PERCONA

A Survey of MySQL Index Types

Dave Stokes
@Stoker
David.Stokes@Percona.com
https://speakerdeck.com/stoker

CONFOO 2024 - A Survey of MySQL Indexes

Nobody complains when the database is quick.

But when things slow down you try to speed things up by adding an index.

Later you try a second index and now things are slower than ever.

Indexes can make things faster but you need to know what type of index to use, how to test query performance, and how to maintain them.

This session will cover when and how to add an index plus detail how to test their performance.

PERCONA

A Survey of MySQL Index Types

Dave Stokes
@Stoker
David.Stokes@Percona.com
https://speakerdeck.com/stoker

Who Am I?

Dave Stokes

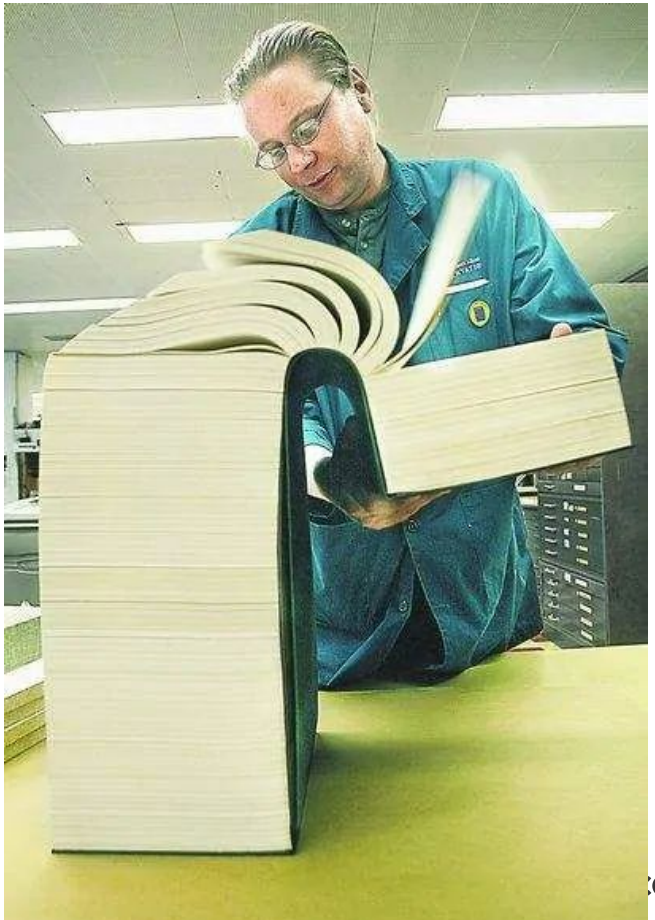
Technology Evangelist at Percona

Author of **MySQL & JSON - A Practical Programming Guide** available on Amazon

Former MySQL AB, Sun Microsystems, InfiniDB, & Oracle

Without an index?

