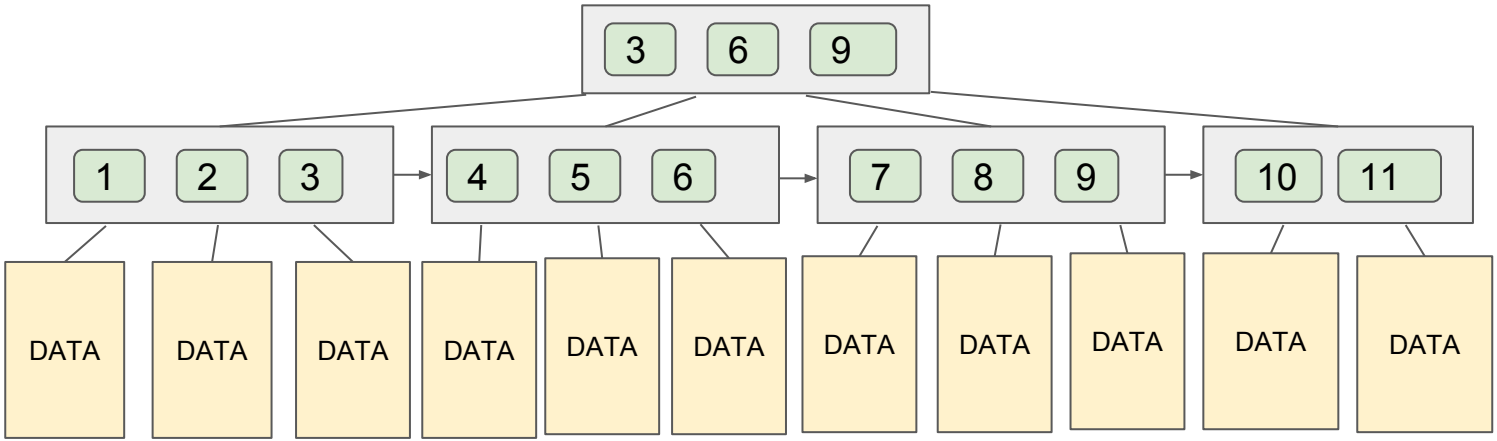
MySQL Indexing Crash Course

Aaron Silverman

Why Use Indexes

How does MySQL store stuff?

- InnoDB Stores Data on Disk
- Multiple Tables are grouped together (by default) to reduce penalty of filesystem access
- Row data stored together unless too large (768B+), then stored “off-table”
- Order of the rows stored on disk is how the data is “clustered”
- Data is indexed using sorted B+ trees that (ideally) fit into memory with data pointers in leaf nodes



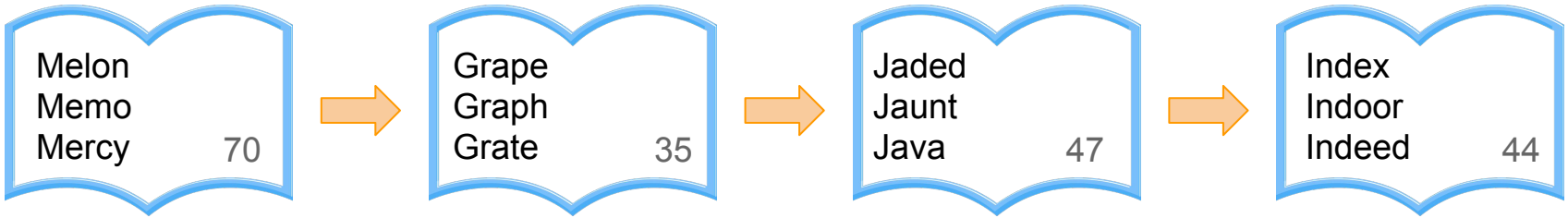
How do indexes work?

B-Tree Indexes work similarly to looking something up in a physical dictionary, iteratively comparing if what you are searching for comes before or after certain subsets of data

Dictionary Example:

- Flip to page near the middle
- Word earlier? Try a page closer to front
- Word later? Try a page closer to end
- Repeat until you find page with word

Example: Looking up “Index” in a dictionary



B-Tree is Great for Single and Range Based Lookups

GOOD

BAD

O(lg(n))

```
WHERE id = 5
WHERE id > 7
WHERE id < 5
WHERE id BETWEEN 6 AND 8
```

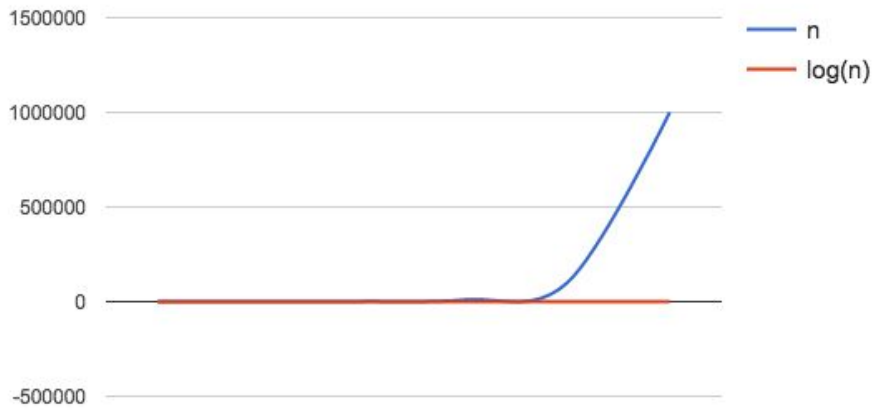
O(n)

```
WHERE id <> 8
```

Another way to think about it: If you had a physical copy of a dictionary it would be easy for you to find words that start with “ch”, but would be very hard (involve reading every page) to find words that have a “ch” in the middle or end.

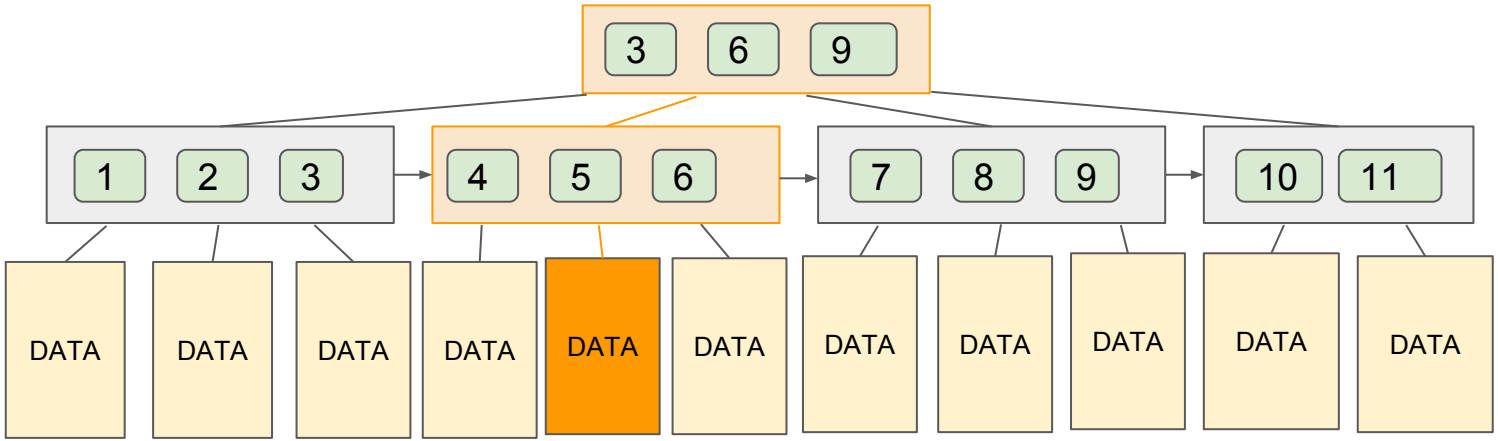
log(n) performance makes a huge difference with large datasets!

