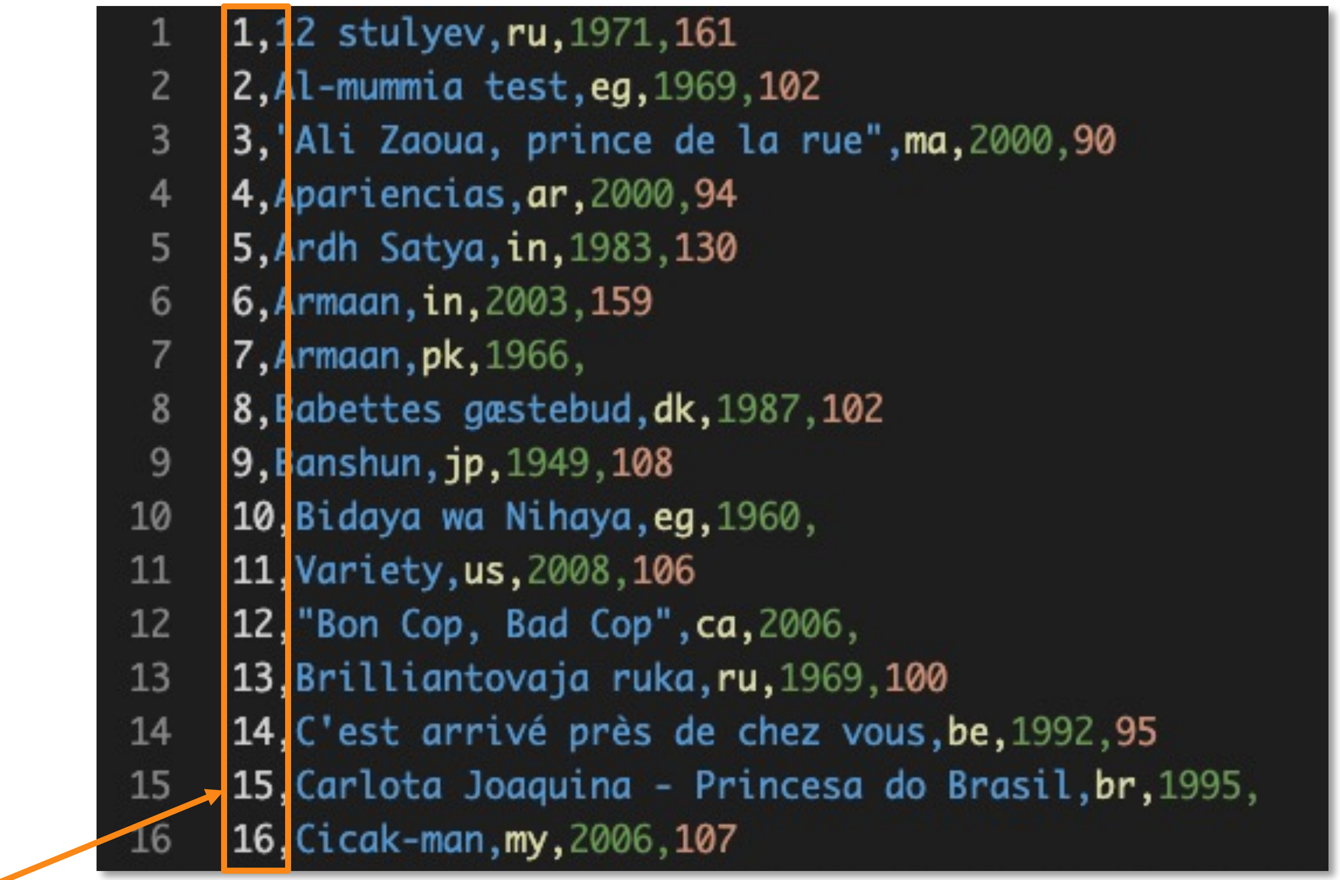


```
1,12 stulyev,ru,1971,161
2,Al-mummia test,eg,1969,102
3,"Ali Zaoua, prince de la rue",ma,2000,90
4,Apariencias,ar,2000,94
5,Ardh Satya,in,1983,130
6,Armaan,in,2003,159
7,Armaan,pk,1966,
8,Babettes gæstebud,dk,1987,102
9,Banshun,jp,1949,108
10,Bidaya wa Nihaya,eg,1960,
11,Variety,us,2008,106
12,"Bon 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
```

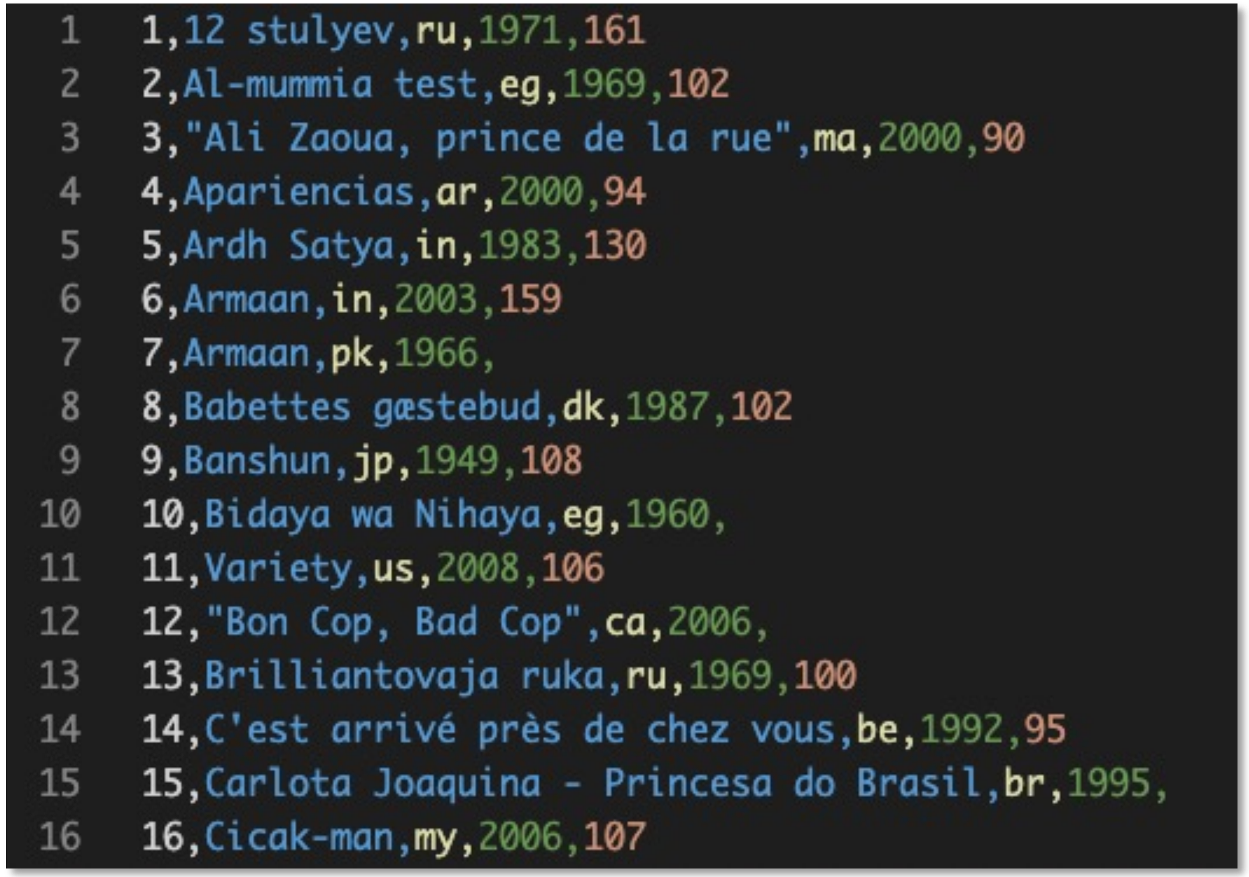
In the current storage structure, the records are sorted by `movieid`

- 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
```

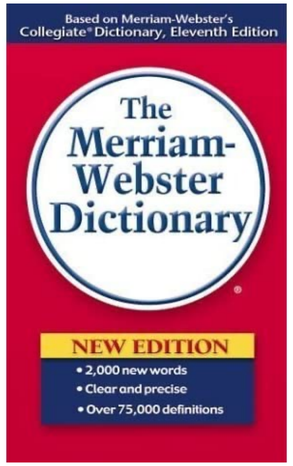
country

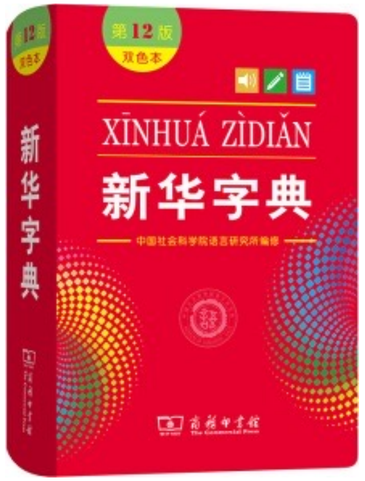
Find the rows `where country = 'cn'`

- 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

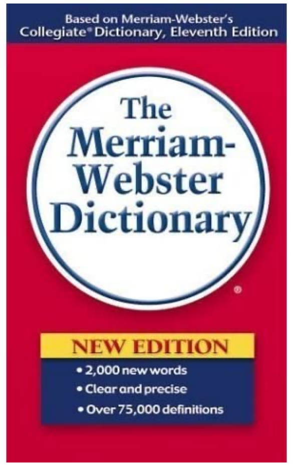

Based on Merriam-Webster's Collegiate® Dictionary, Eleventh Edition
The Merriam-Webster Dictionary
NEW EDITION
• 2,000 new words
• Clear and precise
• Over 75,000 definitions

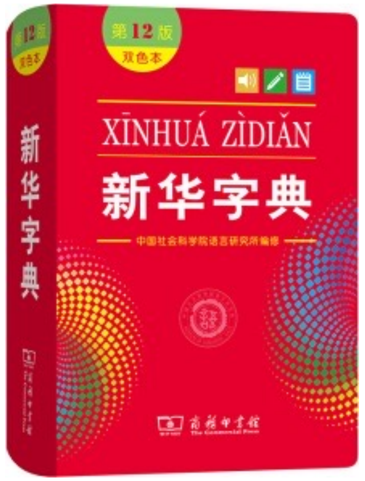


Searching for Record

- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are 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)


Based on Merriam-Webster's Collegiate Dictionary, Eleventh Edition
The Merriam-Webster Dictionary
NEW EDITION
• 2,000 new words
• Clear and precise
• Over 75,000 definitions



Practical Use

