Indexing
(a brief hands-on)

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.

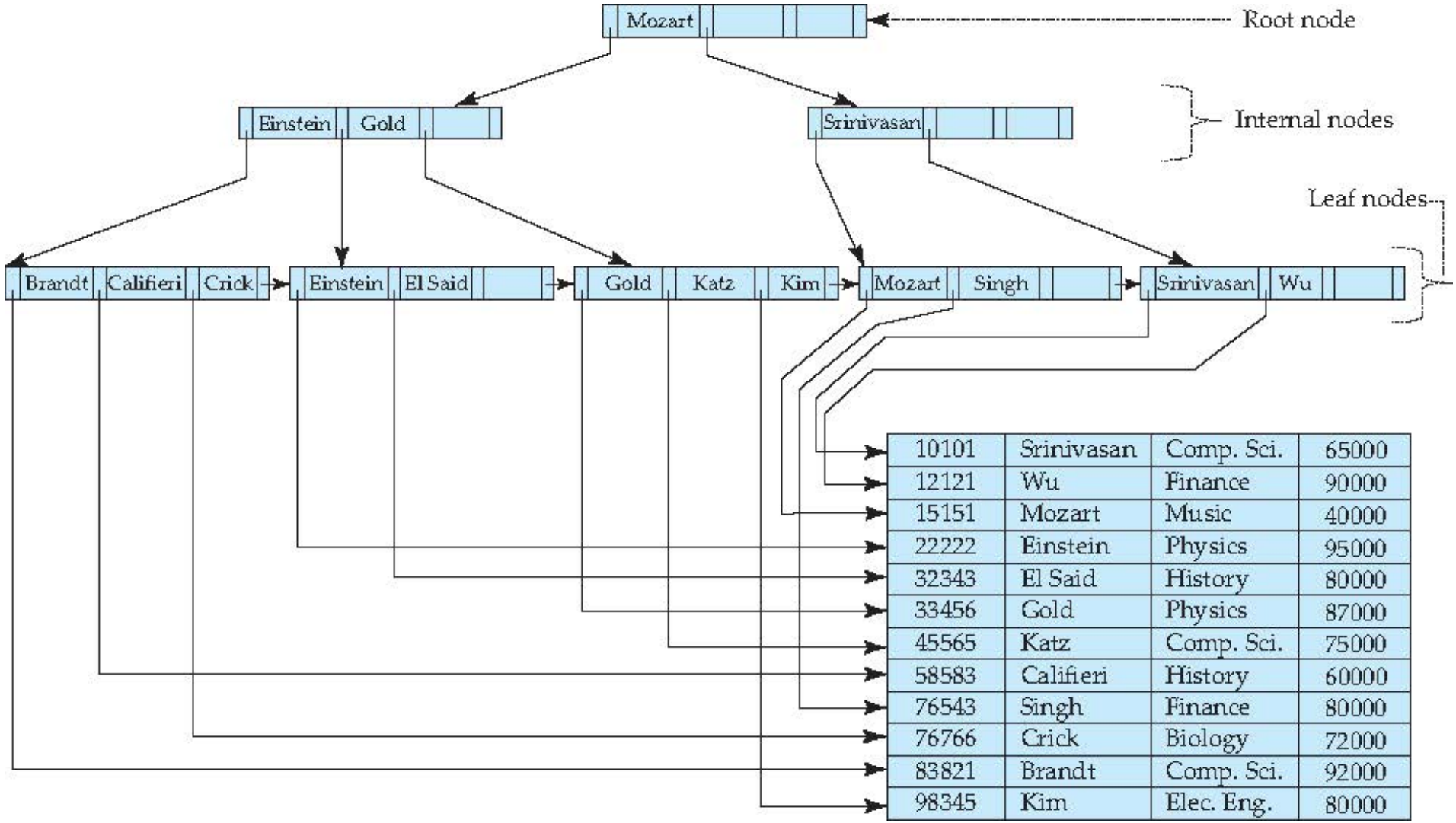
- Index entries
  - act like pointers to the table rows, allowing the query to quickly determine which rows match a condition in the WHERE clause

- 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 are (mostly) stored in B-trees.