n	log(n)
10	1
100	2
1000	3
10000	4
100000	5
1000000	6

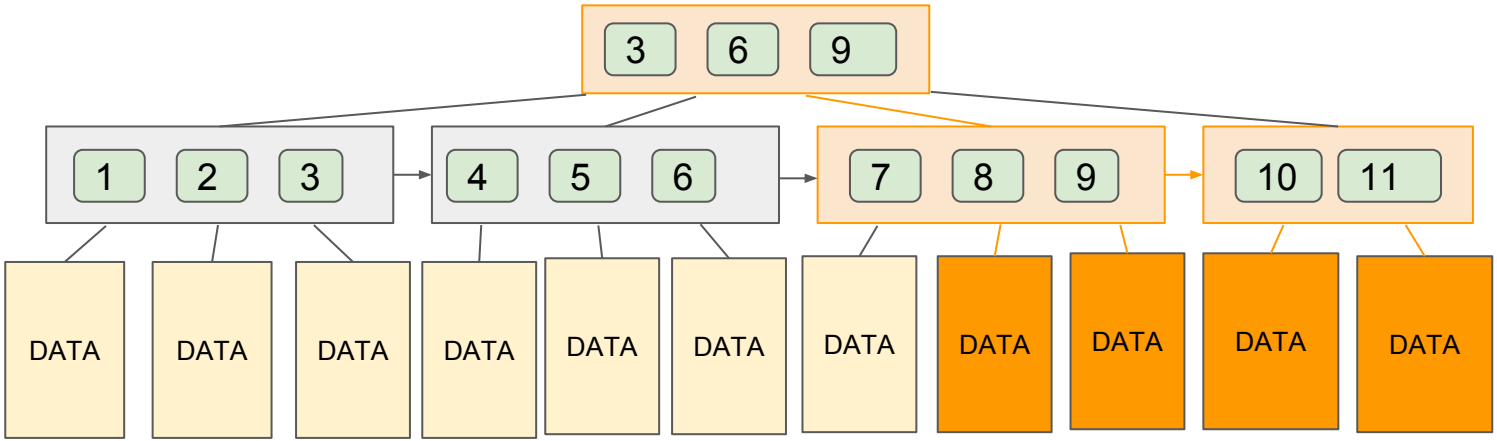


Example Queries

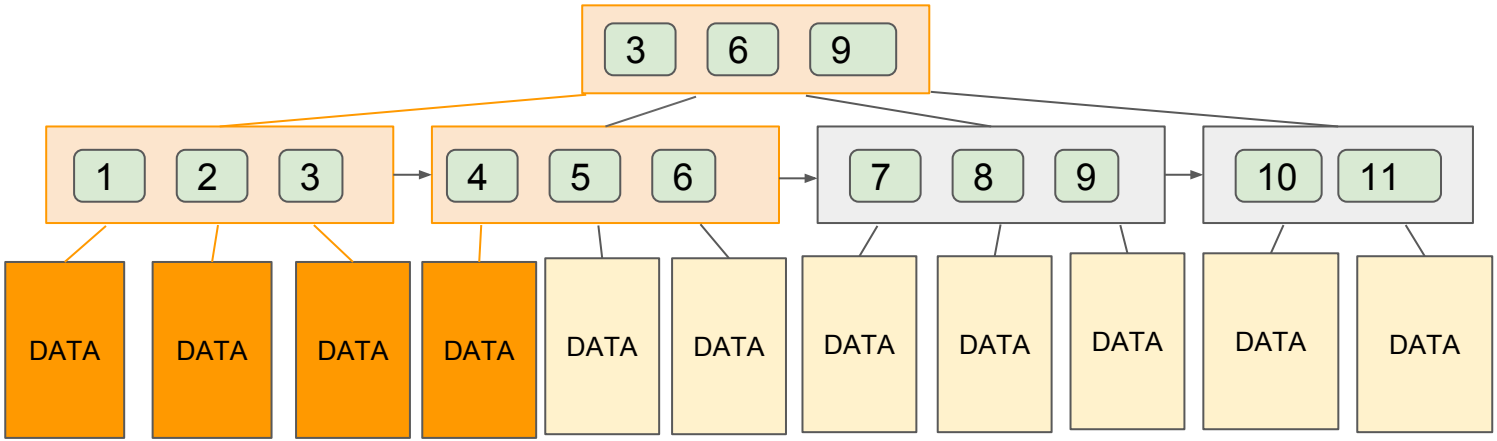
WHERE id = 5


3
6
9
1
2
3
4
5
6
7
8
9
10
11
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

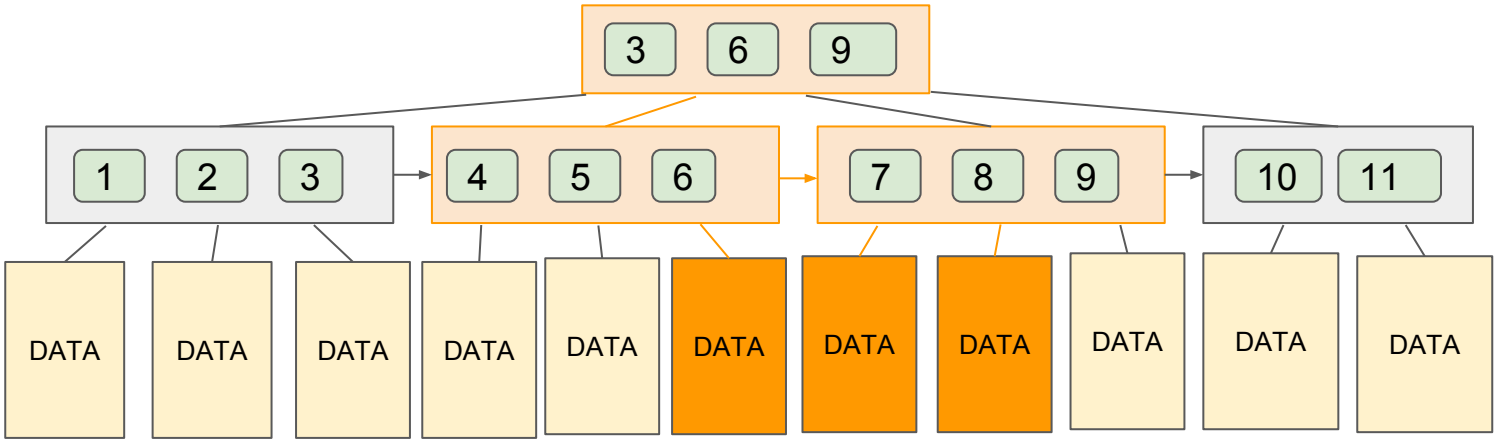
WHERE id > 7


3
6
9
1
2
3
4
5
6
7
8
9
10
11
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

WHERE id < 5


3
6
9
1
2
3
4
5
6
7
8
9
10
11
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