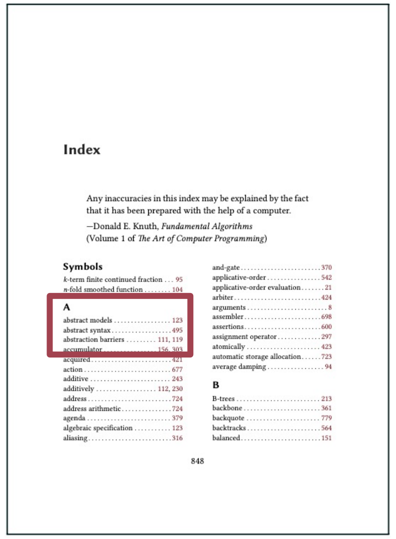
Index in Databases

- 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 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)
  - 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*)

Symbols

k-term finite continued fraction . . . 95
n-fold smoothed function . . . 104

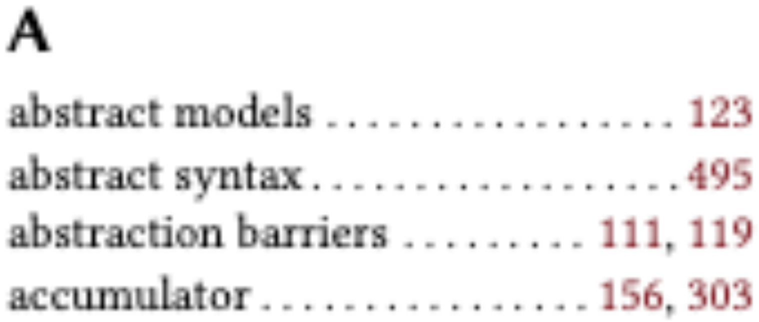
A

abstract models . . . 123
abstract syntax . . . 495
abstraction barriers . . . 111, 119
accumulator . . . 156, 303
acquired . . . 421
action . . . 677
additive . . . 243
additively . . . 112, 230
address . . . 724
address arithmetic . . . 724
agenda . . . 379
algebraic specification . . . 123
aliasing . . . 316
and-gate . . . 370
applicative-order . . . 542
applicative-order evaluation . . . 21
arbiter . . . 424
arguments . . . 8
assembler . . . 698
assertions . . . 600
assignment operator . . . 297
atomically . . . 423
automatic storage allocation . . . 723
average damping . . . 94

B

B-trees . . . 213
backbone . . . 361
backquote . . . 779
backtracks . . . 564
balanced . . . 151

848

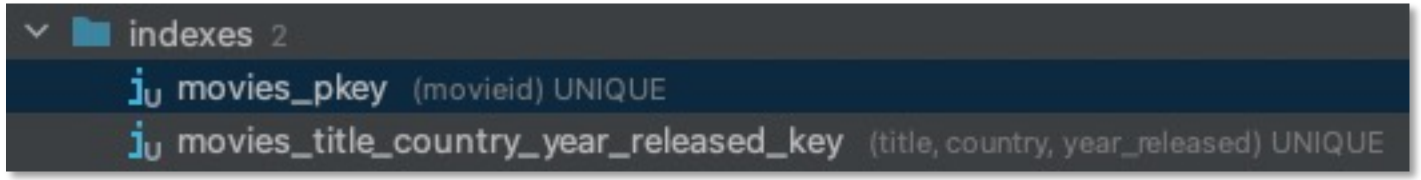


A

abstract models . . . 123
abstract syntax . . . 495
abstraction barriers . . . 111, 119
accumulator . . . 156, 303

Index in Databases

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

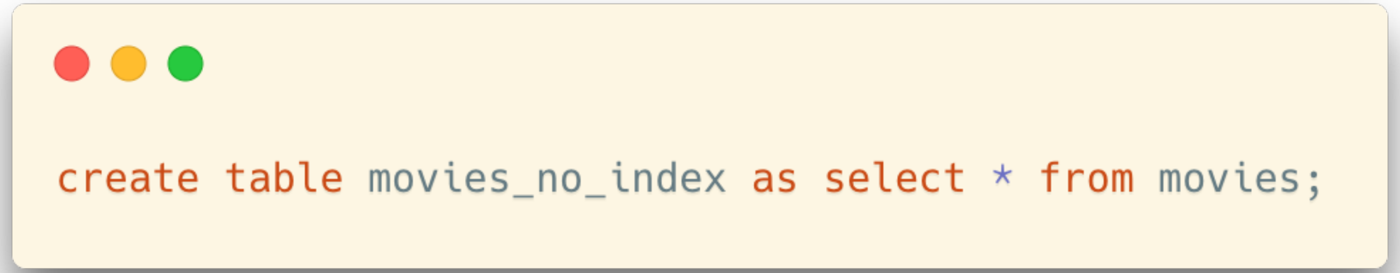


