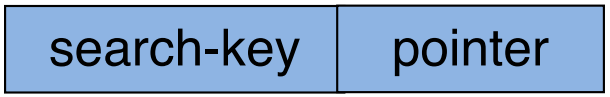
Indexing

Indexing - Basic Concepts

- Indexing mechanisms are used to speed up access to desired data
  - E.g., author catalog in library
- An **index** consists of records (called **index entries**) of the form



  - **Search Key** - attribute or set of attributes used to look up records in a file.

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Ordered Indices

- **Ordered index**
  - index entries are stored sorted on the search key value
  - e.g., author catalog in library.

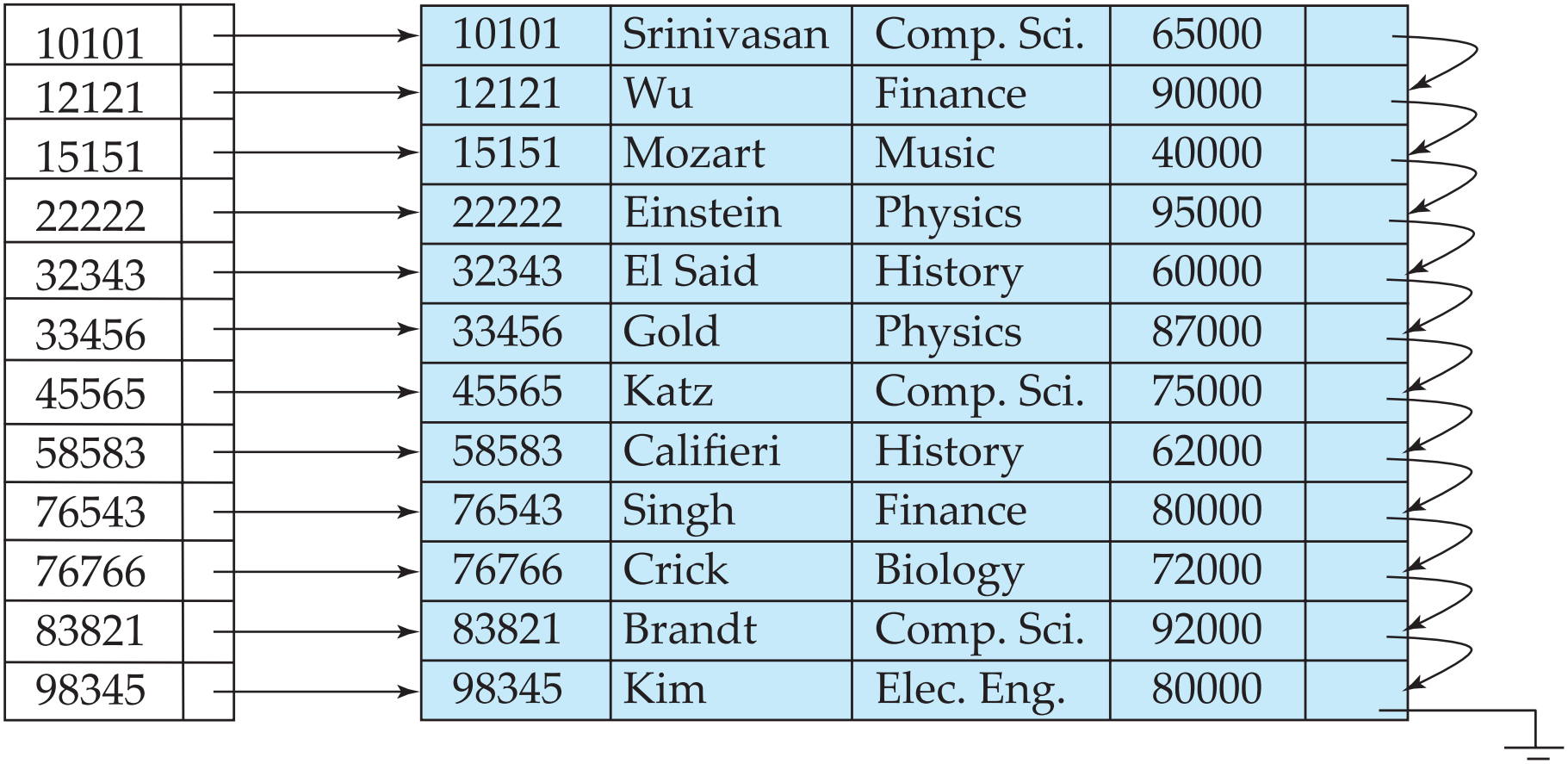
- **Primary index / clustering index**
  - in a sequentially ordered file,
  - the index whose search key specifies the sequential order of the file.
  - The search key of a primary index is usually the primary key.

- **Secondary index / non-clustering index**
  - an index whose search key specifies an order different from the sequential order of the file.

- **Index-sequential file**
  - ordered sequential file with a primary index.

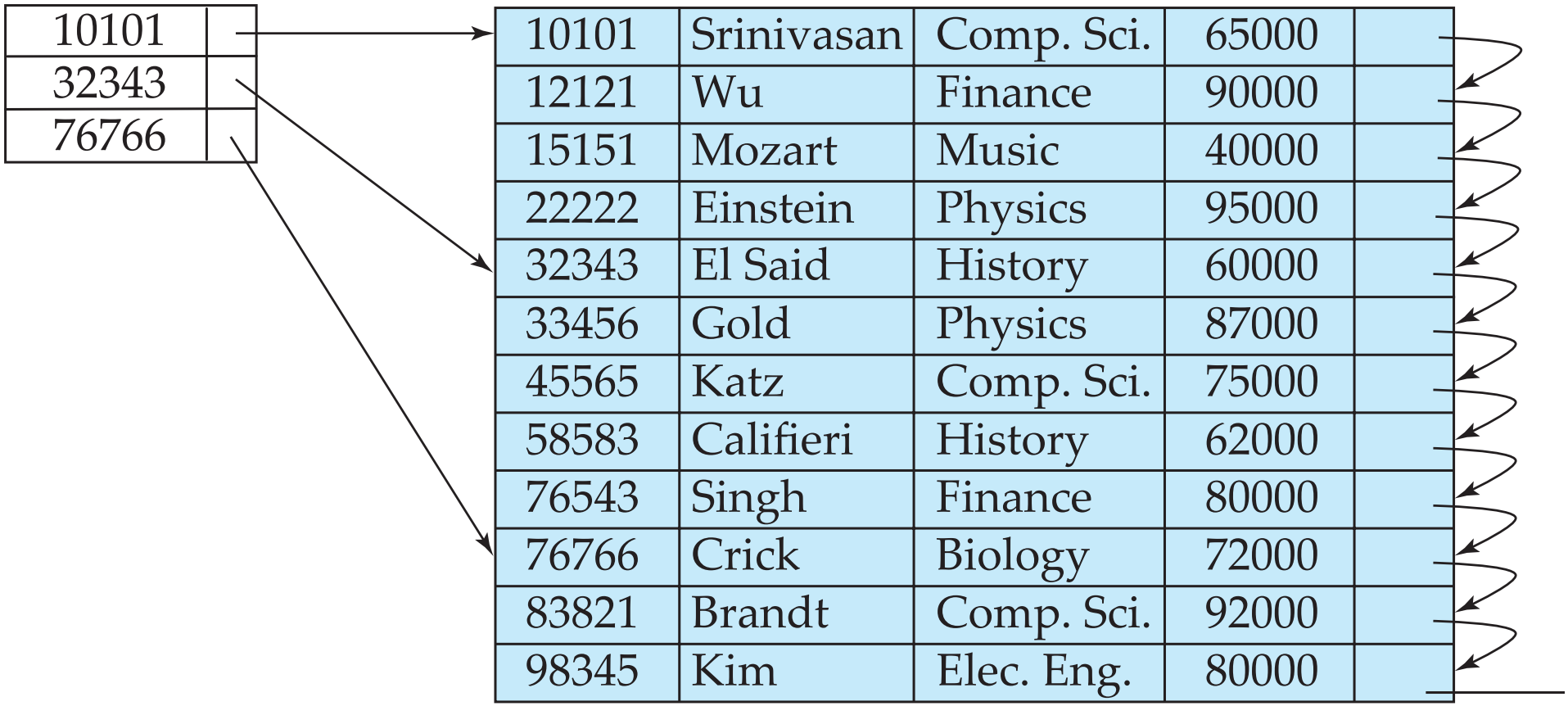
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