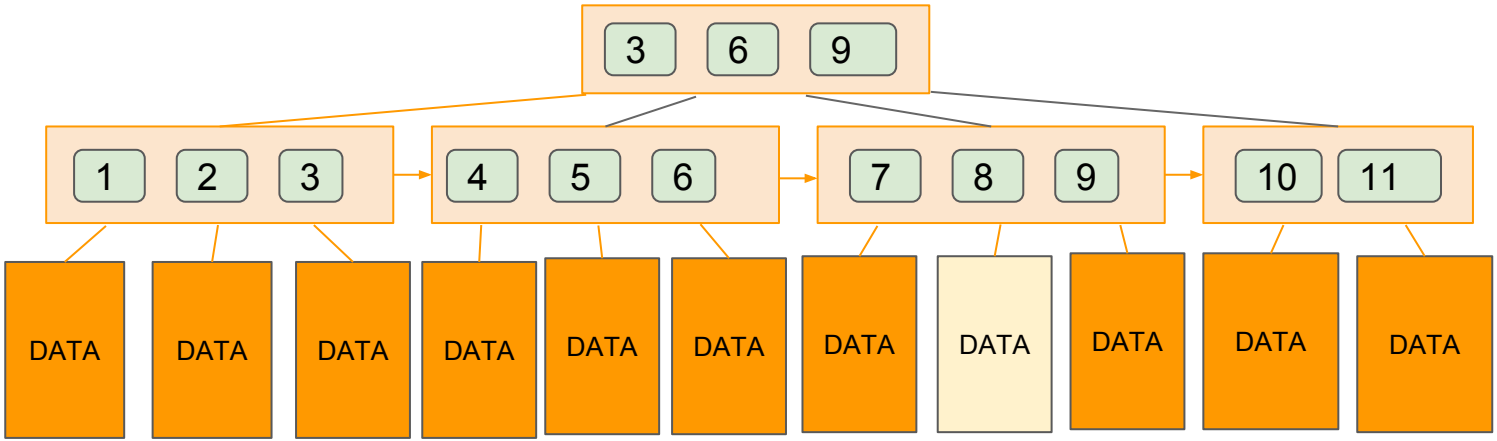
WHERE id BETWEEN 6 AND 8



WHERE id <> 8

BAD

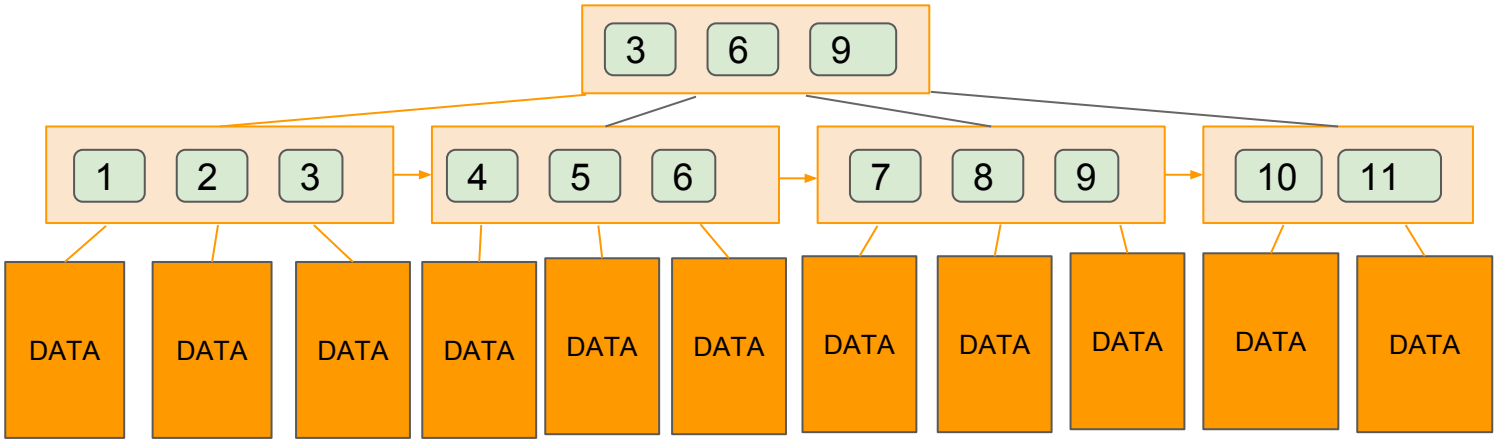
Querying with a negation necessitates table scan



WHERE unindexed_column = ‘uh oh’

BAD

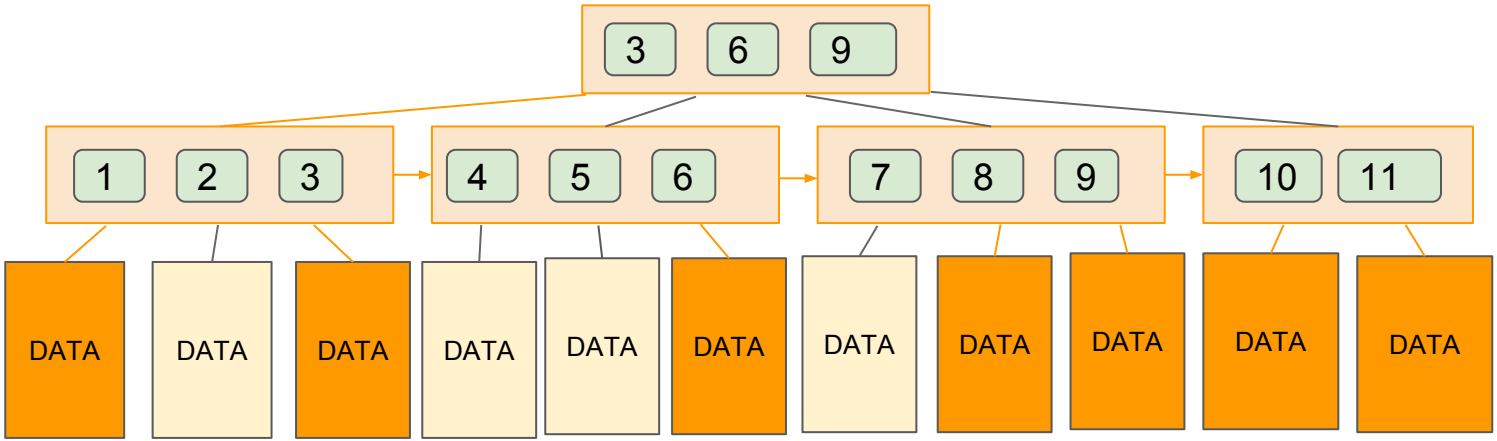
Querying based on a value not indexed results in a table scan



WHERE myFunction(id) > 10

BAD

Querying based on a function output value is also not indexed and requires full index scan



DEMO

Uses dummy database from https://github.com/datacharmer/test_db

You can try these out yourself!

SQL commands in green

Responses and/or times in gray (some parts omitted)

Example:

```
use employees;
Database changed
```

DEMO

Two relevant tables we'll be using:

```
CREATE TABLE `employees` (
 `emp_no` int(11) NOT NULL PRIMARY KEY,
 `birth_date` date NOT NULL,
 `first_name` varchar(14) NOT NULL,
 `last_name` varchar(16) NOT NULL,
 `gender` enum('M','F') NOT NULL,
 `hire_date` date NOT NULL
);
```

Employees - 300,024 rows

Salaries - 2,844,047 rows

```
CREATE TABLE `salaries` (
 `emp_no` int(11) NOT NULL,
 `salary` int(11) NOT NULL,
 `from_date` date NOT NULL,
 `to_date` date NOT NULL,
 PRIMARY KEY (`emp_no`,`from_date`),
 CONSTRAINT `salaries_ibfk_1` FOREIGN KEY   (`emp_no`) REFERENCES `employees` (`emp_no`)
);
```

DEMO

Employees

emp_no	birth_date	first_name	last_name	gender	hire_date
10001	9/2/53	Georgi	Facello	M	6/26/86
10002	6/2/64	Bezalel	Simmel	F	11/21/85
10003	12/3/59	Parto	Bamford	M	8/28/86

Salaries

emp_no	salary	from_date	to_date
10001	85097	6/22/01	6/22/02
10001	88958	6/22/02	1/1/99
10002	65828	8/3/96	8/3/97

DEMO

```
SELECT * FROM employees WHERE emp_no = 20000;
1 row in set (0.01 sec)

SELECT * FROM employees WHERE last_name = 'Matzke' AND first_name = 'Jenwei';
1 row in set (0.10 sec)

SELECT min(emp_no), max(emp_no) FROM salaries;
1 row in set (0.00 sec)

SELECT min(salary), max(salary) FROM salaries;
1 row in set (0.85 sec)

SELECT count(*) FROM salaries WHERE emp_no BETWEEN 20000 AND 25000;
1 row in set (0.03 sec)

SELECT count(*) FROM salaries WHERE salary BETWEEN 70000 AND 75000;
1 row in set (0.59 sec)
```