indexes 2
movies_pkey (movieid) UNIQUE
movies_title_country_year_released_key (title, country, year_released) UNIQUE

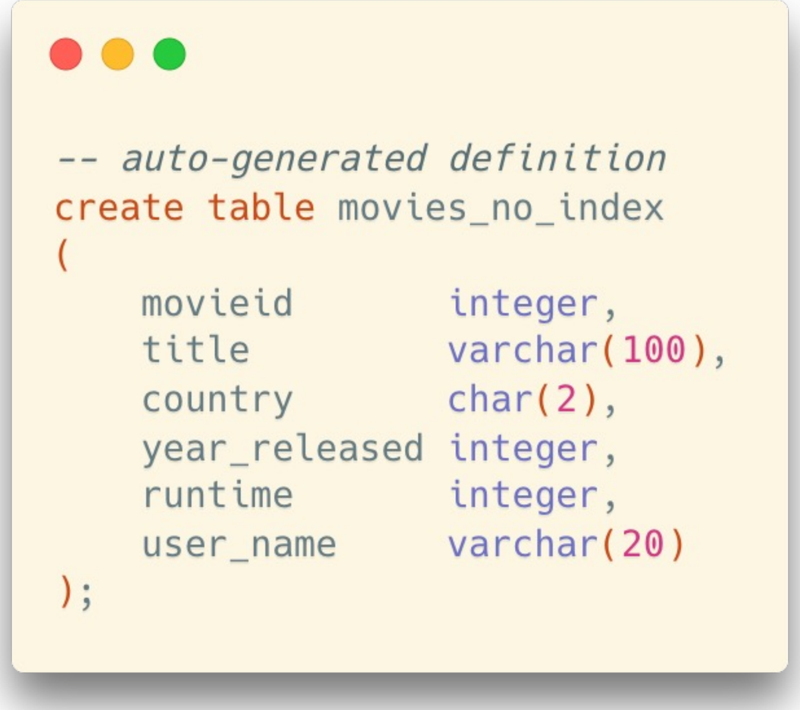
  - In PostgreSQL, indexes are built automatically on columns with `primary key` or `unique` constraints

Experiment on Using Indexes

- Duplicate a table with no index



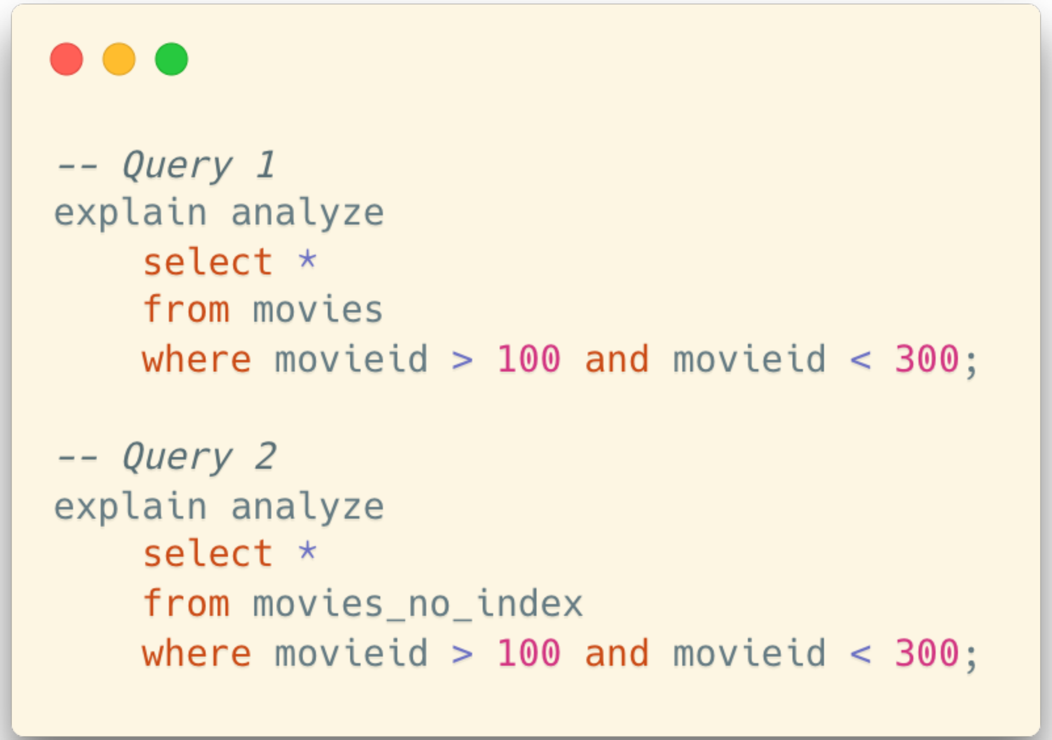
```
create table movies_no_index as select * from movies;
```



```
-- 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)
);
```

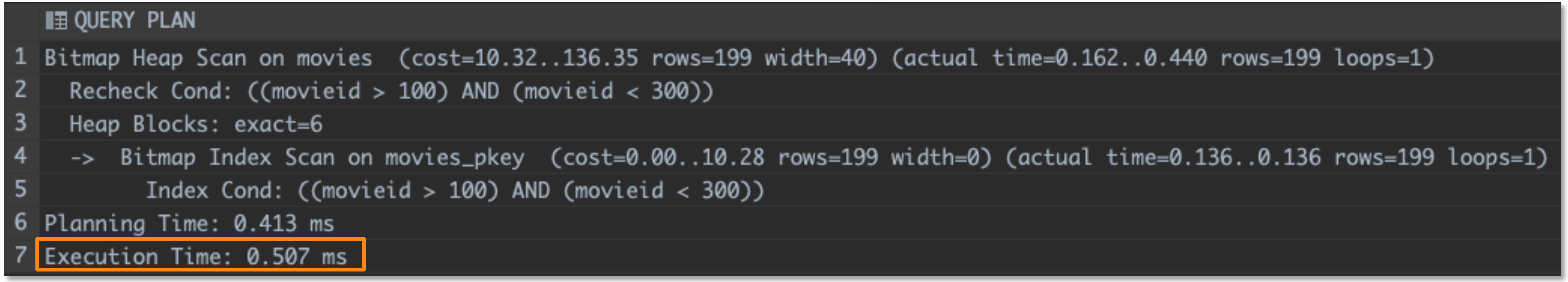
Experiment on Using Indexes

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



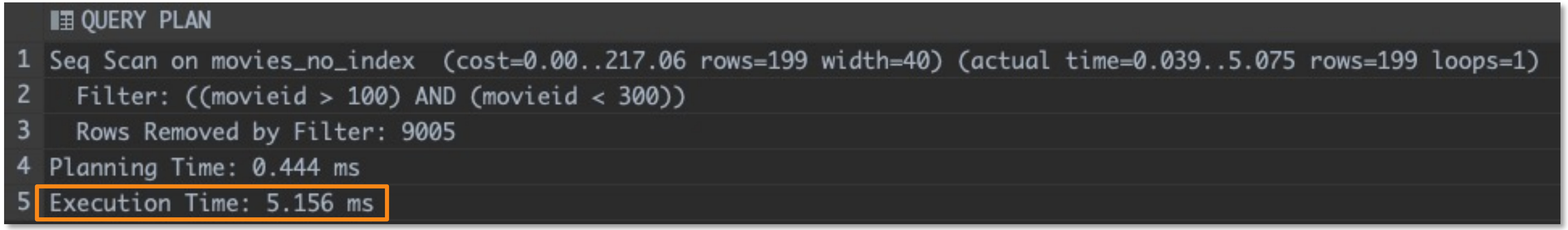
```
-- Query 1
explain analyze
    select *
    from movies
    where movieid > 100 and movieid < 300;

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

Query 1
(on movies)



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

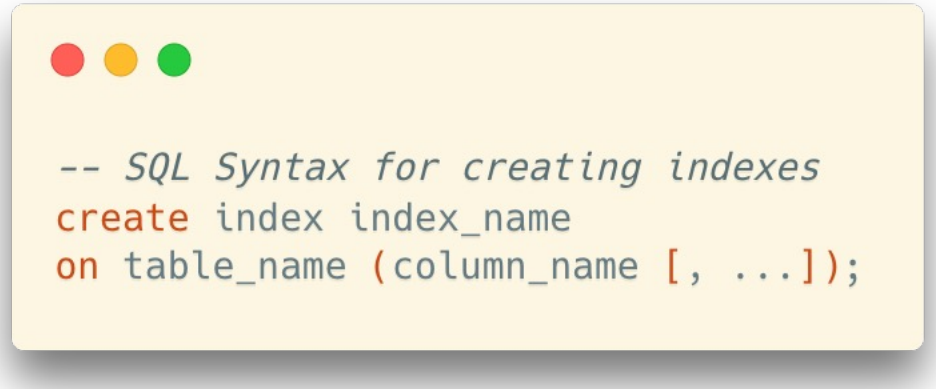
Query 2
(on movies_no_index)



```
QUERY PLAN
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)
  Filter: ((movieid > 100) AND (movieid < 300))
  Rows Removed by Filter: 9005
Planning Time: 0.444 ms
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

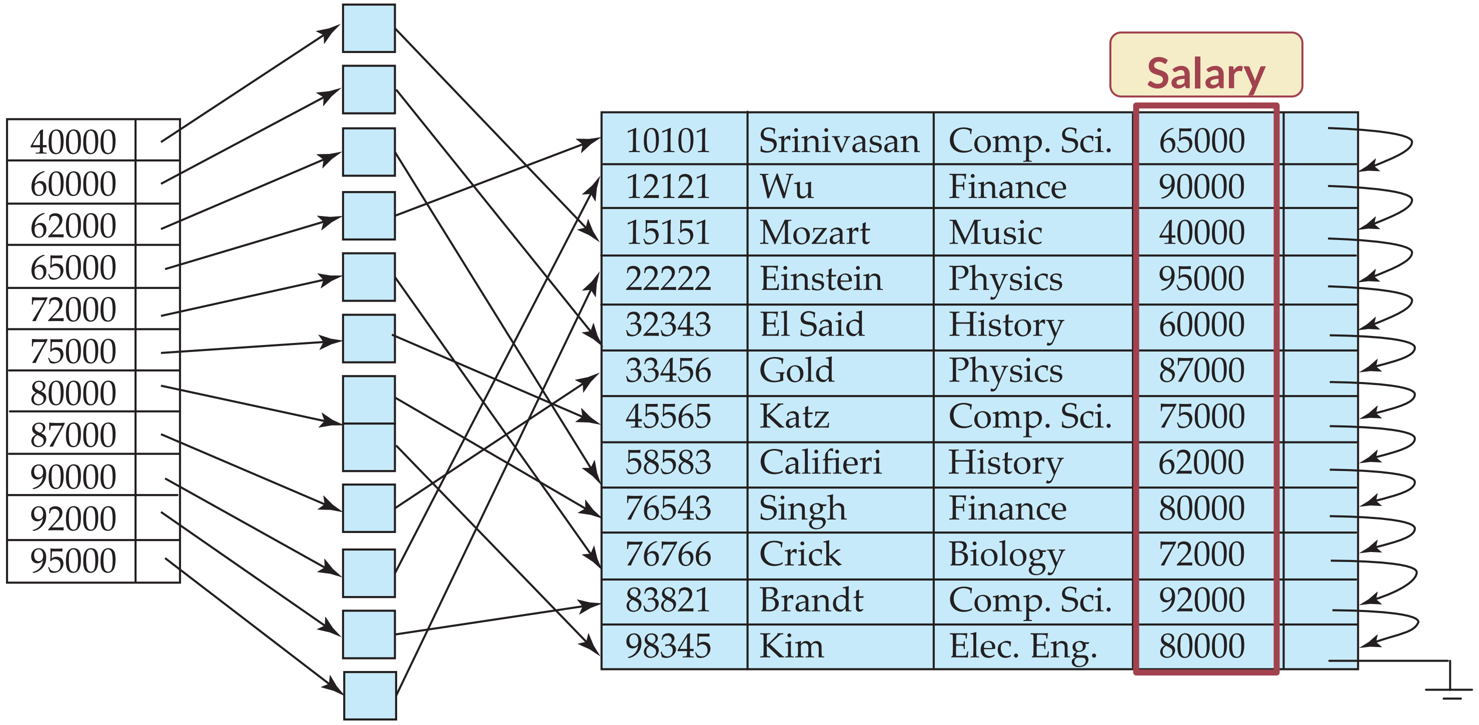
- 1) In terms of storage structure, is the index completely separated with the data records?