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:

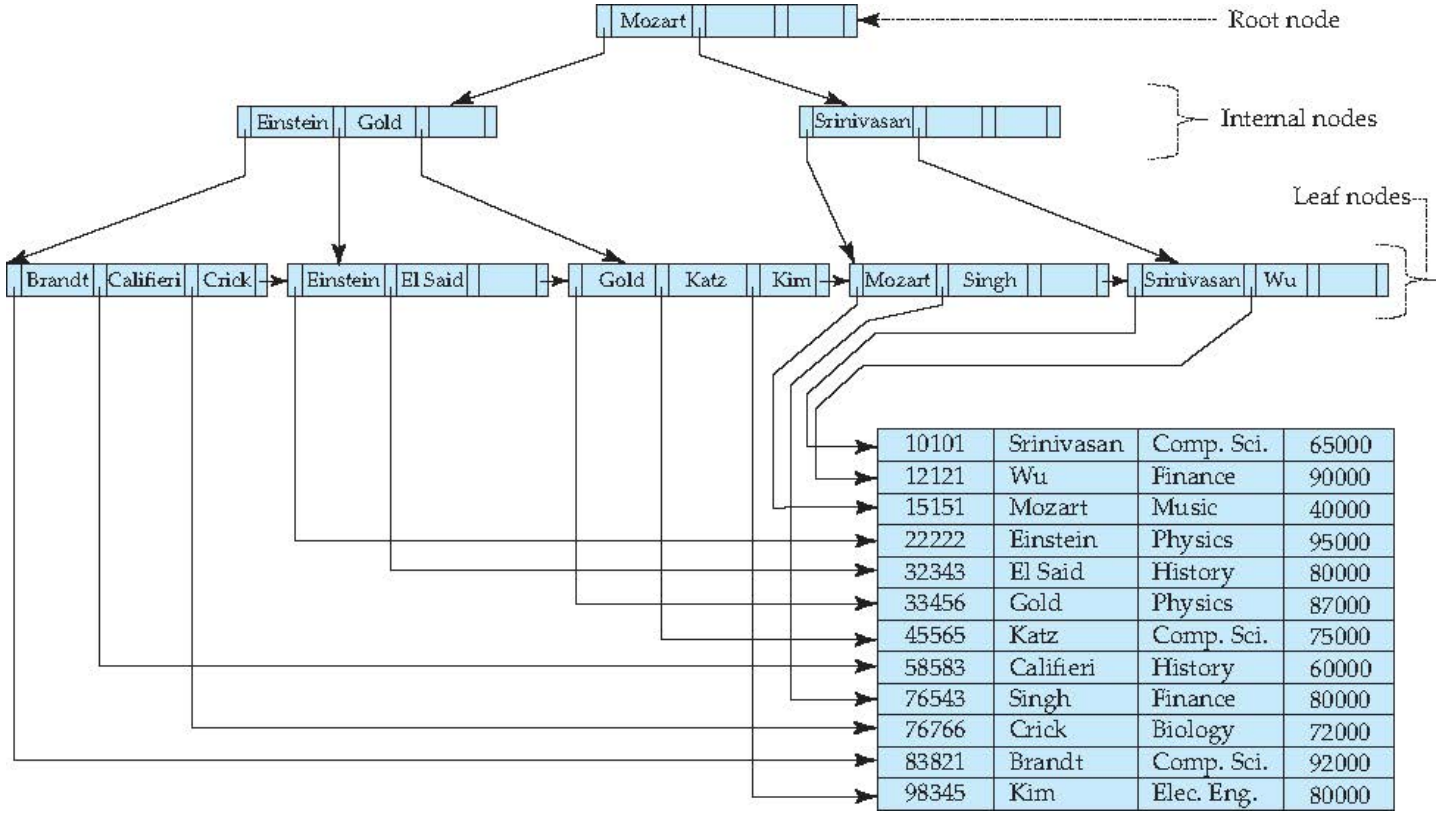
n=4 (maximum 4 children per node)