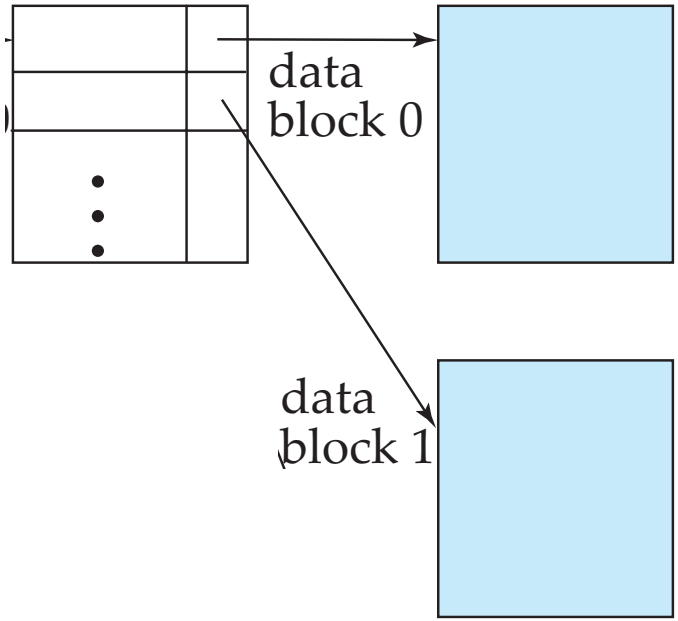
Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- E.g. index on *ID* attribute of *instructor* relation



Sparse Index Files (Cont.)

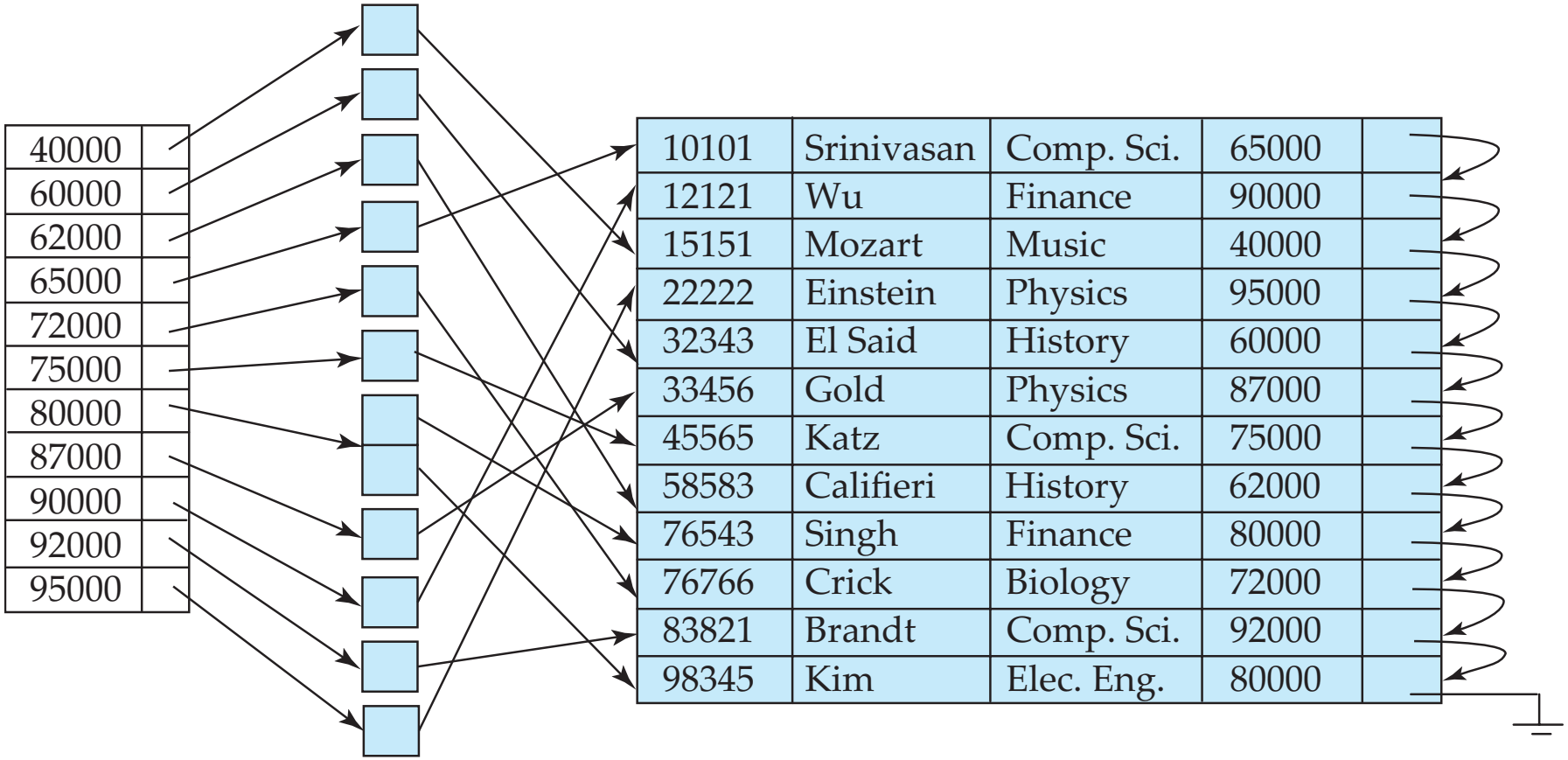
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Secondary Indices Example

- A secondary index:
  - Search key order different from the sequential order of the file
  - 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

Secondary index on *salary* field of *instructor*



Primary and Secondary Indices

- **Indices offer substantial benefits** when searching for records.