    - No ⇒ **Integrated index**
        - PK index in a MySQL InnoDB database
        - PK index in a SQL Server database
    - Yes ⇒ **External index**
        - Indexes in a PostgreSQL database
        - Indexes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



40000
60000
62000
65000
72000
75000
80000
87000
90000
92000
95000

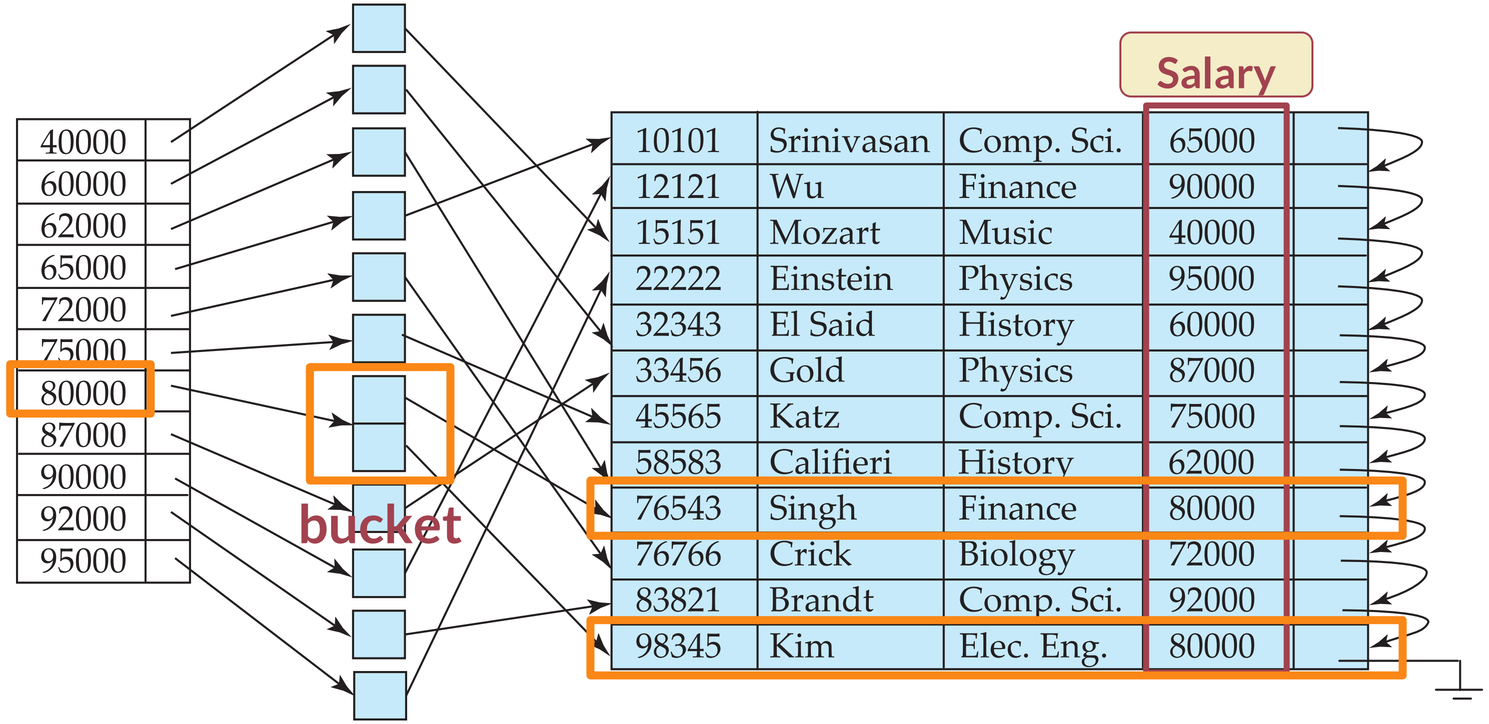
			Salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

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)
  - 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

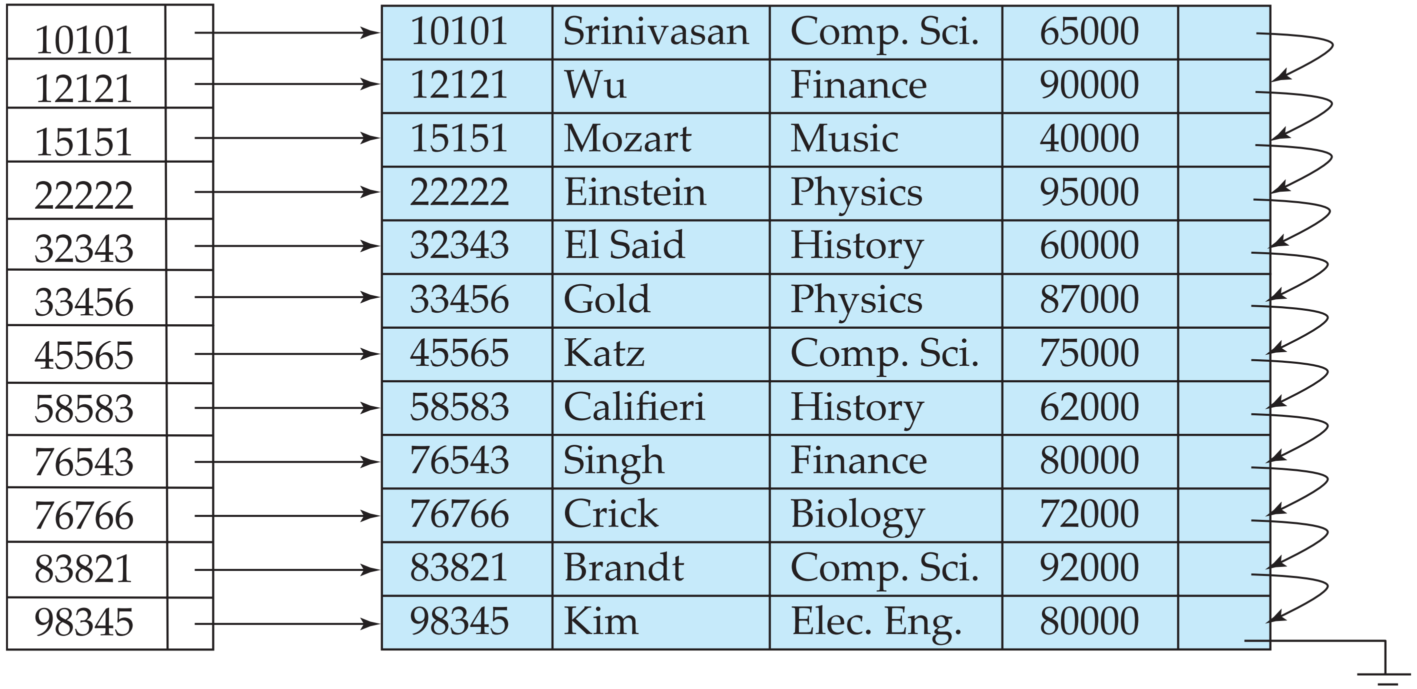


40000
60000
62000
65000
72000
75000
80000
87000
90000
92000
95000

			Salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Index Taxonomy

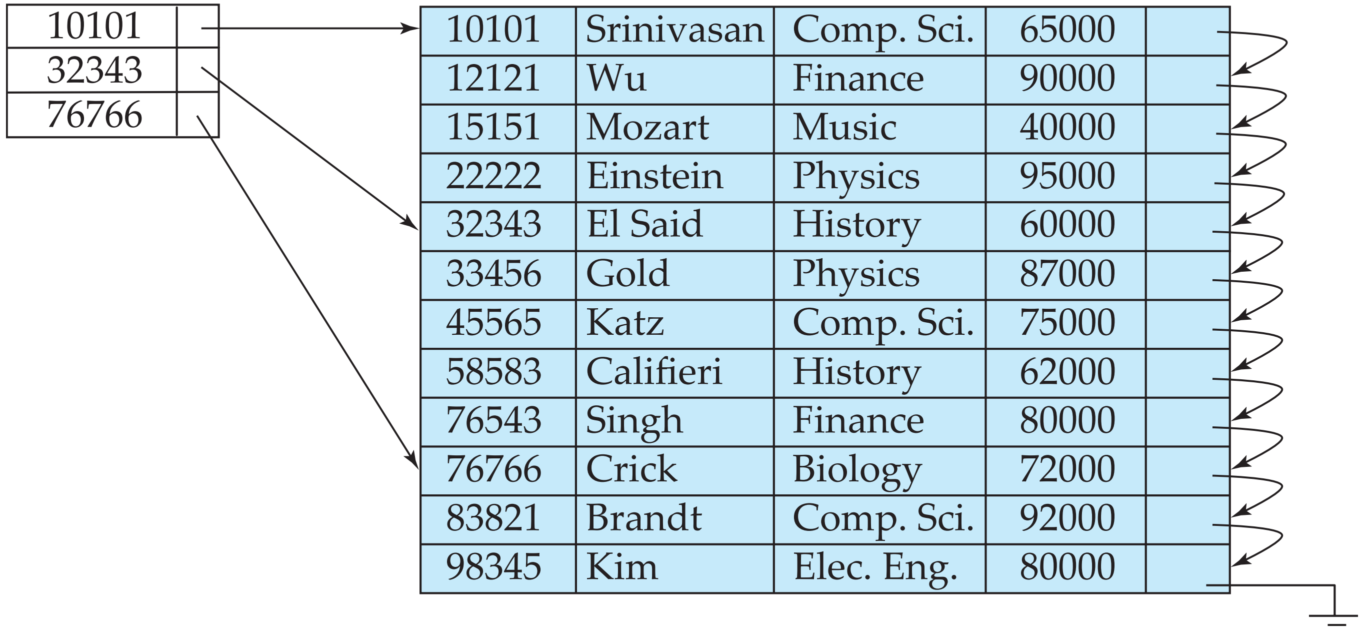
- 3) Does every search key in the data file correspond to an index entry?