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



B+-Tree Node Structure

- Typical node

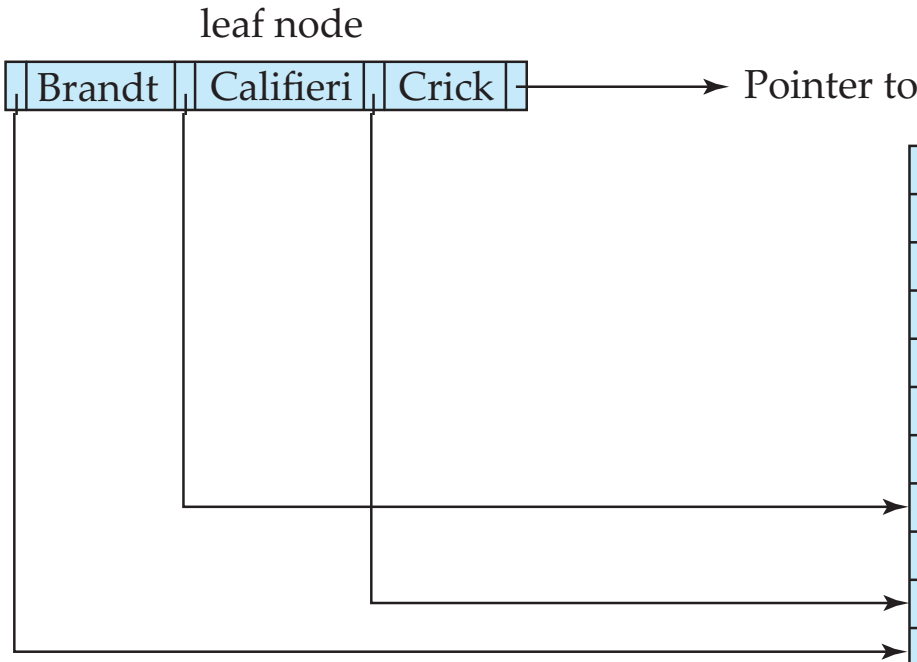
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

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,
- 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
- P_n points to next leaf node in search-key order



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 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,
  - for a table without an obvious primary key, an option is
    - create a separate column with auto-increment values to use as the primary key.

- 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 ...:

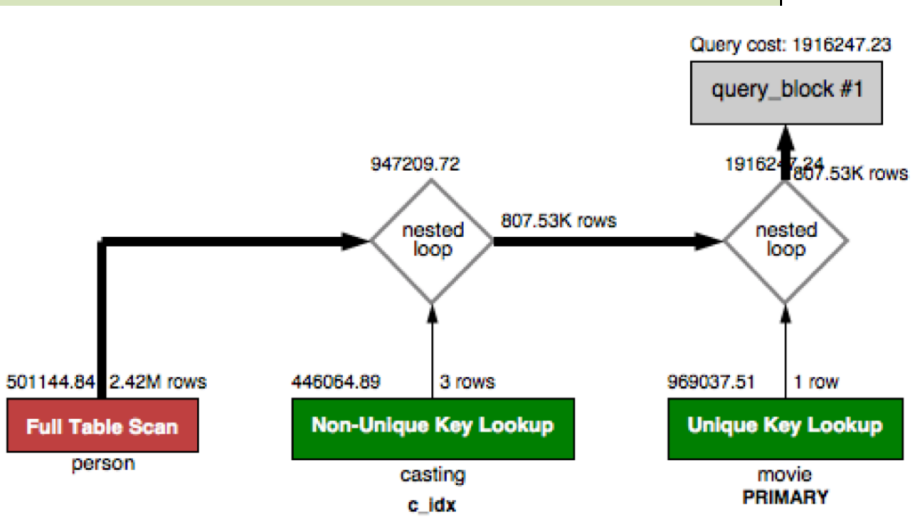
```
mysql> select count(*)
from movie natural join casting natural join person
where name like'Pacino%';
...
1 row in set (0.42 sec)

mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (18.82 sec)
```