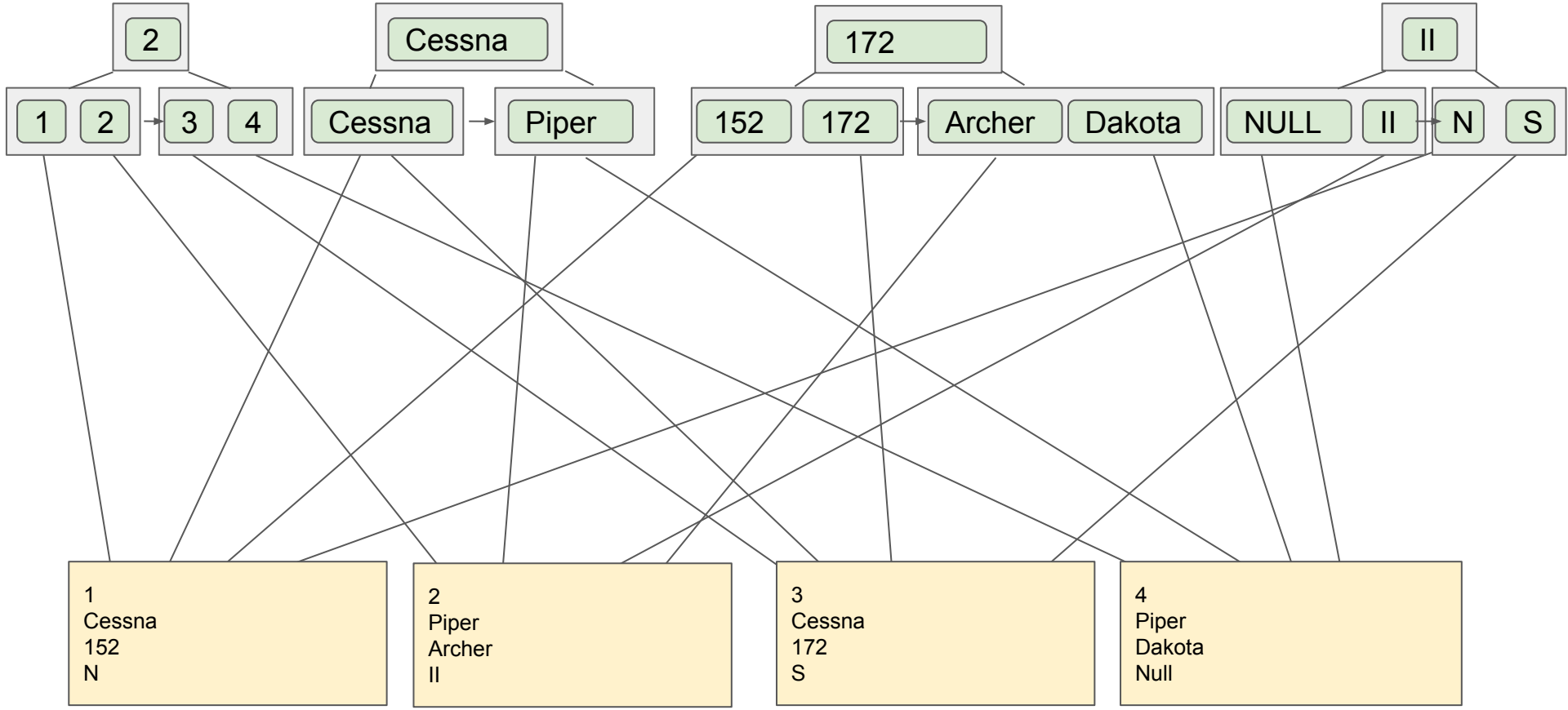
Why not index every column?

```
CREATE INDEX (id)
CREATE INDEX (make)
CREATE INDEX (model)
CREATE INDEX (type)Ω
```

VERY BAD

You will be creating a spread out copy of your table, wasting space and memory.

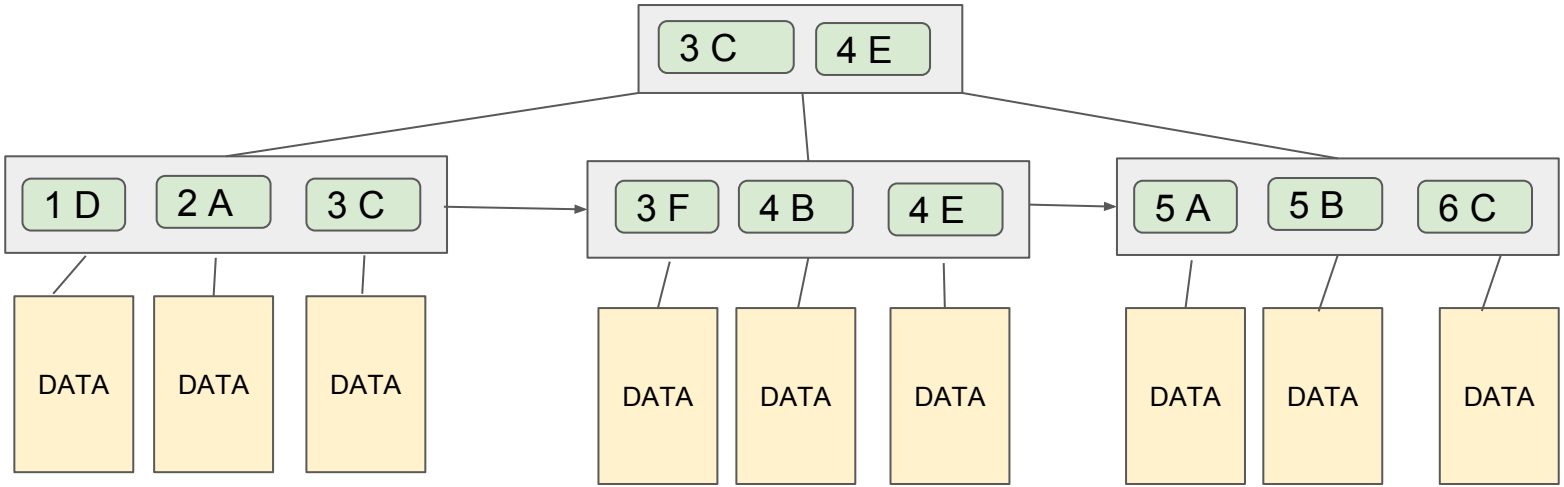
Each write (UPDATE, INSERT, DELETE) will require every index to be updated,
Which makes writing a lot of work...



Note: in reality secondary index leaf nodes are pointers to primary key not the records

Compound Indexes

- Allow sorting based on multiple columns
- Store the data in the index



Subsets of compound Indexes can be used only from left to right

```
CREATE INDEX (num, letter)
```

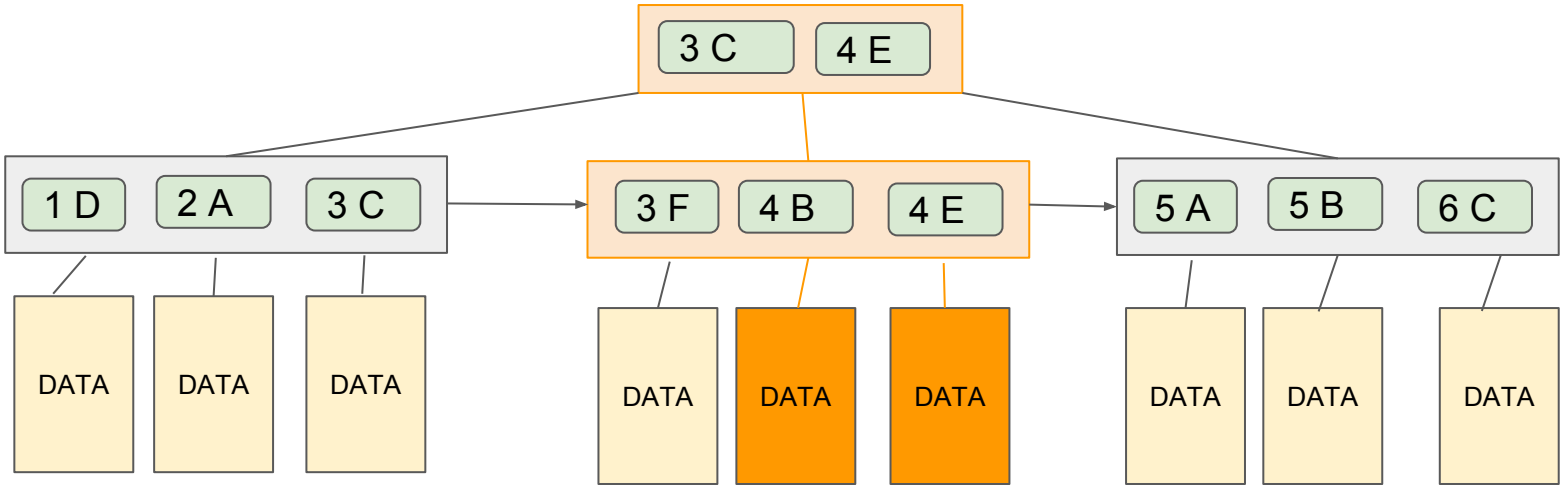
GOOD

```
WHERE num = 4
WHERE num = 5 AND letter < 'C'
```