  - Yes ⇒ **Dense Index**
  - No ⇒ **Sparse Index**



10101
12121
15151
22222
32343
33456
45565
58583
76543
76766
83821
98345

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Dense Index



10101
32343
76766

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

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

- 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 $k_{\text{sub } i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**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

B-tree

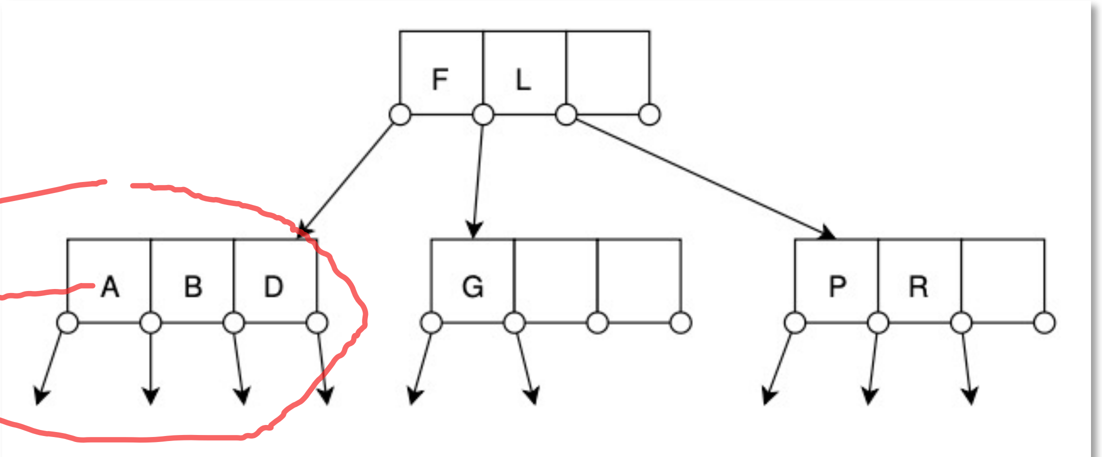


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

- 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 $k_{\text{sub } i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**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

B-tree

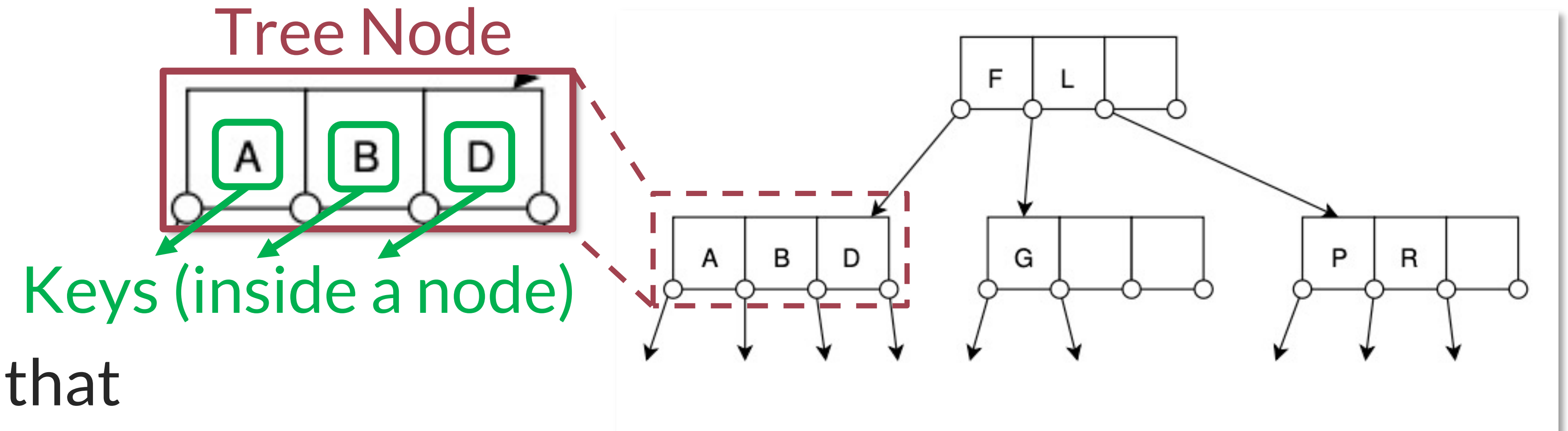


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

- 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 $k_{\text{sub } i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**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

B-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 $k_{\text{sub } i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**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

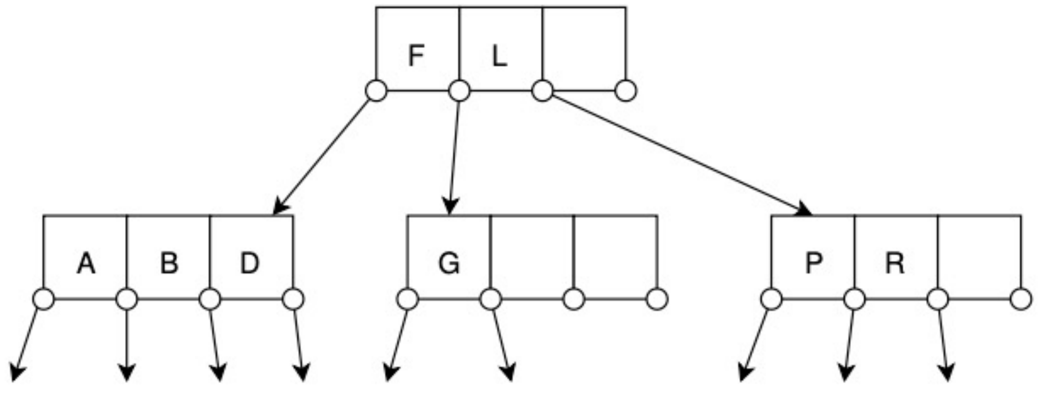


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

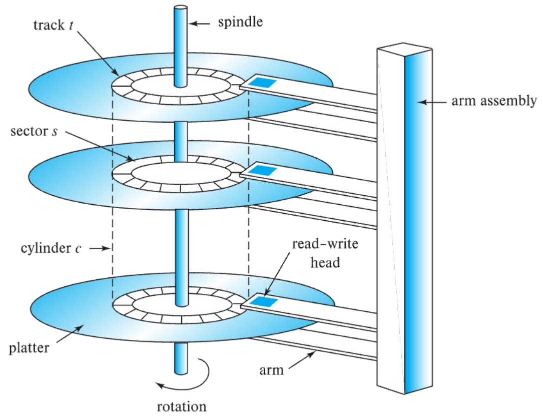
- [m/2] is the called the minimum branching factor (a.k.a. minimum degree) of the tree
- A B-tree of order m is usually called a "[m/2]-m tree", like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, . . .
  - In practice, the order m is much larger (~100)

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

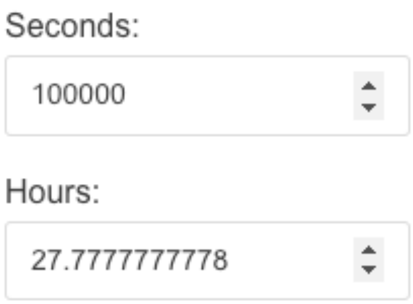
- Height of a *B*-tree: $h \leq 1 + \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right)$
  - If we take an 50-100 tree with 1M records:
    - $h \leq 1+\log_{100/2}(1000000/2) = 4.354$ (i.e., 4 levels)
  - Why do we use B-trees?