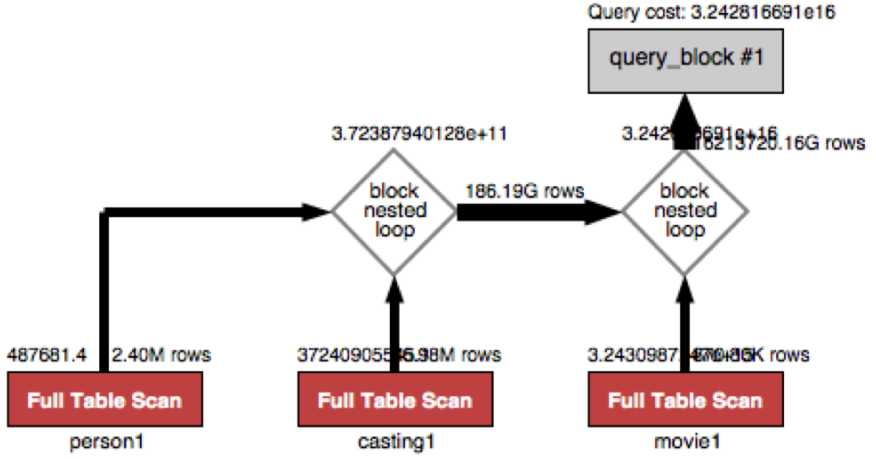
A test on the movie database

- compare the query plans

```
mysql> select count(*)
from movie natural join casting natural join person
where name like'Pacino%';
...
1 row in set (0.42 sec)
```




```
mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (18.82 sec)
```

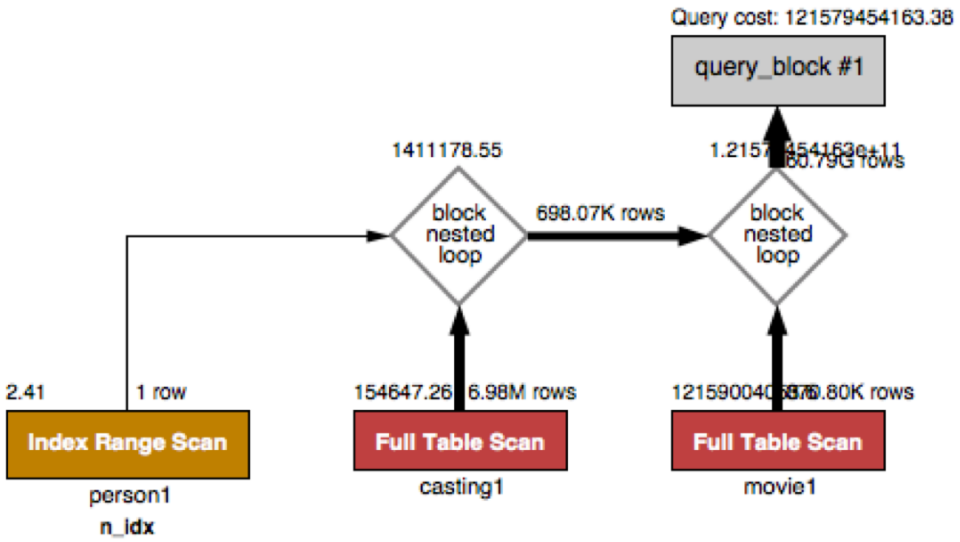


A test on the movie database

- adding a special purpose index

```
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 (18.79 sec)
```