- **BUT**: Updating indices **imposes overhead on database modification** – when a file is modified, index on the file must be updated accordingly,

Multilevel Index

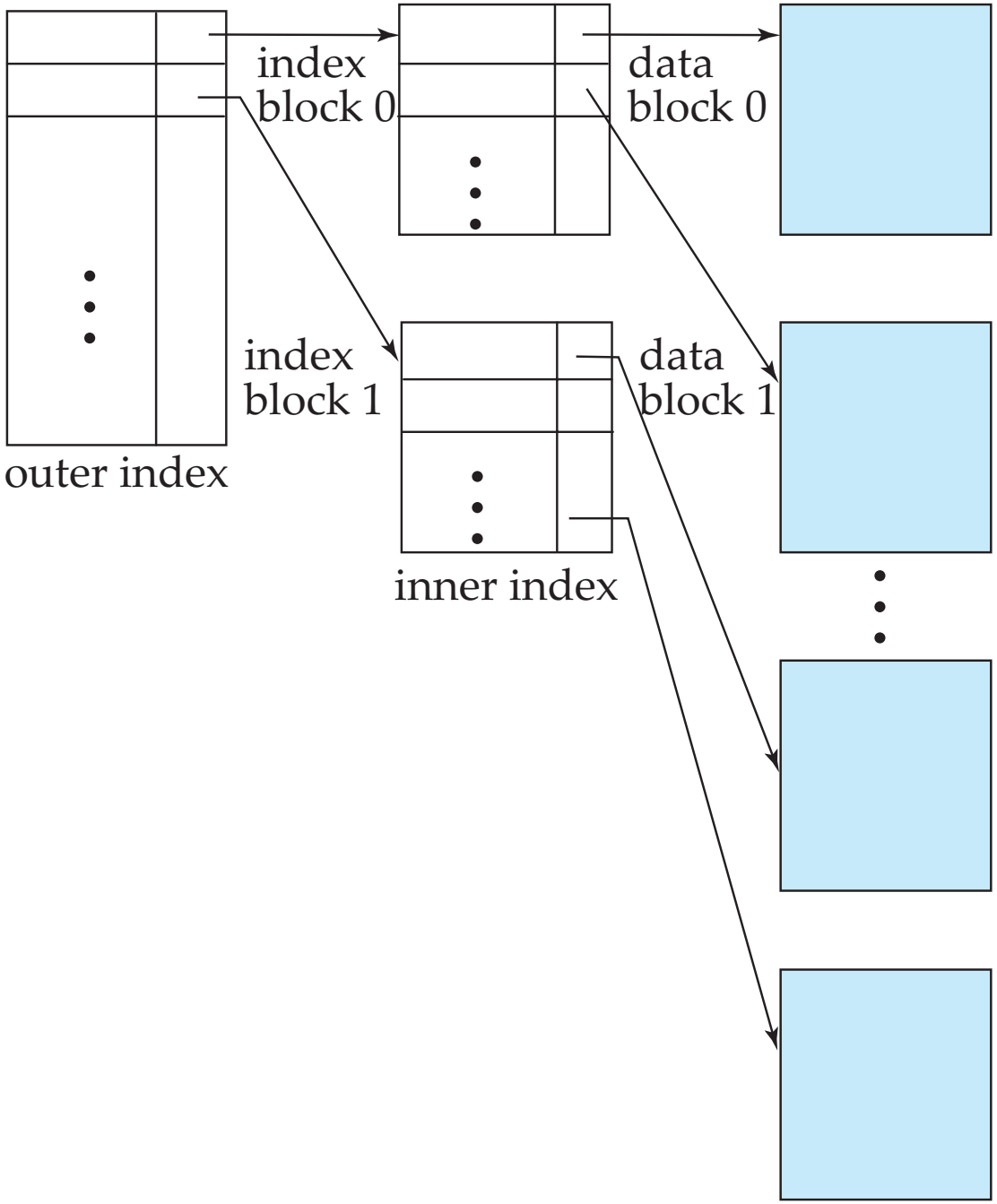
- If primary index does not fit in memory, access becomes expensive.

- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - **outer index – a sparse index of primary index**
  - **inner index – the primary 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.

- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



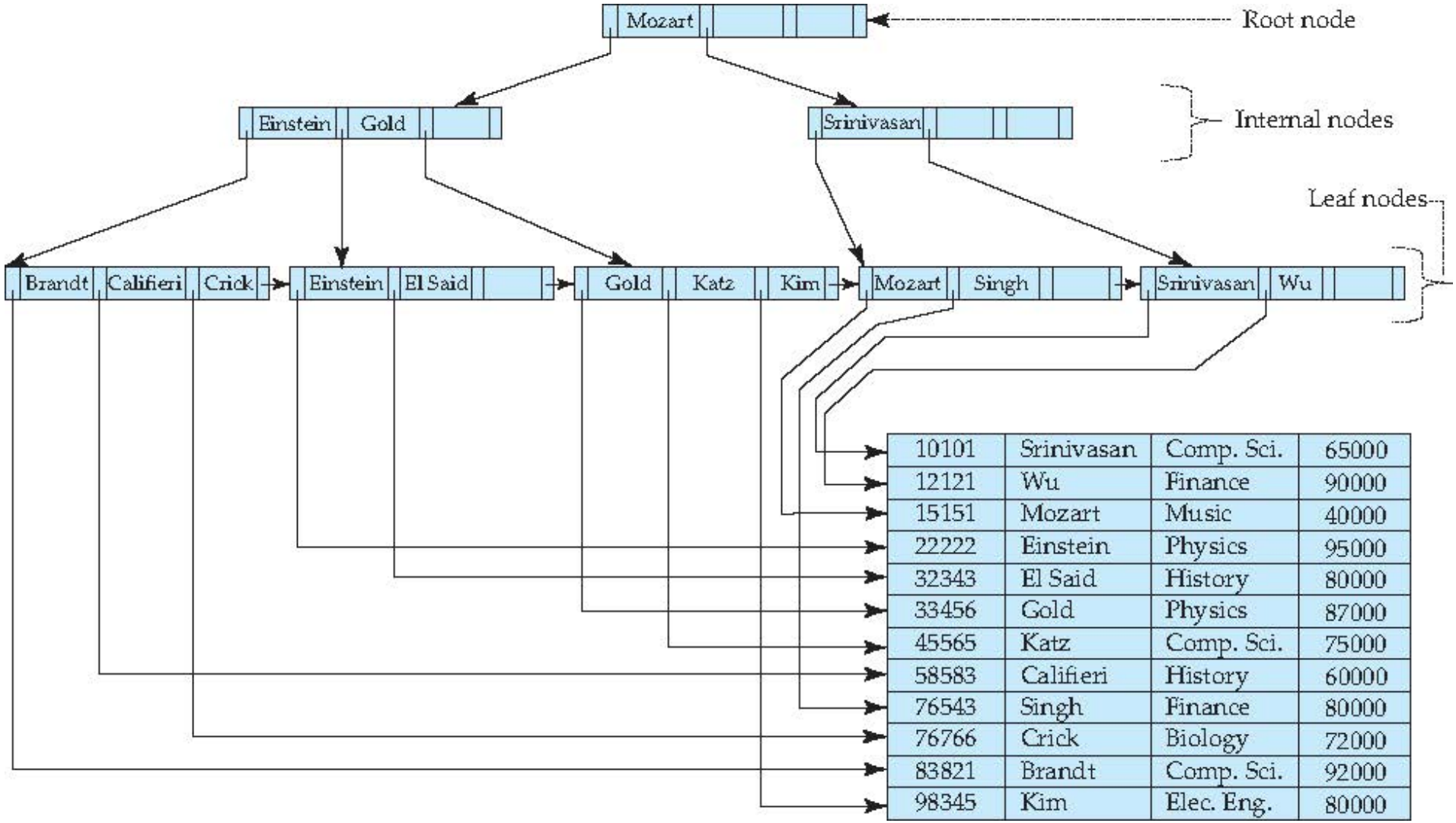
B^+-Tree Index Files

B^+-tree indices is an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local changes to accommodate insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.

- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B+-Tree



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

branching factor:

n=4 (maximum 4 children per node)

B+-Tree Index Files

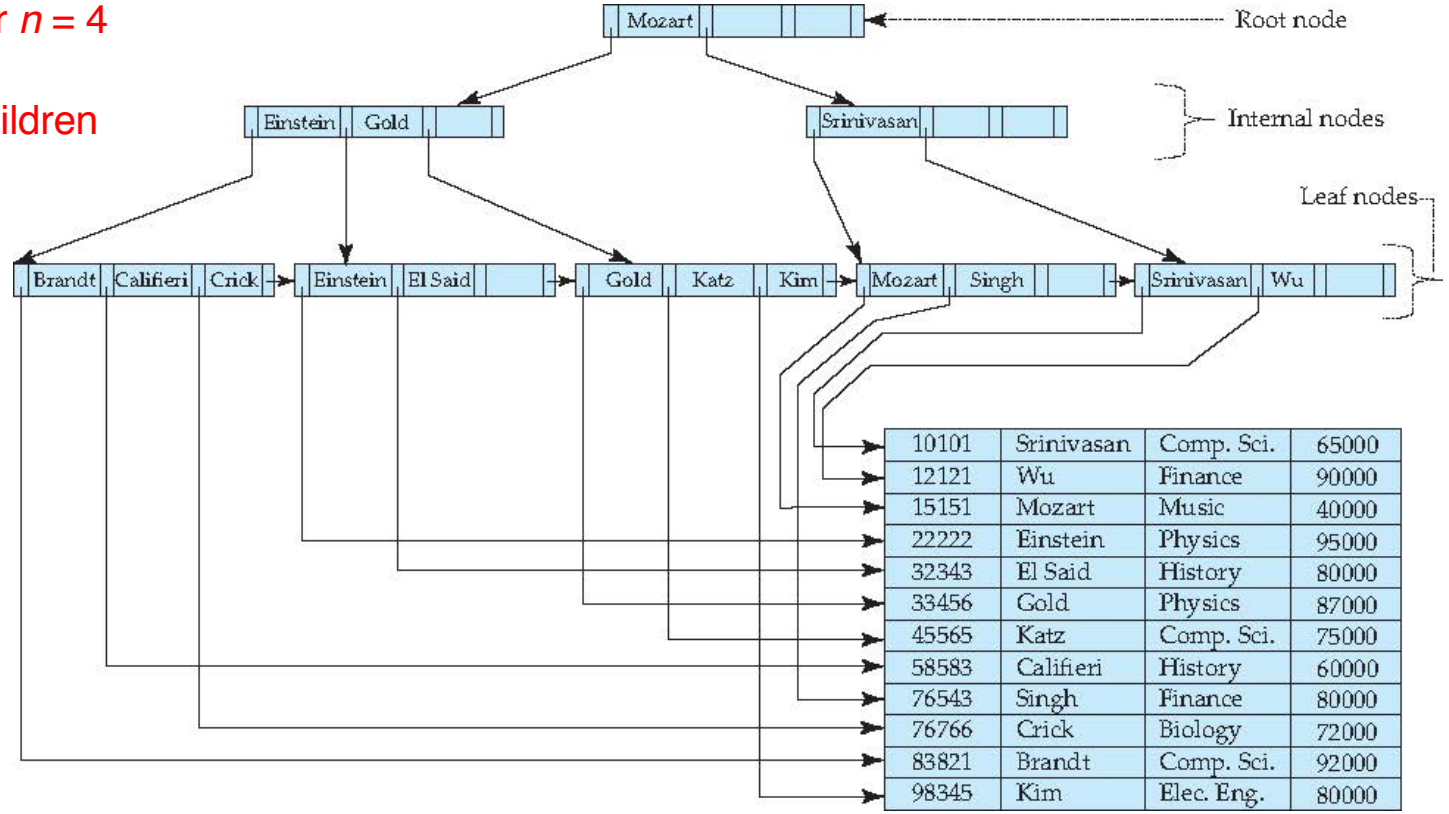
A B+-tree with branching factor n
is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values

Branching factor $n = 4$

between 2 and 4 children

between 2 and 3 values



B+-Tree Node Structure

- Typical node

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

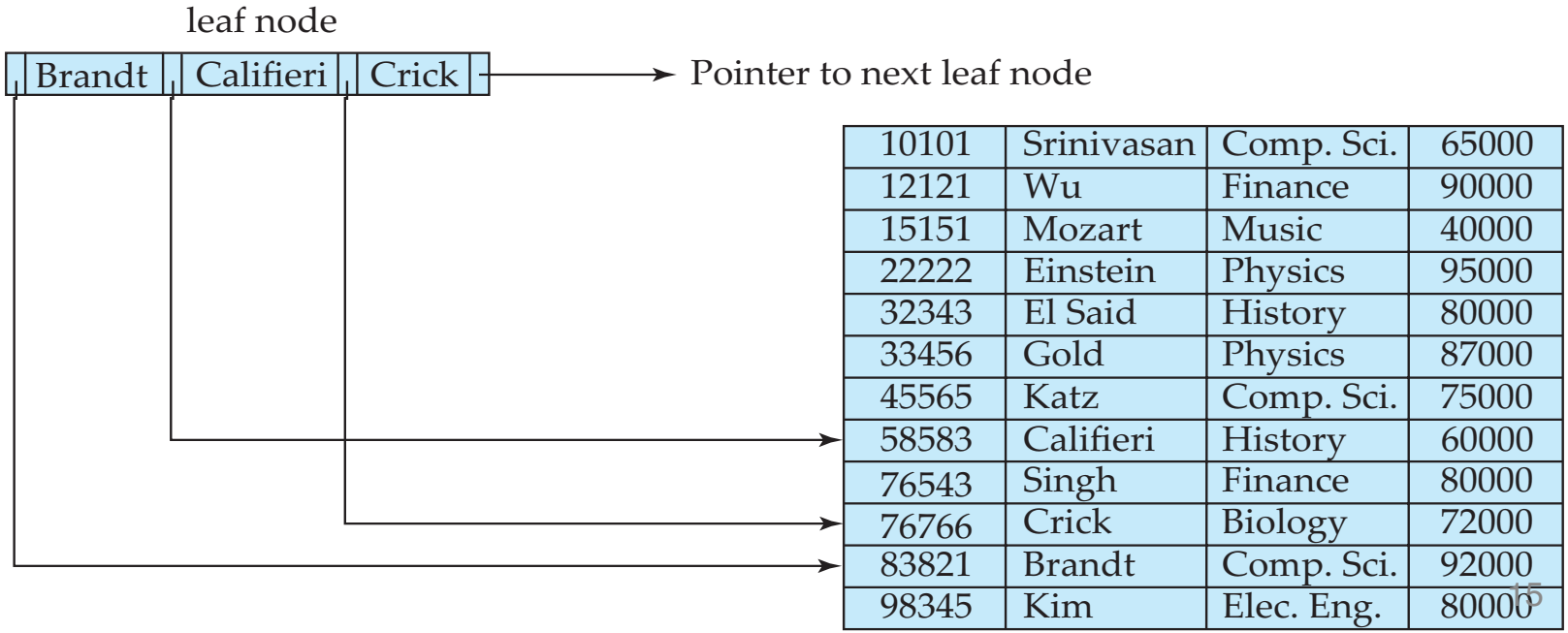
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- Leaf nodes are linked and ordered:

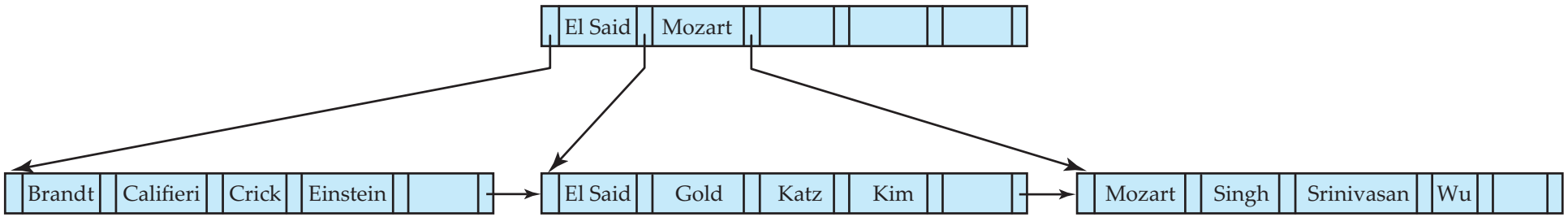
  - P_n points to next leaf node in search-key order
  - If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values