    - We can set the size of a B-tree node as the disk page size
      -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

$1\text{ns} = 10^{-6}\text{ms}$

Access time: 50-70ns



Seconds:

100000

Hours:

27.7777777778

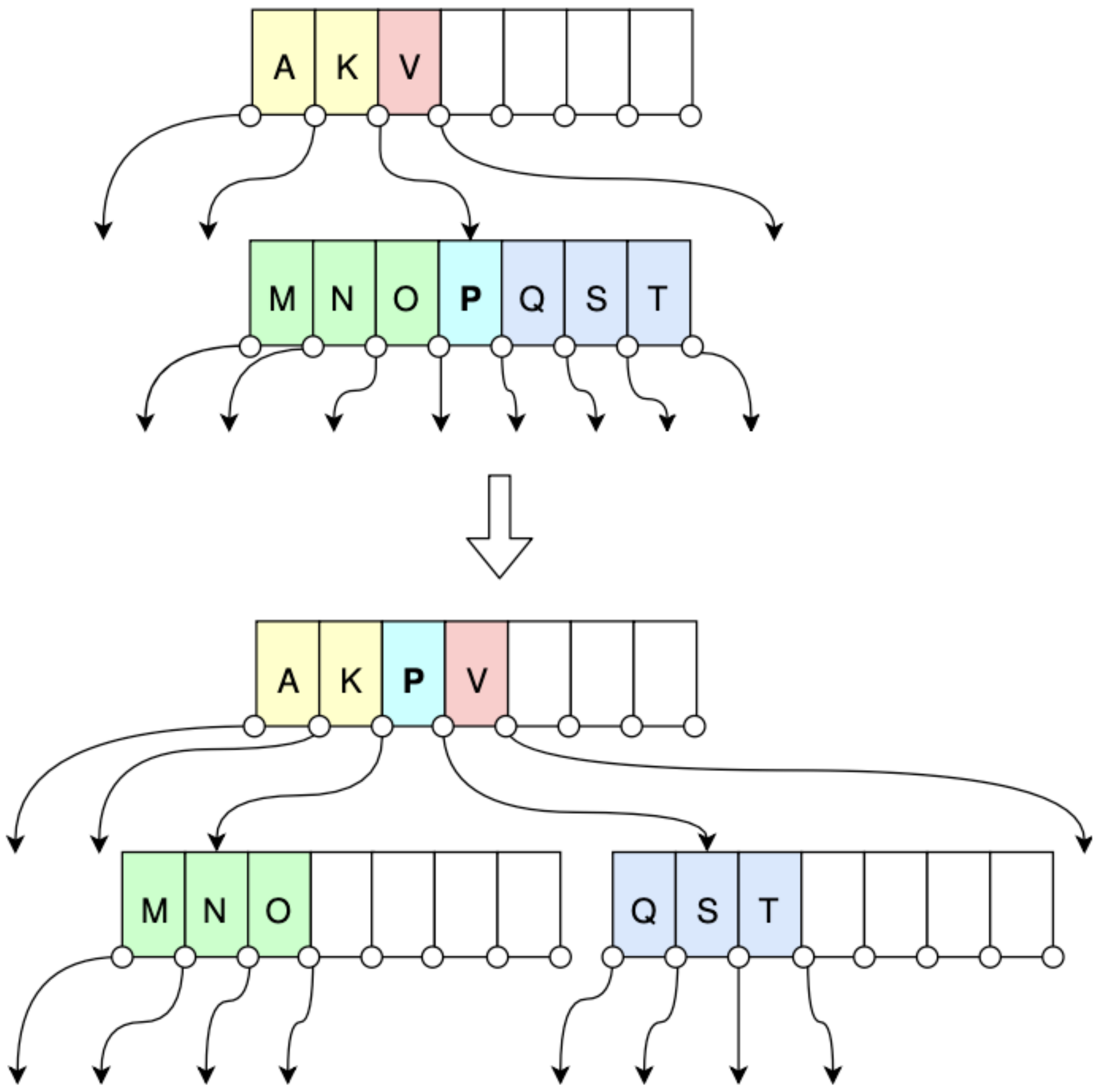
B-tree

- 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"

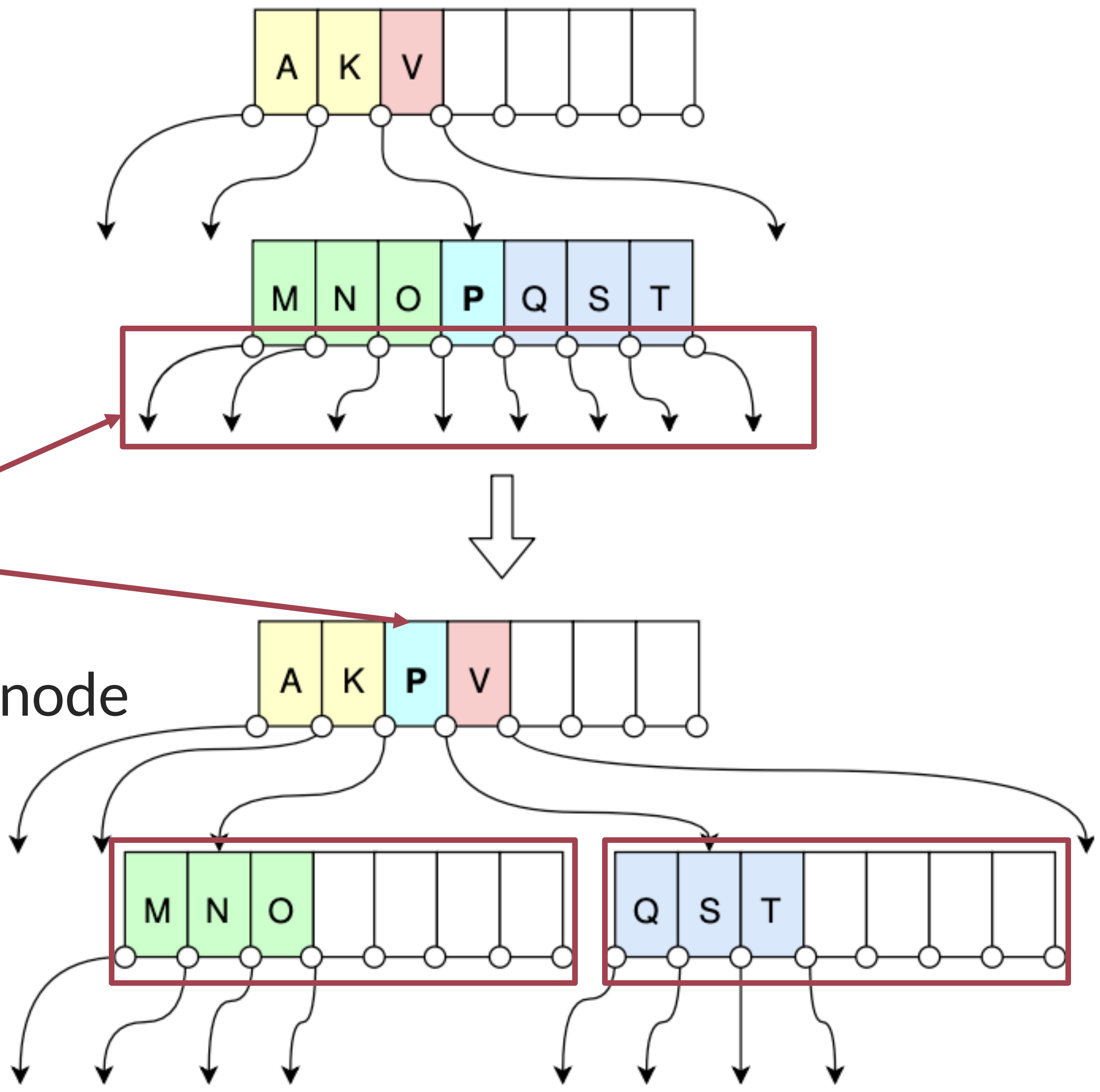


B-tree

- 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 (7)

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

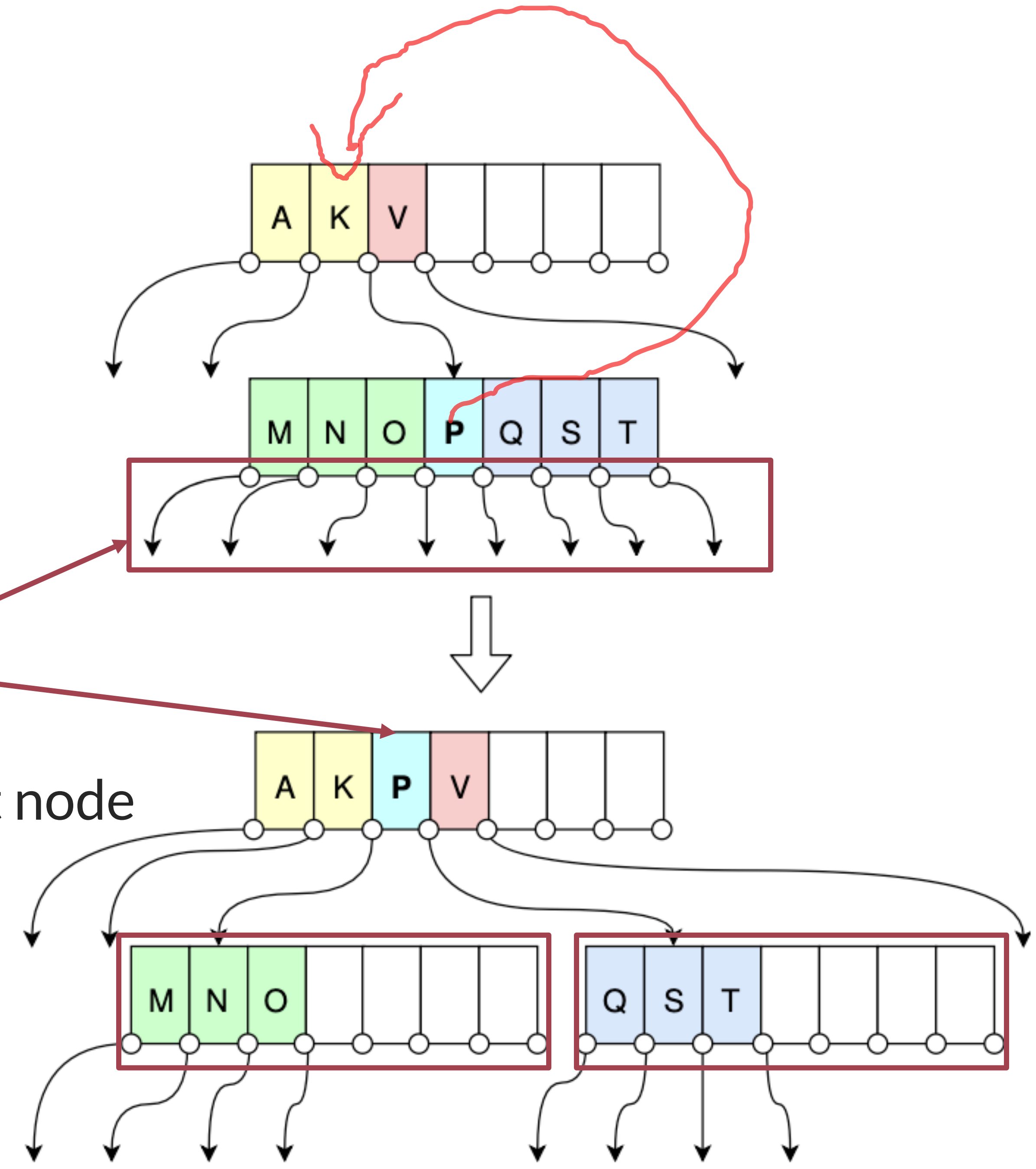


B-tree

- 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 (7)

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