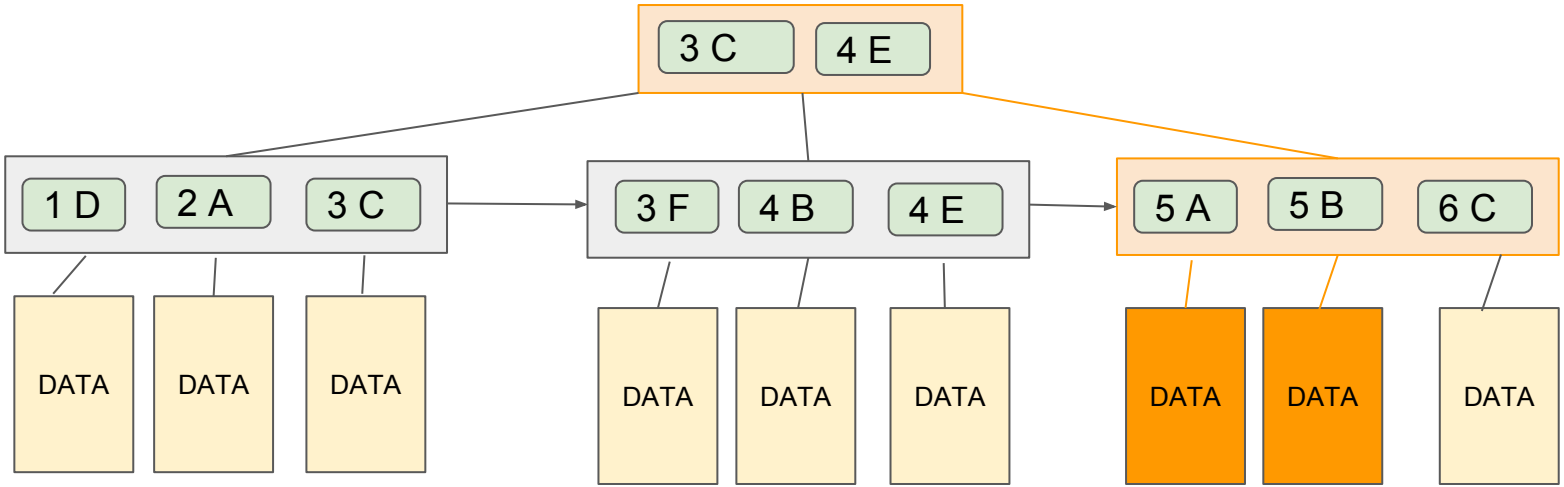
BAD

```
WHERE letter = 'C'
```

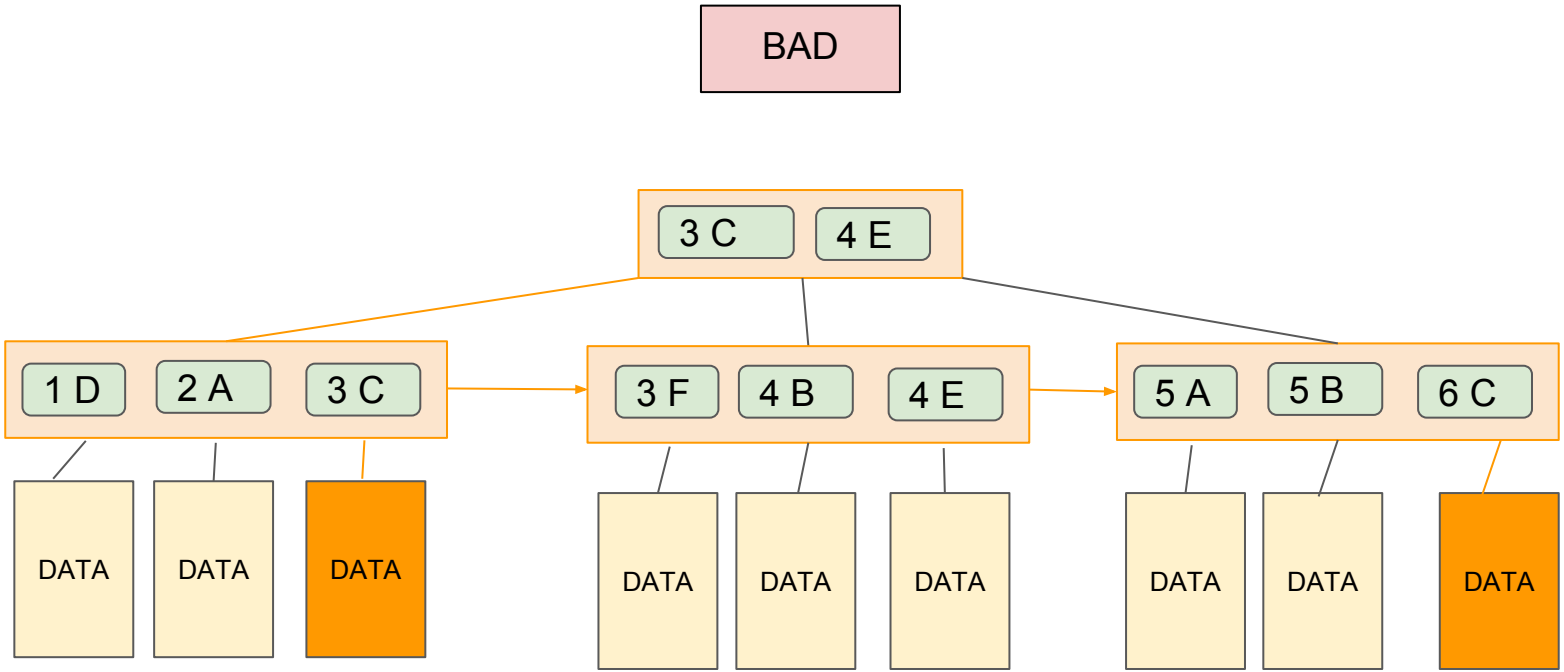
WHERE num = 4


3 C
4 E
1 D
2 A
3 C
3 F
4 B
4 E
5 A
5 B
6 C
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

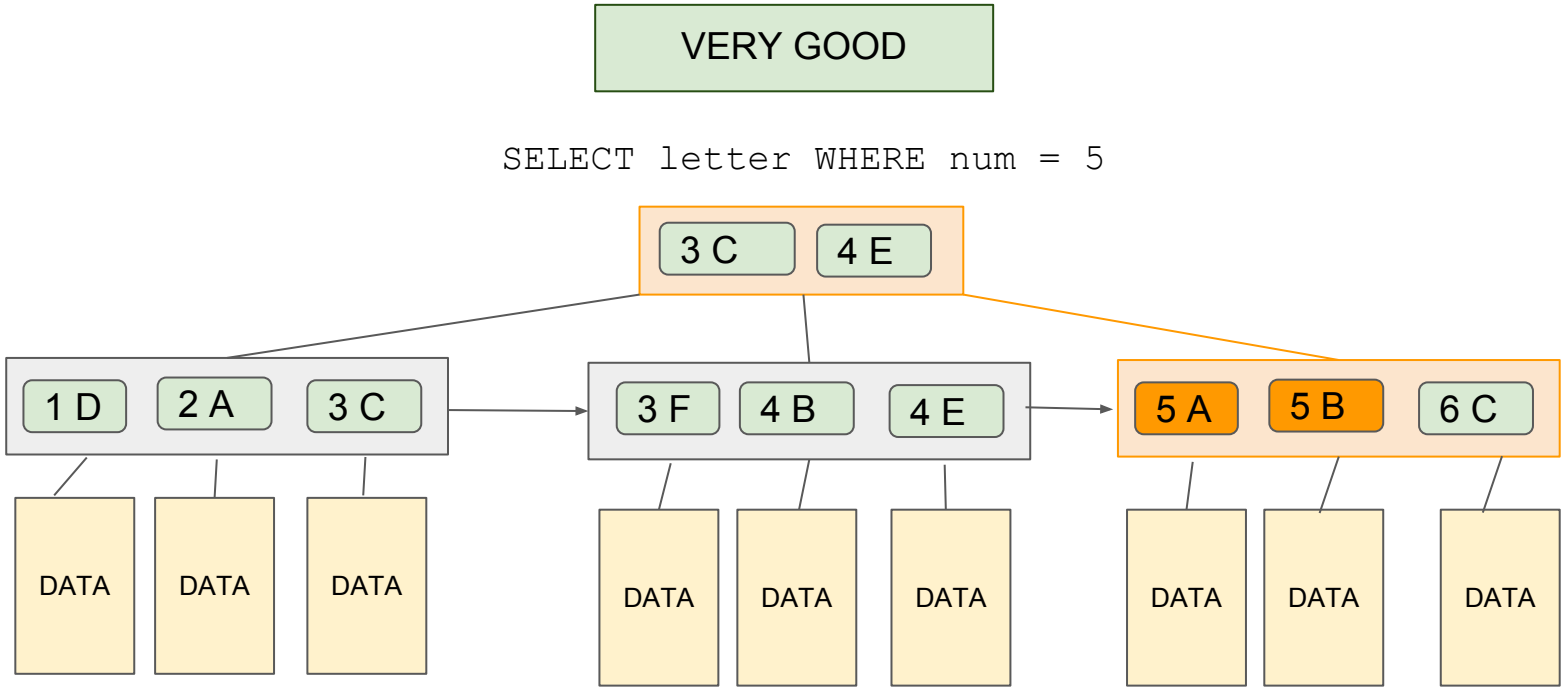
WHERE num = 5 AND letter < ‘C’



WHERE letter = ‘c’



If the data you need is in the index, then you don't even have to fetch the rows! Its a "Covering Index"

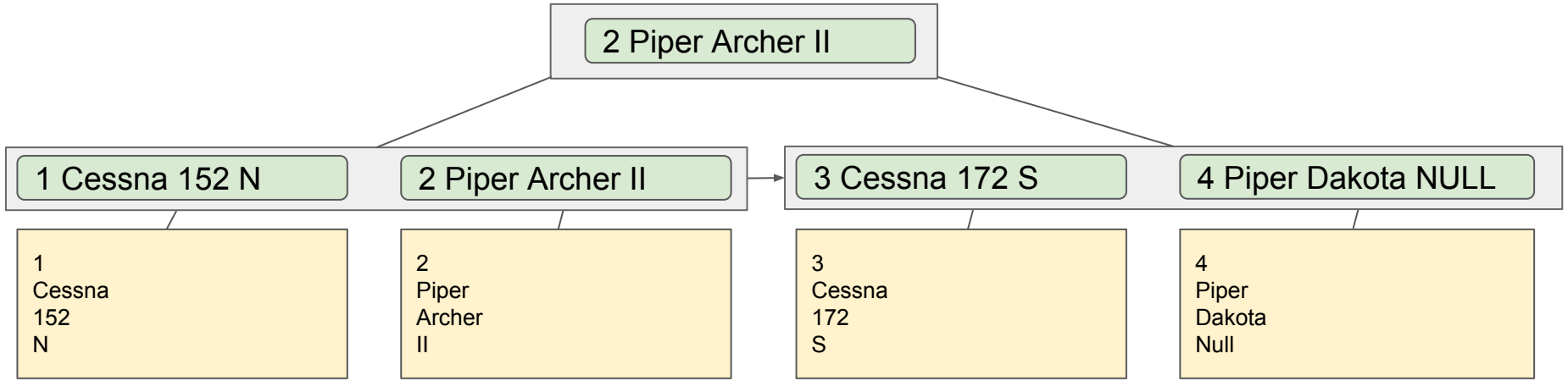


Why not use a compound index to cover your whole table?

```
CREATE INDEX (id, make, model, type)
```

VERY BAD

You will just be creating a more complicated copy of your table. It probably won't fit into memory so MySQL will still have to go to disk.



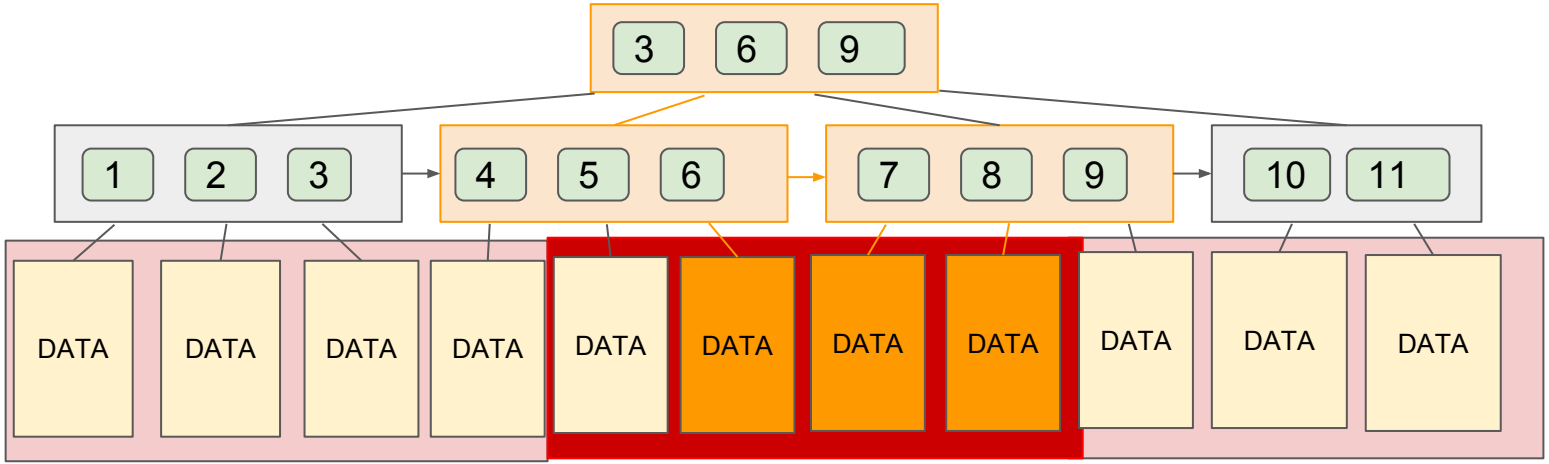
DEMO

```
SELECT max(to_date) FROM salaries WHERE from_date >= '1990-10-01' AND from_date <
'1990-11-01';
1 row in set (0.82 sec)

CREATE INDEX ix_salaries_from_date ON salaries(from_date);
Query OK, 0 rows affected (5.35 sec)

SELECT max(to_date) FROM salaries WHERE from_date >= '1990-10-01' AND from_date <
'1990-11-01';
1 row in set (0.04 sec)

ALTER TABLE salaries DROP INDEX ix_salaries_from_date;
Query OK, 0 rows affected (0.02 sec)
```

DEMO

```
CREATE INDEX ix_salaries_from_date_to_date ON salaries(from_date, to_date DESC);
Query OK, 0 rows affected (5.65 sec)

SELECT max(to_date) FROM salaries WHERE from_date >= '1990-10-01' AND from_date <
'1990-11-01';
1 row in set (0.01 sec)

SELECT max(to_date) FROM salaries WHERE year(from_date) = 1990 AND month(from_date) =
10;
1 row in set (0.54 sec)
```

Clustering places sequential data sequential on the disk, resulting in faster data retrieval as the disk has less places to go to get the data