Non-Leaf Nodes in B+-Trees

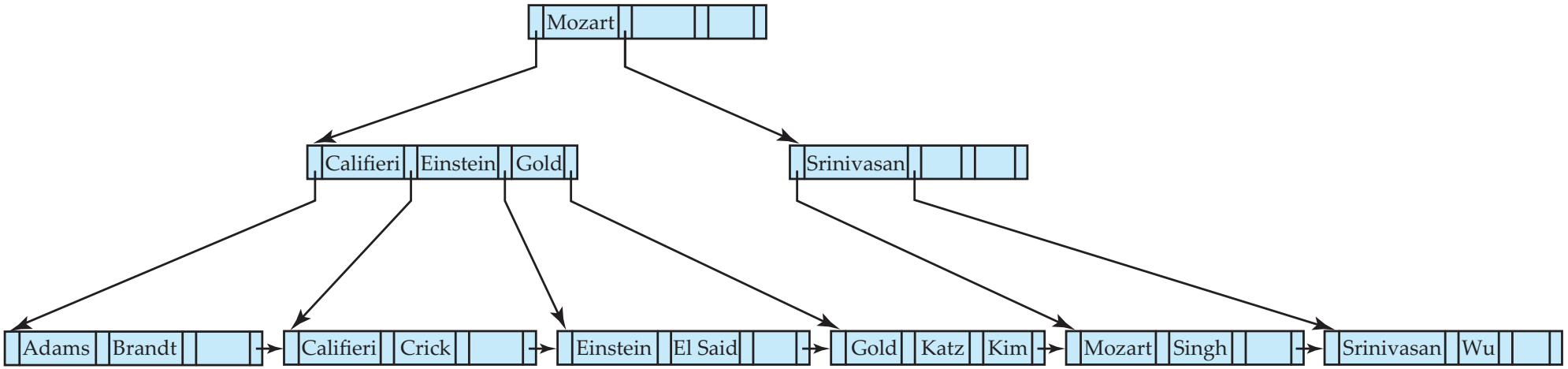
- Non leaf nodes form a multi-level sparse index on the leaf nodes.

- 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 less or equal to than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}



Queries on B+-Trees

- Find record with search-key value *V.*
  1. *C=root*
  2. While C is not a leaf node {
     1. Let *i* be least value such that $V \leq K_i$.
     2. If no such exists, set *C = last non-null pointer in C*
     3. Else { if (V= K_i) Set C = P_{i+1} else set *C* = P_i}
     }
  3. Let *i* be least value such that $K_i = V$
  4. If there is such a value *i,* follow pointer P_i to the desired record.
  5. Else no record with search-key value *k* exists.



Updates on B+-Trees

- Updates on B+-trees
  - Involve insertion or deletion of key values
  - May involve splitting and coaleshing of nodes
  - May call for balancing og the tree to ensure the required properties

- Details and update algorithms are described in the DSC book section 11.3

- More important to understand the principles than the details of updating B+-trees

Exercise, example and tool

11.3 Construct a B^+-tree for the following set of key values:

$$(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)$$

Assume that the tree is initially empty and values are added in ascending order. Construct B^+-trees for the cases where the number of pointers that will fit in one node is as follows:

a. Four

b. Six

c. Eight

- http://goneill.co.nz/btree-demo.php

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name>(<attribute-list>)

  - example

    create index *title_index* **on** *movie(title)*

- To drop an index

  drop index <index-name> **on** <relation-name>

  - example

    drop index *title_index* **on** *movie*

Why index?

- To improve the performance of SELECT operations
  - create indexes on one or more of the columns that are tested in the query.

- Index entries
  - act like pointers to the table rows, allowing the query to quickly determine which rows match a condition in the WHERE clause

- All columns can be indexed.
  - tempting to do so
  - however, unnecessary indexes waste space and time
    - to determine which indexes to use may be difficult for the optimizer
    - indexes add to the cost of inserts, updates, and deletes
  - find the right balance

Index Definition in MySQL

- MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are (mostly) stored in B-trees.

- Exceptions:
  - Indexes on spatial data types use R-trees; MEMORY tables also support hash indexes;
  - Inverted lists are used for FULLTEXT indexes.

Index use in MySQL

- MySQL uses indexes as follows
  - to find the rows matching a WHERE clause quickly
  - to eliminate rows from consideration
  - to retrieve rows from other tables when performing joins
  - to find the MIN() or MAX() value for a specific indexed column
  - to sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index

What to consider

- Considerations
  - If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows (the most selective index).
  - in a multiple-column index, any leftmost prefix can be used

- Indexes are less important
  - on small tables,
  - on big tables mainly accessed by “report queries” processing most or all of the rows
    - when a query needs to access most of the rows, reading sequentially is faster than working through an index.

Index Types in MySQL

- The primary key index
  - The primary key always has an associated index,

- Column Indexes
  - The most common type of index involves a single column,
  - The B-tree data structure lets the index quickly
    - find a specific value, a set of values, or a range of values, corresponding to operators such as **=**, **>**, **<**, **BETWEEN**, **IN**, and so on, in a **WHERE** clause.

The movie database

- 3 tables: movie, casting, person:
  - **movie(mid, title, production_year)**
  - **casting(mid, pid)**
  - **person(pid, name, gender)**

Not all rows from the 3 tables are shown here

Full content are:
- movie: 887.047 rows
- casting: 6.995.056 rows
- person: 2.422.836 rows

movie

mid	title	production_year
2438281	Accordion Player	1888
2546319	Brighton Street Scene	1888
2831850	Hyde Park Corner	1889
3004390	Man Walking Around the Corner	1887
3127044	Passage de Venus	1874
3205178	Roundhay Garden Scene	1888
3214201	Sallie Gardner at a Gallop	1878
3462286	Traffic Crossing Leeds Bridge	1888

casting

pid	mid
1158190	2438281
1158190	3205178
2197644	3205178
2743436	3205178
3489247	3205178

person

pid	name	gender
1158190	Le Prince, Adolphe	m
2197644	Whitley, Joseph	m
2743436	Hartley, Annie	f
3489247	Whitley, Sarah	f

A test on the movie database

- 3 new tables copying data from the existing

```
create table movie1 as (select * from movie);
create table casting1 as (select * from casting);
create table person1 as (select * from person);
```

- a clear performance difference ...:

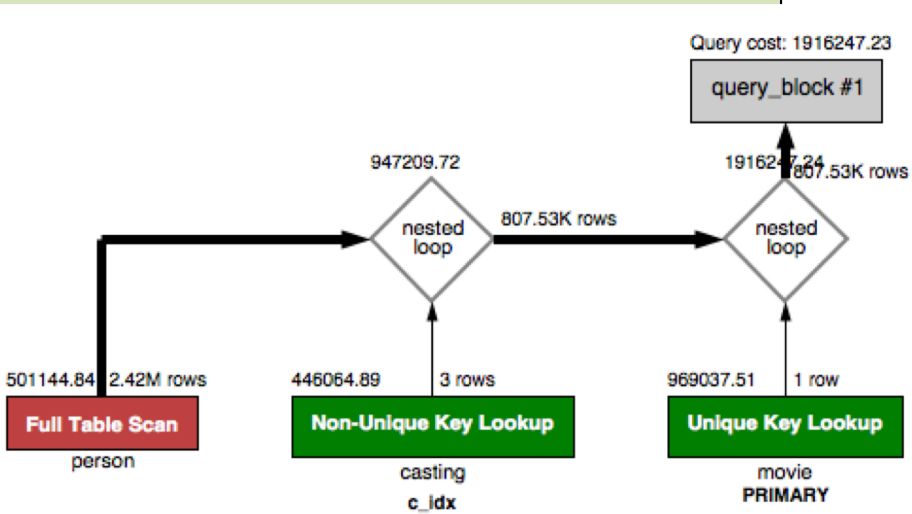
```
mysql> select count(*)
from movie natural join casting natural join person  where
name like'Pacino%';
...
1 row in set (0.35 sec)

mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (14.25 sec)
```