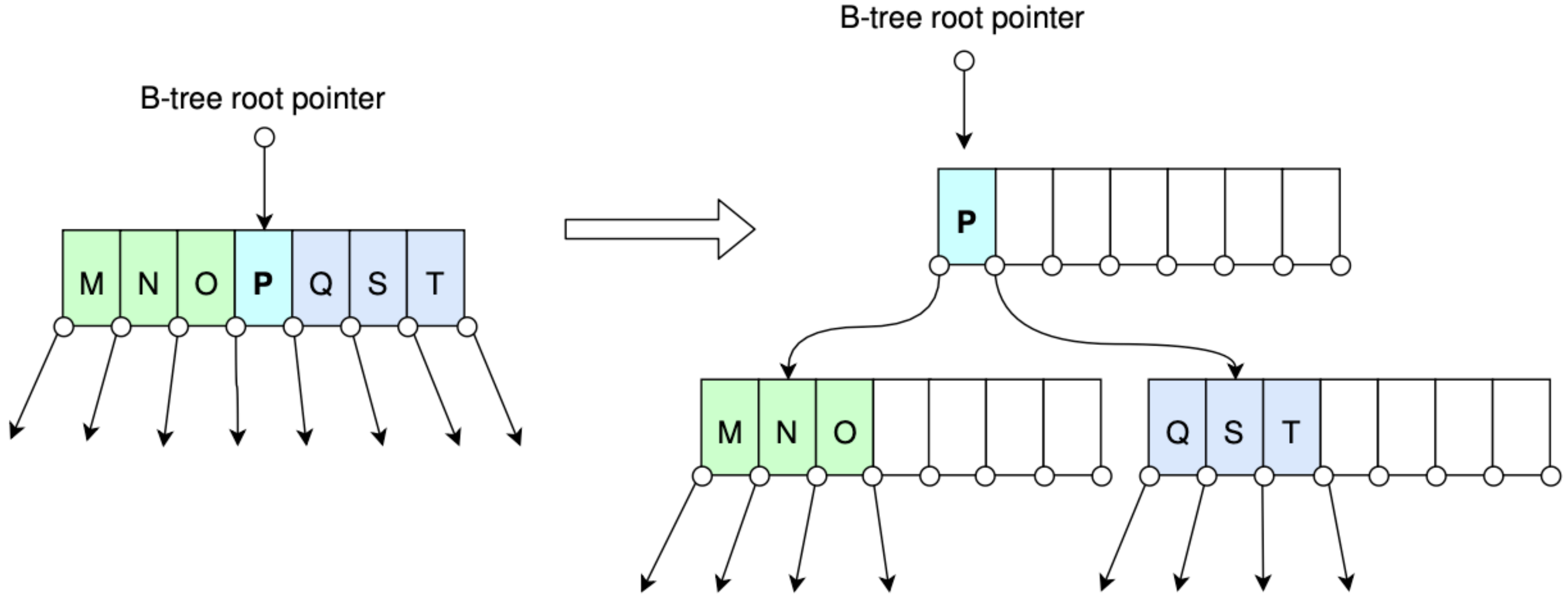


- What if the parent (or even the root) node is also full?

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

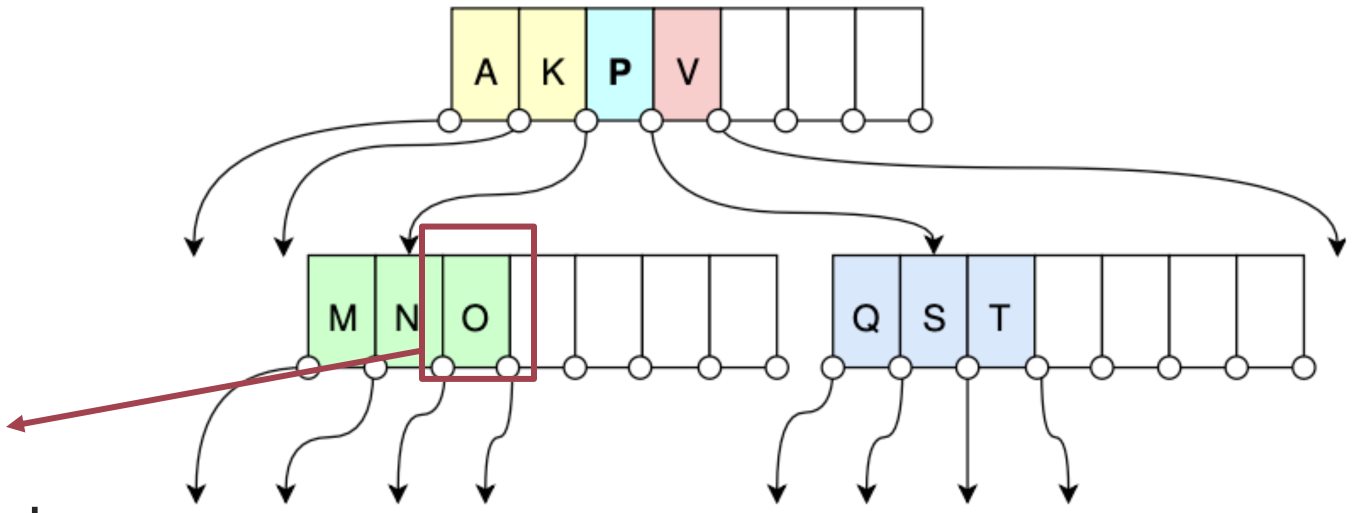


Note that the height of the B-tree is increased by 1

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

B+-tree

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

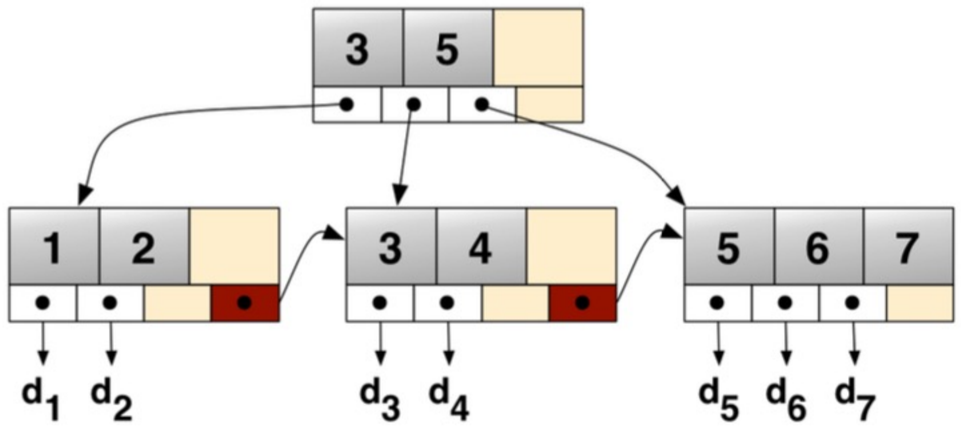


For example, we have accessed the node for letter "O"

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

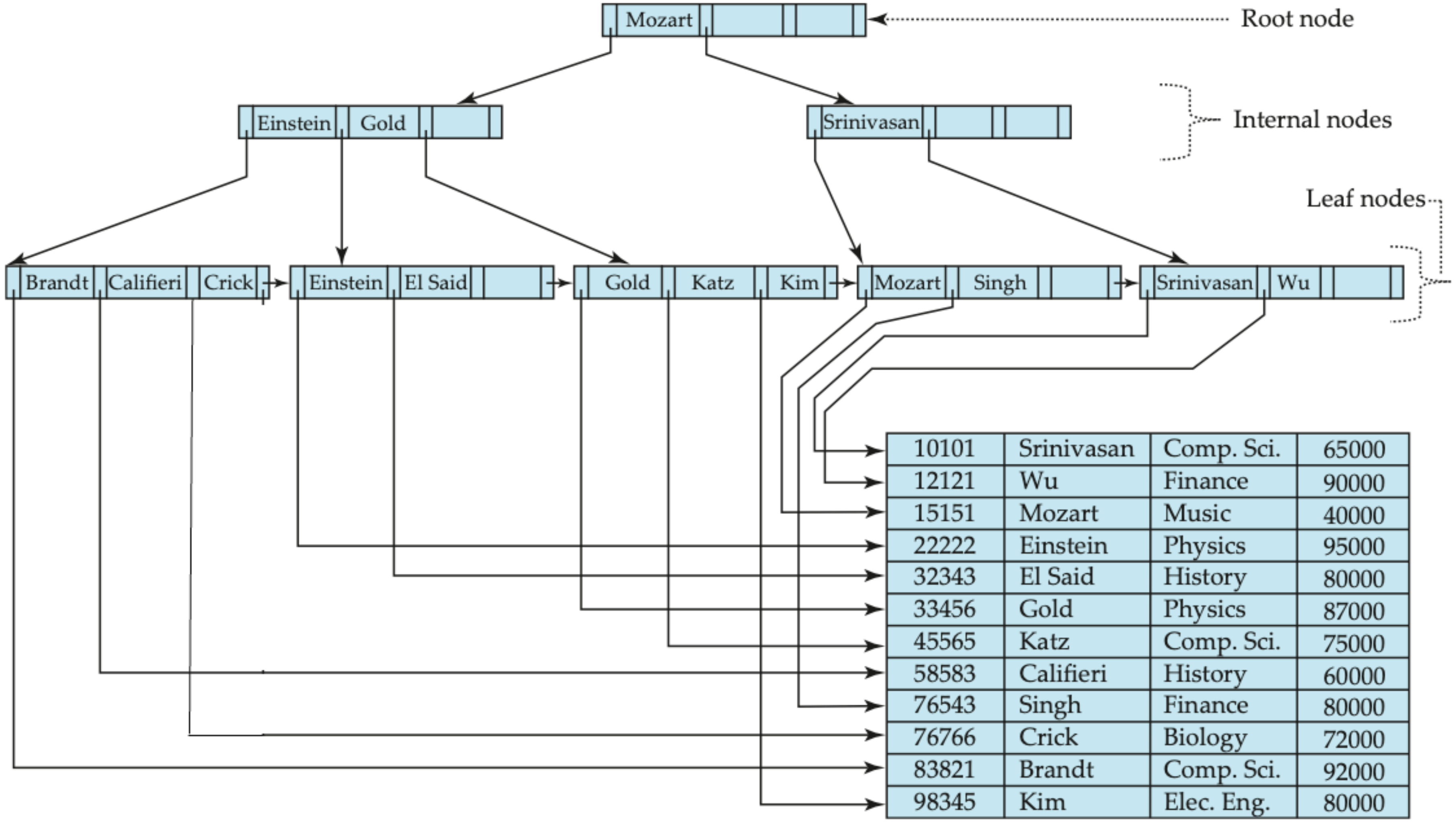
B+-tree

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



B+-tree

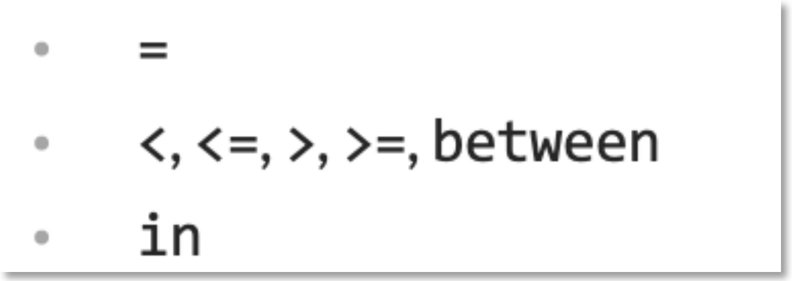
- A complete example of a B+-tree