DEMO

We'll create our own salaries table clustered on a numeric arbitrary id:

```
CREATE TABLE salaries_unclustered (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `emp_no` int(11) NOT NULL,
 `salary` int(11) NOT NULL,
 `from_date` date NOT NULL,
 `to_date` date NOT NULL,
 PRIMARY KEY pk_salaries_unclustered (id),
 UNIQUE KEY uq_salaries_unclustered_emp_from_date(`emp_no`,`from_date`),
 CONSTRAINT `salaries_unclustered_ibfk_1` FOREIGN KEY (`emp_no`) REFERENCES `employees` (`emp_no`)
);
```

```
INSERT INTO salaries_unclustered (emp_no, salary, from_date, to_date)
SELECT emp_no, salary, from_date, to_date
FROM salaries
ORDER BY rand();
```

DEMO

Salaries Unclustered

id	emp_no	salary	from_date	to_date
1	83580	6/3/46	1/7/00	1/6/01
2	478007	3/17/30	9/21/91	9/20/92
3	412998	11/27/15	9/1/99	8/31/00
…	...	...	...	...
396770	83580	4/30/10	1/8/93	1/8/94

DEMO

```
SELECT emp_no, salary, from_date, to_date FROM salaries WHERE emp_no BETWEEN 23456 AND 24467;
9601 rows in set (0.02 sec)

SELECT emp_no, salary, from_date, to_date FROM salaries_unclustered WHERE emp_no BETWEEN
23456 AND 24467;
9601 rows in set (0.05 sec)

UPDATE salaries SET salary = salary*2 WHERE emp_no BETWEEN 23456 AND 24467;
Query OK, 9601 rows affected (0.05 sec)

UPDATE salaries_unclustered SET salary = salary*2 WHERE emp_no BETWEEN 23456 AND 24467;
Query OK, 9601 rows affected (0.12 sec)

UPDATE salaries SET salary = salary/2 WHERE emp_no BETWEEN 23456 AND 24467;
Query OK, 9601 rows affected (0.06 sec)

UPDATE salaries_unclustered SET salary = salary/2 WHERE emp_no BETWEEN 23456 AND 24467;
Query OK, 9601 rows affected (0.10 sec)
```

Additional Index Info

How are Indexes Used?

This could be a whole presentation...

Rules of thumb:

- Indexes help with SELECT, GROUP, and ORDER in that order
- Indexes help with JOIN on the column being joined with
- Indexes only help when cardinality (uniqueness) is high
- Compound indexes should first have columns used by WHERE and then the columns used for ORDER
- Usually better to extend an existing index than create a new one
- Indexes are better when on smaller columns; ideally whole index fits in memory
    - Integers and dates (stored as integers under the hood) good
    - Floats and smaller character columns bad
    - Large character columns very bad

DEMO

```
SELECT count(distinct left(last_name, 1)) / count(*) AS s1,
 count(distinct left(last_name, 2)) / count(*) AS s2,
 count(distinct left(last_name, 3)) / count(*) AS s3,
 count(distinct left(last_name, 4)) / count(*) AS s4,
 count(distinct left(last_name, 5)) / count(*) AS s5,
 count(distinct left(last_name, 6)) / count(*) AS s6,
 count(distinct left(last_name, 7)) / count(*) AS s7,
 count(distinct left(last_name, 8)) / count(*) AS s8,
 count(distinct left(last_name, 9)) / count(*) AS s9,
 count(distinct left(last_name, 10)) / count(*) AS s10,
 count(distinct left(last_name, 11)) / count(*) AS s11,
 count(distinct left(last_name, 12)) / count(*) AS s12,
 count(distinct left(last_name, 13)) / count(*) AS s13,
 count(distinct left(last_name, 14)) / count(*) AS s14,
 count(distinct left(last_name, 15)) / count(*) AS s15,
 count(distinct left(last_name, 16)) / count(*) AS s16
FROM employees;
```

Note: This dummy data had lots of repeat last names. Typical text datasets (names, email) see selectivity approach 1.0

Characters	Selectivity	Characters	Selectivity
1	0.0001	9	0.0055
2	0.0007	10	0.0055
3	0.0028	11	0.0055
4	0.0045	12	0.0055
5	0.0051	13	0.0055
6	0.0054	14	0.0055
7	0.0054	15	0.0055
8	0.0054	16	0.0055

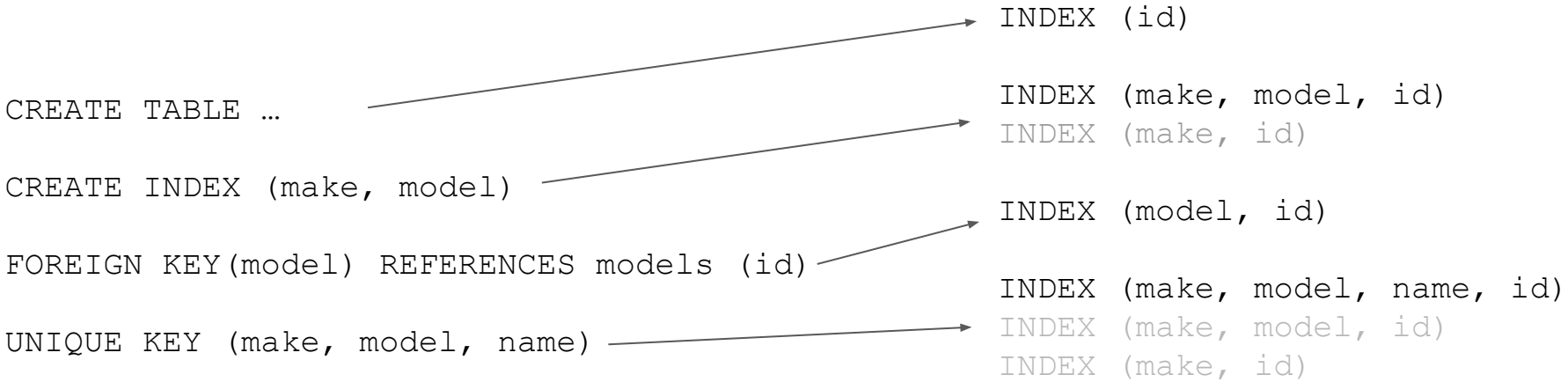
92% efficacy of full index achieved with just 5 characters
100% efficacy of full index achieved with 9 of the characters

DEMO

```
CREATE INDEX ix_employees_last_name ON employees (last_name);
Query OK, 0 rows affected (0.72 sec)

ANALYZE TABLE employees;
1 row in set (0.01 sec)

SELECT stat_value FROM mysql.innodb_index_stats WHERE index_name = 'ix_employees_last_name'
AND stat_name = 'size';
1 row in set (0.01 sec) #result: 417

CREATE INDEX ix_employees_last_name_first5 ON employees (last_name(5));
Query OK, 0 rows affected (1.26 sec)

ANALYZE TABLE employees;
1 row in set (0.01 sec)

SELECT stat_value FROM mysql.innodb_index_stats WHERE index_name =
'ix_employees_last_name_first5' AND stat_name = 'size';
1 row in set (0.01 sec) #result: 353
```

Implicit Indexes

- Primary key is automatically indexed
- Left parts of compound indexes can be used
- All secondary indexes have id appended
- Foreign key is automatically indexed in parent and child tables (if no index present)
- Unique keys create index



Query Optimization

This could also be a whole presentation...