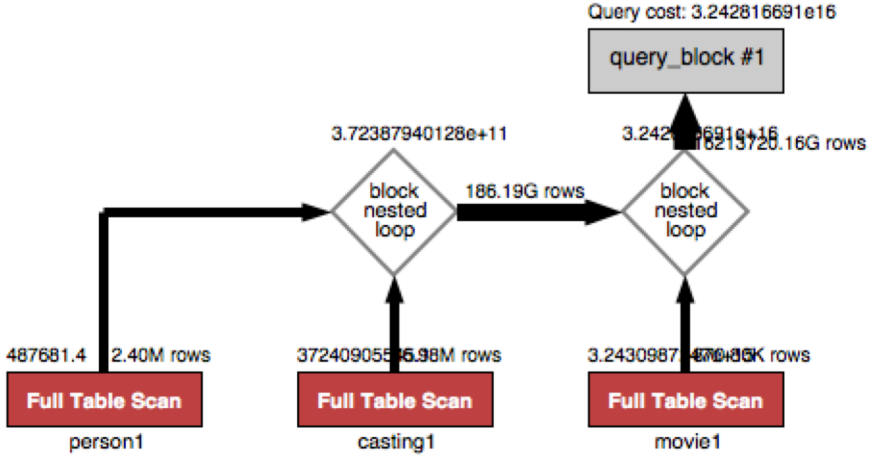
A test on the movie database

- compare the query plans

```
mysql> select count(*)
from movie natural join casting natural join person
where name like'Pacino%';
...
1 row in set (0.35 sec)
```




```
mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (14.25 sec)
```

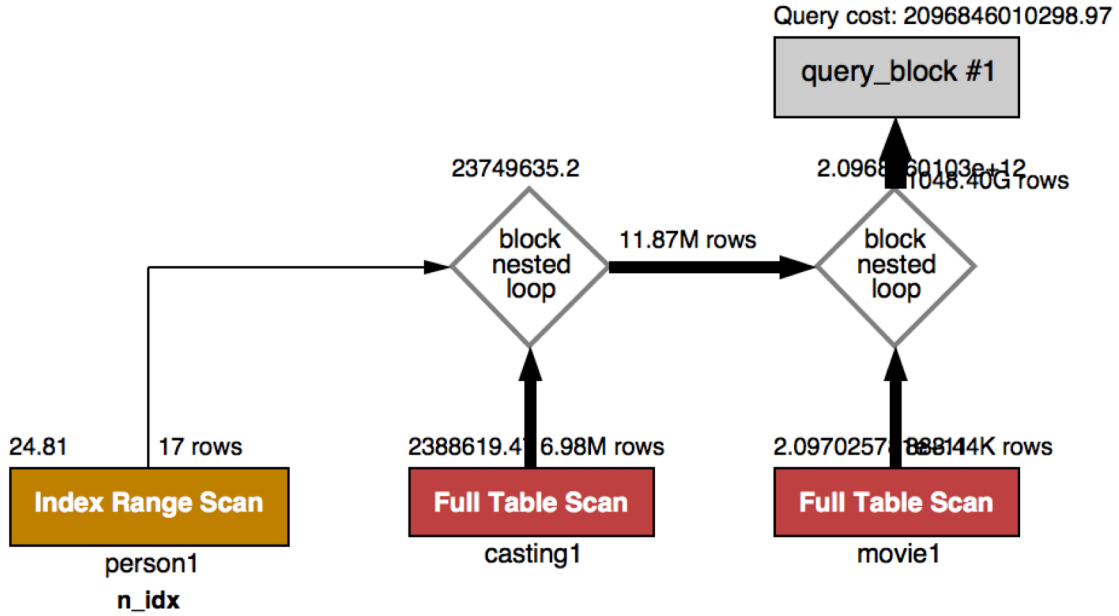


A test on the movie database

- adding an index on name in person1

```
mysql> create index `n_idx` on person1(`name`);
mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (13.46 sec)
```

- not much change here

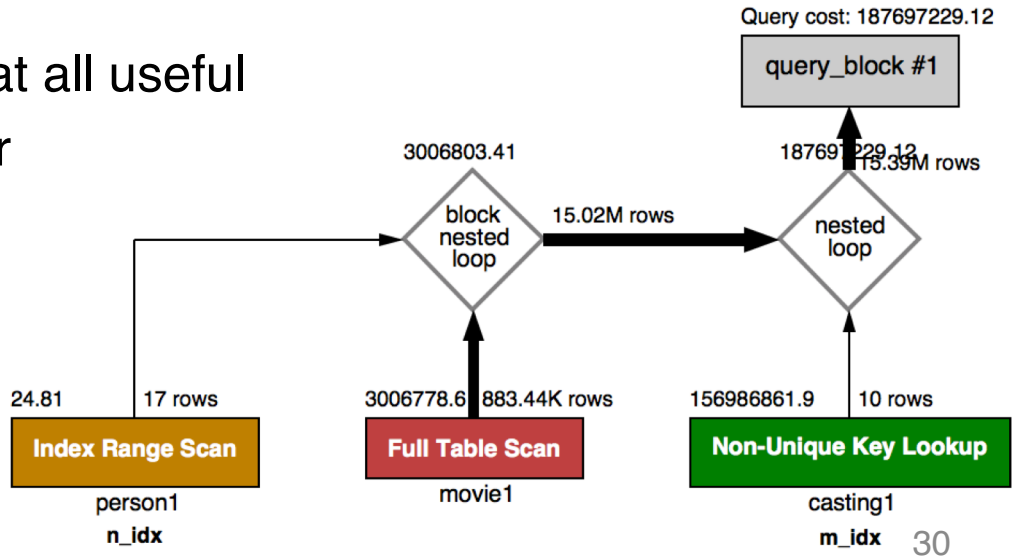


A test on the movie database

- indexing **mid on casting1**:

```
mysql> create index `m_idx` on casting1(`mid`);
mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (3 min 7.22 sec)
```

- significantly worse performance!
- an index on mid is apparently not at all useful
- and even, as we see, the optimizer fails to ignore it