  - Data stored only in leaves
    - No need to squeeze data into non-leaf nodes
  - Leaves are linked sequentially
    - Faster table traversal from top to bottom
    - Better support for range queries



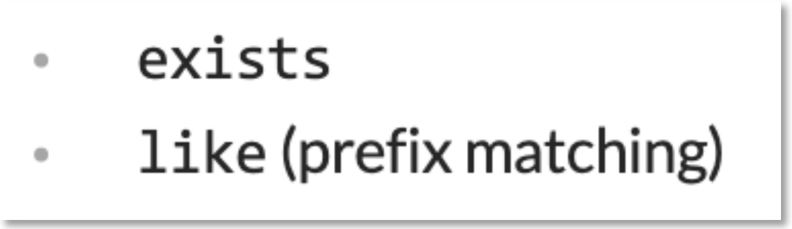
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

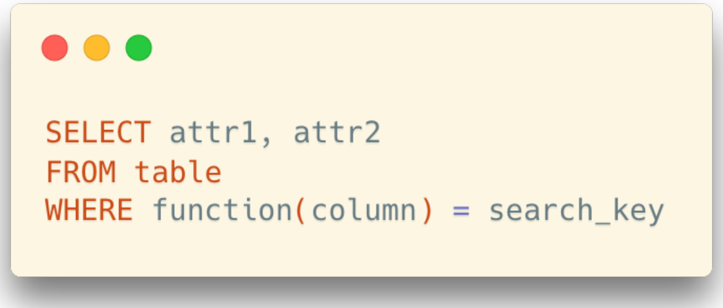


- =
- <, <=, >, >=, between
- in

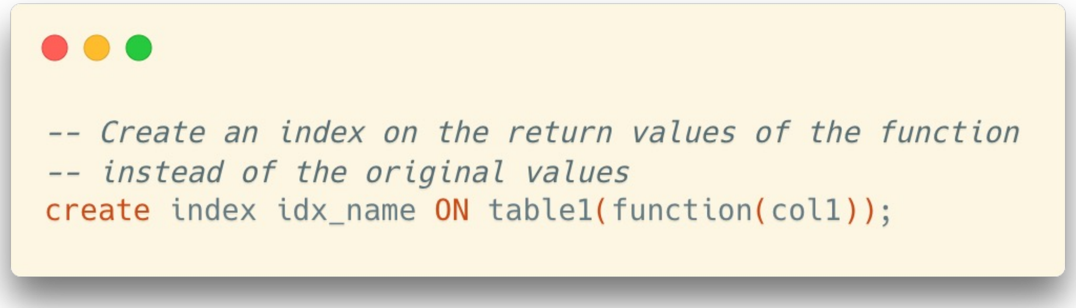


- exists
- like (prefix matching)

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

- 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

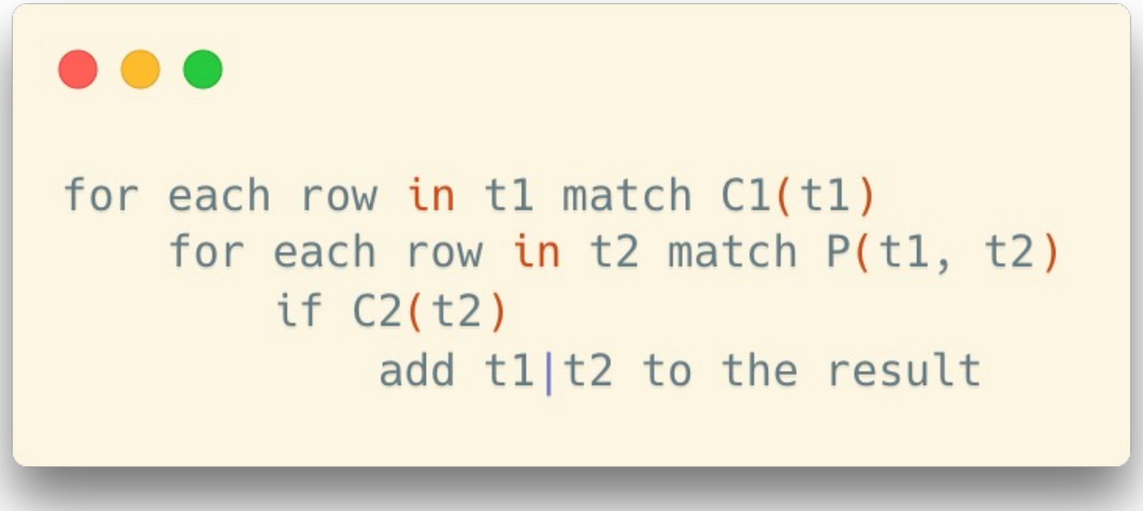
bucket 7

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.