- not much change here

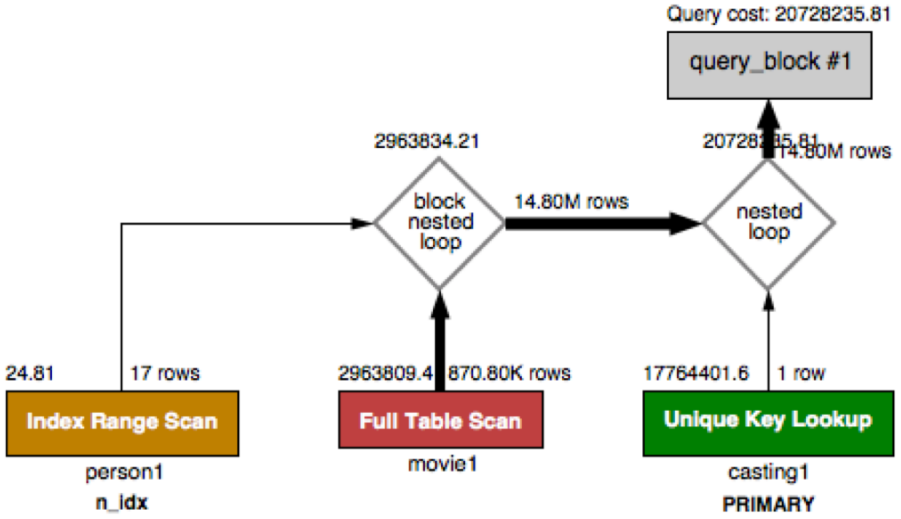


A test on the movie database

- adding a primary key on (mid, pid)

```
mysql> alter table casting1 add constraint PRIMARY KEY (`mid`,`pid`);
mysql> select count(*)
from movie1 natural join casting1 natural join person1
where name like'Pacino%';
...
1 row in set (18.82 sec)
```

- not much change here either

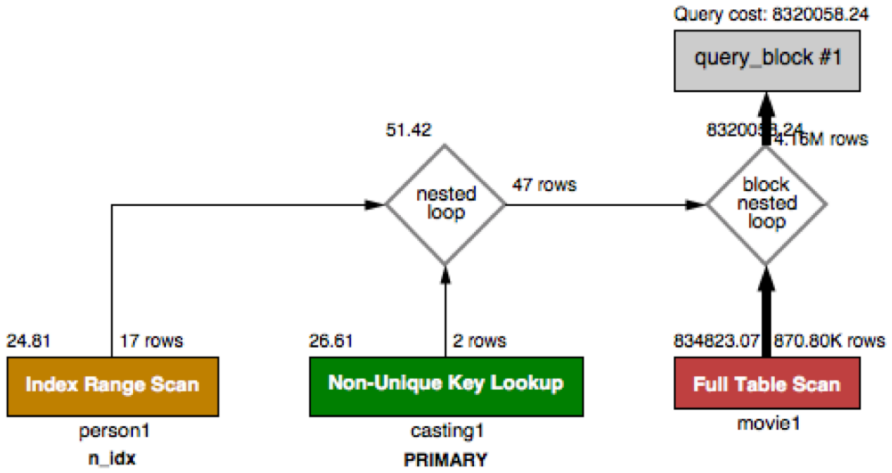


A test on the movie database

- rather than primary key on (mid, pid) as above
  we can try with one on (pid, mid)

```
mysql> alter table casting1 drop PRIMARY KEY;
mysql> alter table casting1 add constraint PRIMARY KEY (`pid`,`mid`);
mysql> select count(*) from movie1 natural join casting1 natural join
person1 where name like'Pacino%';
...
1 row in set (4.99 sec)
```

- this gives a significant difference



A test on the movie database

- an finally to avoid the full table scan on movie1:
  - adding a primary key on movie1(mid)

```
mysql> alter table movie1 add constraint PRIMARY KEY (`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 where the tables was "conventionally" indexed by primary keys
- why?