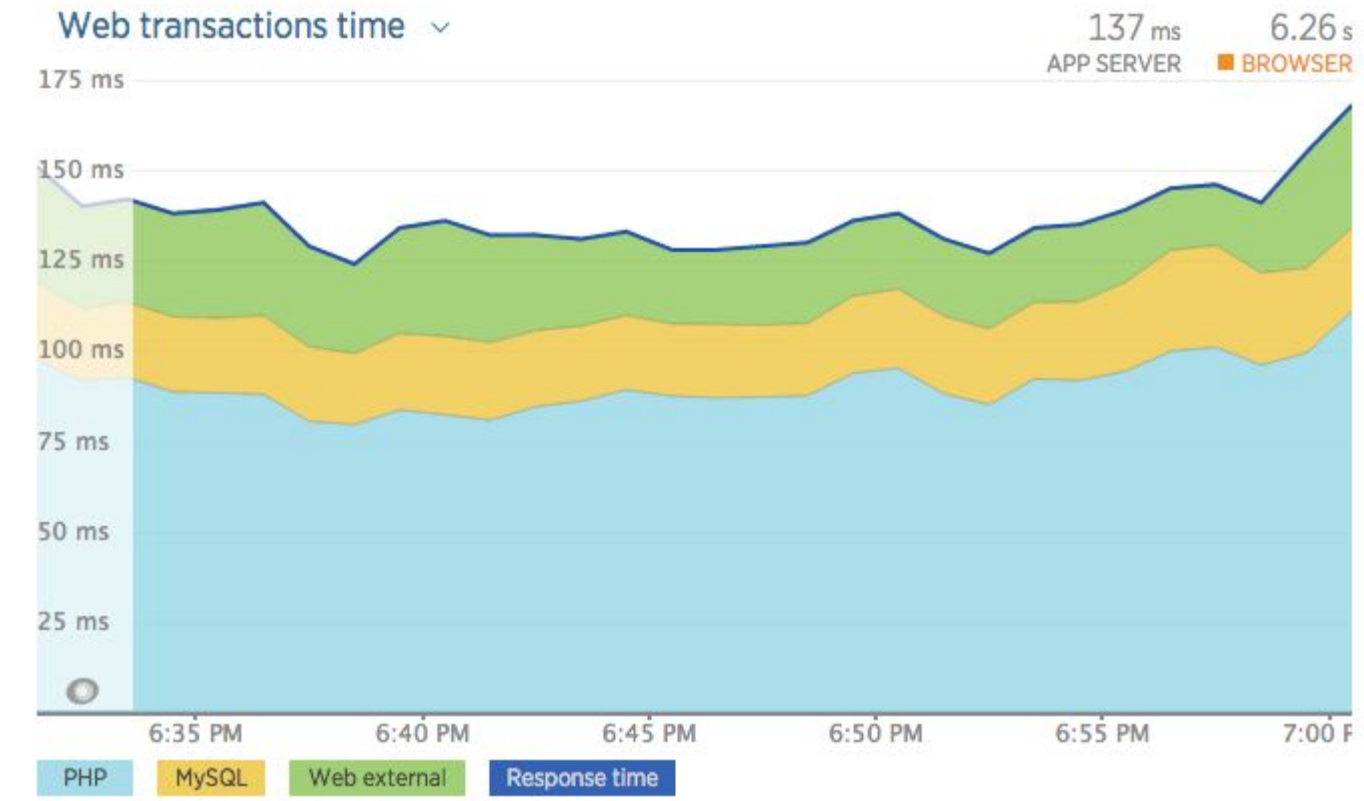
Rules of thumb:

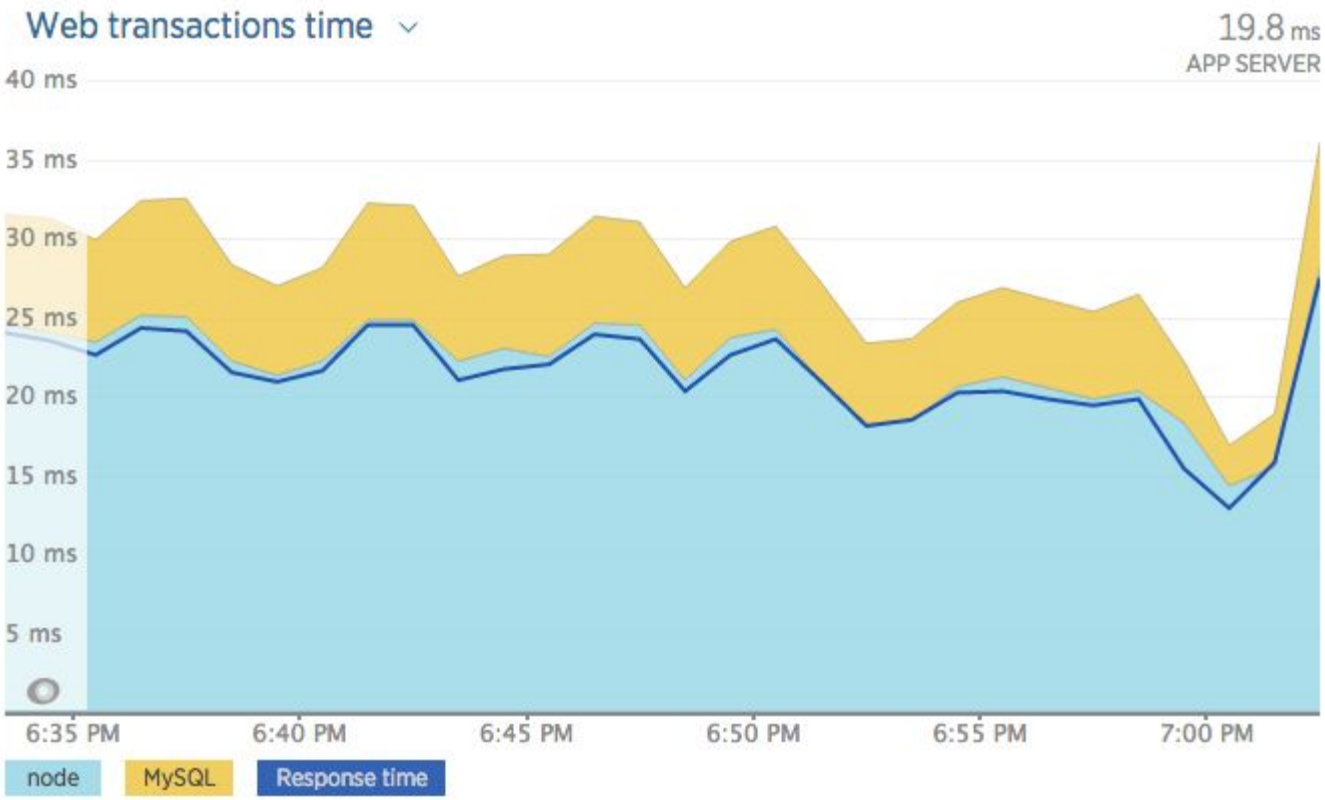
- Create indexes based on how you know you will be querying the table
- Don’t add indexes “just in case”
- If your new query needs a new index, think if you can rewrite it so it does not, or if you can extend an existing index
- Need speed? Select only columns in covering index
- Long (> ~768) varchars, text, and blobs stored off table only select when needed
- Avoid nested subqueries
- Avoid functions in WHERE, GROUP BY, HAVING, and ORDER BY clauses
- Temporary tables (used by subqueries, and UNION) will force disk use if large
- Stuck? Run ANALYZE TABLE and then use EXPLAIN and dive in

DEMO

```
EXPLAIN SELECT * FROM employees WHERE last_name like 'a%';
#Index used

EXPLAIN SELECT * FROM employees WHERE last_name like '%a';
#No indexes available

EXPLAIN SELECT *
FROM employees e
JOIN salaries s
  ON e.emp_no = s.emp_no
WHERE last_name LIKE 'a%';
#Indexed used for each table

EXPLAIN SELECT *
FROM employees e
JOIN salaries s
  ON s.from_date < date_add(e.hire_date, INTERVAL 3 month)
WHERE last_name LIKE 'a%';
#Index only used for first table
```

Appendix


Web transactions time
137 ms
APP SERVER
6.26 s
BROWSER
175 ms
150 ms
125 ms
100 ms
75 ms
50 ms
25 ms
6:35 PM
6:40 PM
6:45 PM
6:50 PM
6:55 PM
7:00 F
PHP
MySQL
Web external
Response time


Web transactions time
19.8 ms
APP SERVER
40 ms
35 ms
30 ms
25 ms
20 ms
15 ms
10 ms
5 ms
6:35 PM
6:40 PM
6:45 PM
6:50 PM
6:55 PM
7:00 PM
node
MySQL
Response time