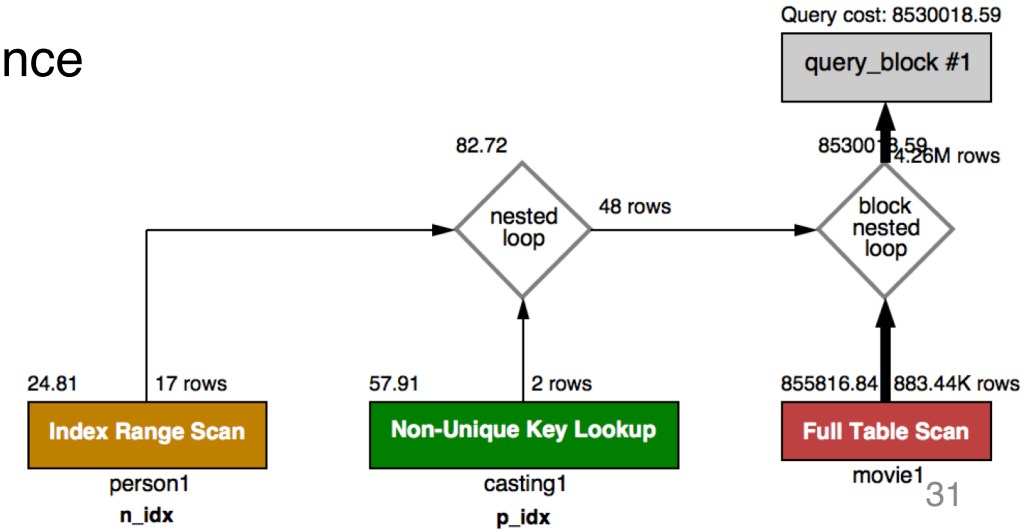


A test on the movie database

- rather we can try to index **pid on casting1**

```
mysql> drop index `m_idx` on casting1;
mysql> create index `p_idx` on casting1(`pid`);
mysql> select count(*) from movie1 natural join casting1 natural join
person1 where name like'Pacino%';
...
1 row in set (5.22 sec)
```

- this clearly improves the performance
- so why is this better?

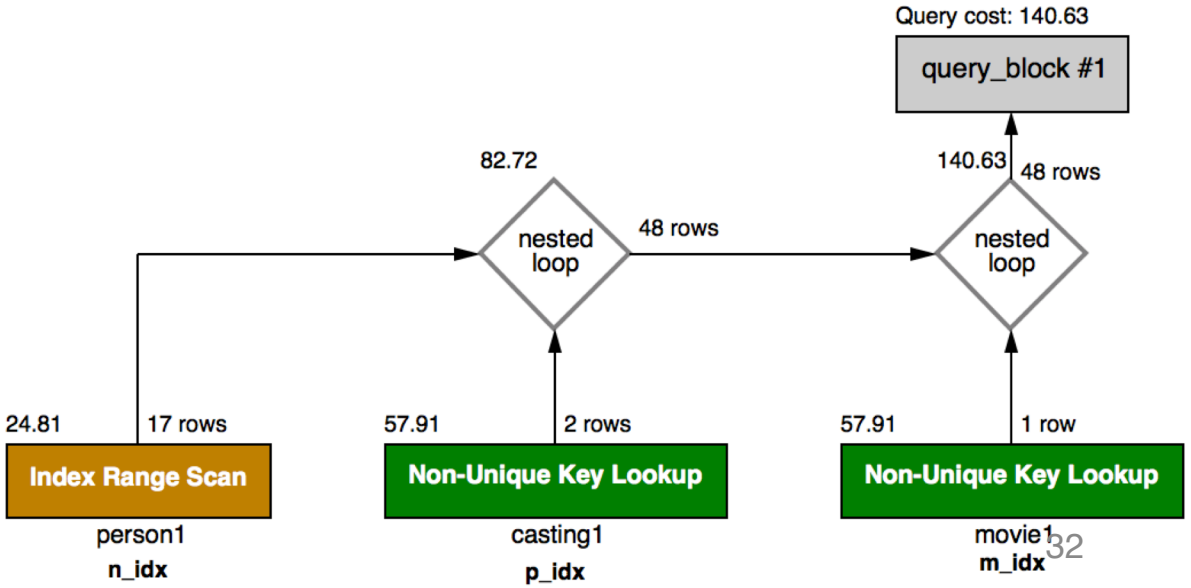


A test on the movie database

- an finally to avoid the full table scan on movie1:
  - adding an index on **movie1(mid)**

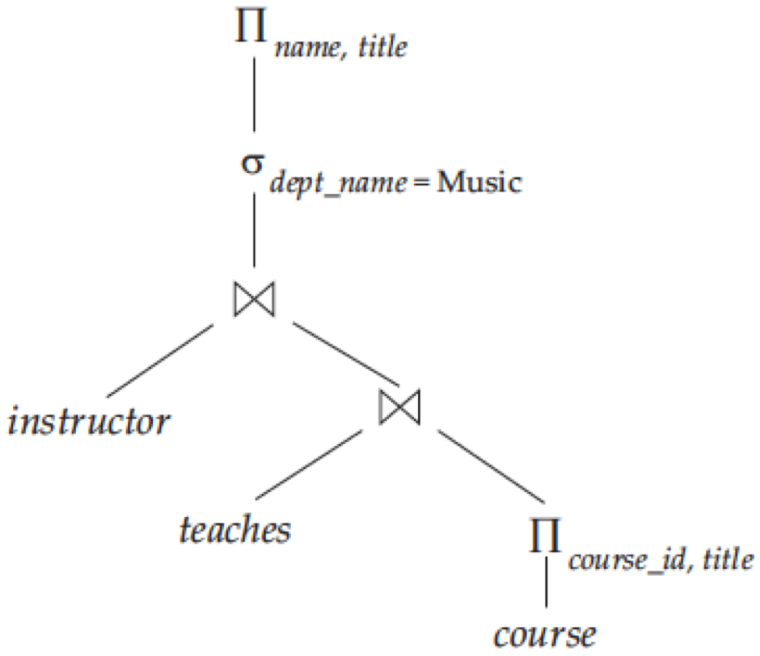
```
mysql> create index `m_idx` on movie1(`mid`);
mysql> select count(*)  from movie1 natural join casting1 natural join
person1  where name like'Pacino%';
...
1 row in set (0.00 sec)
```

- even better than what we saw in the first place

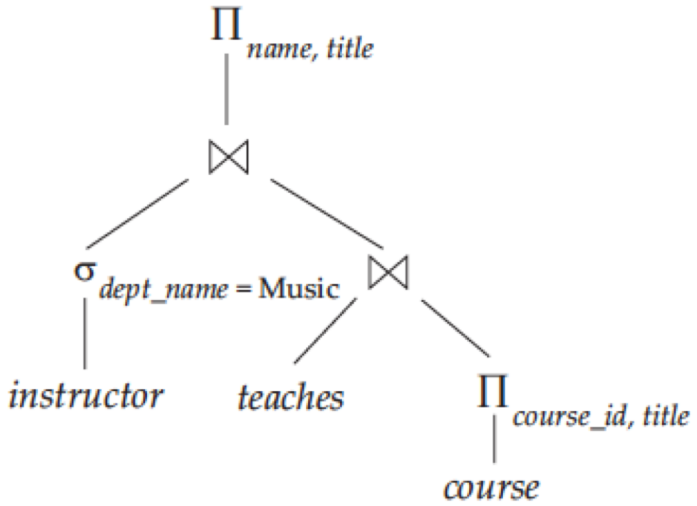


A query and two alternative plans

- Query
  - SELECT name, title
    FROM course, teaches, instructor
    WHERE teaches.course_id = course.course_id
    AND instructor.id = teaches.id
    AND dept_name="Music"



$$\Pi_{name,title}\,(\sigma_{dept\_name="Music"}\,(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))))$$

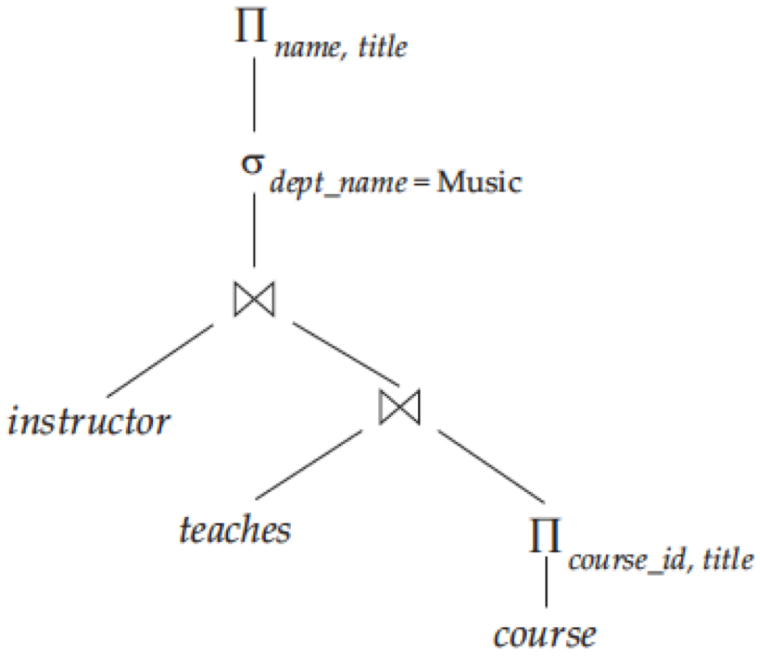


$$\Pi_{name,title}\,((\sigma_{dept\_name="Music"}\,(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,title}(course)))$$

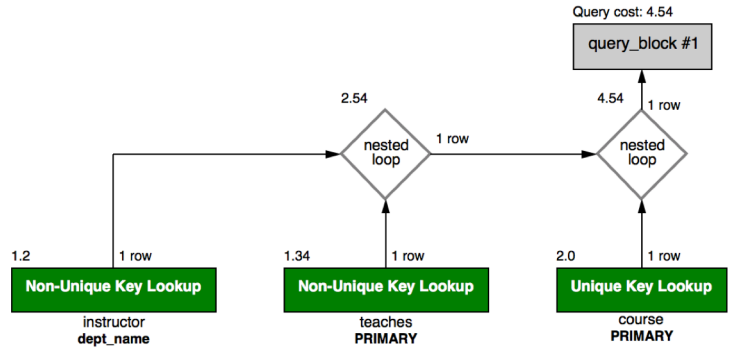
Which is better?

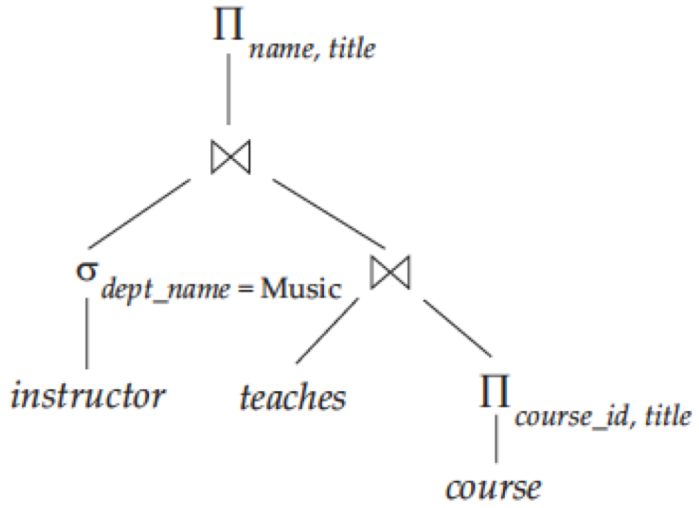
A query and two alternative plans

- Query
  - SELECT name, title
    FROM course, teaches, instructor
    WHERE teaches.course_id = course.course_id
    AND instructor.id = teaches.id
    AND dept_name="Music"



$$\Pi_{name,title}\,(\sigma_{dept\_name="\text{Music}"}\,(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))))$$

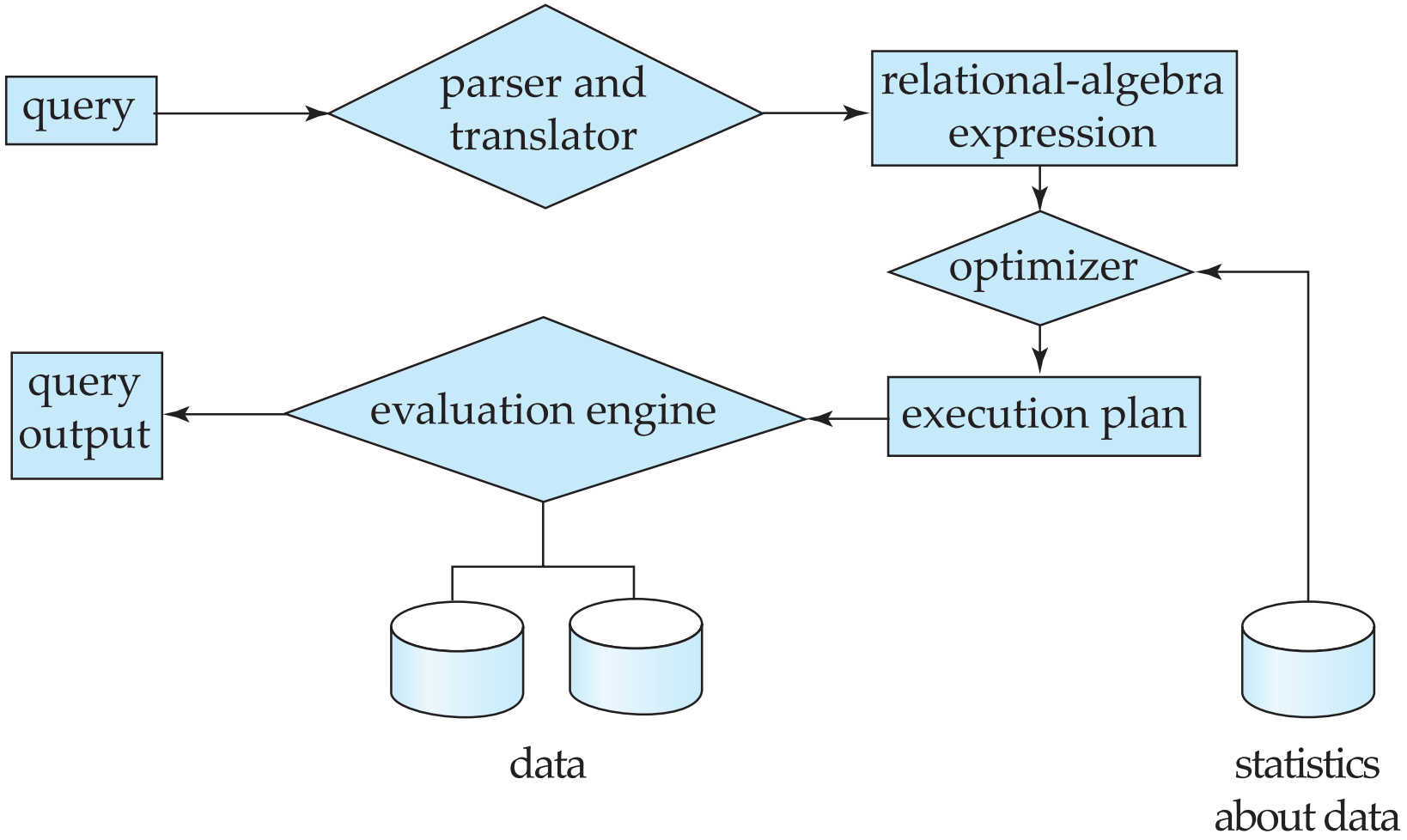




$$\Pi_{name,title}\,((\sigma_{dept\_name="\text{Music}"}\,(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,title}(course)))$$

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.