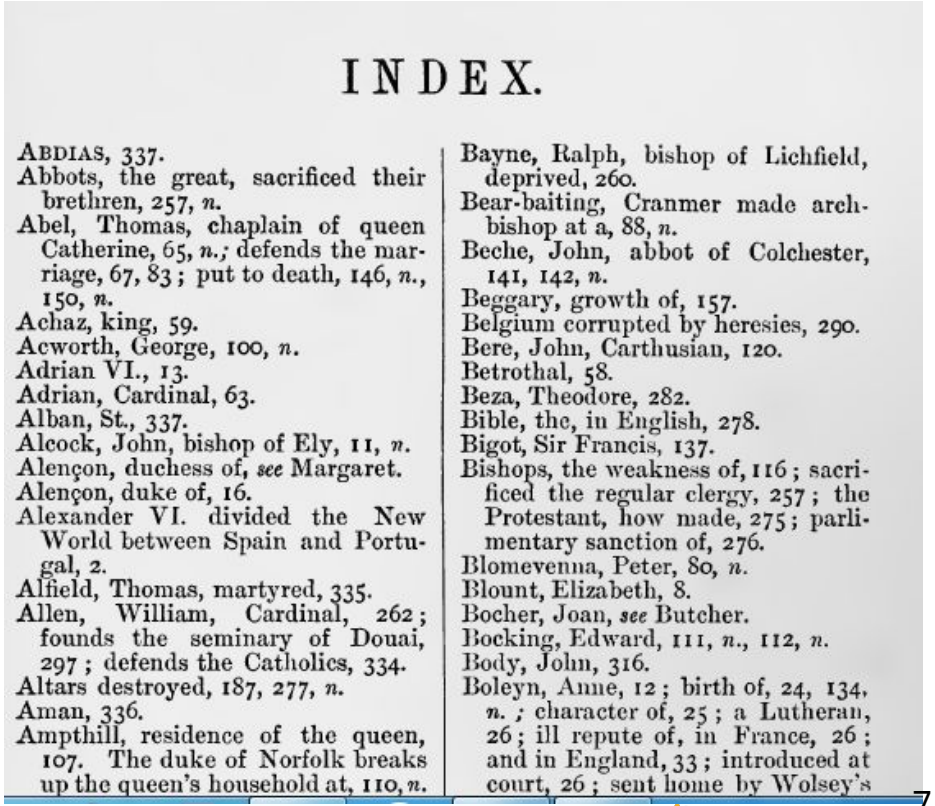
- The entire table (or file) must be read from beginning to end.
- Data may not be ordered.
- Time consuming



InnoDB Indexes

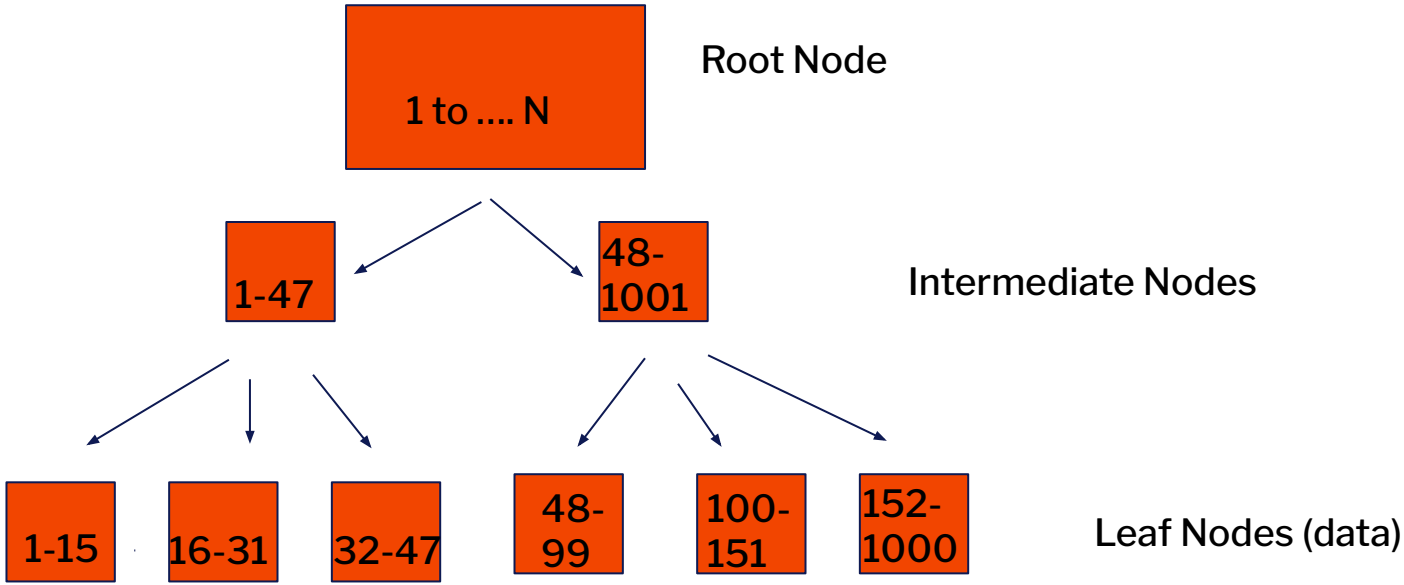
With an index

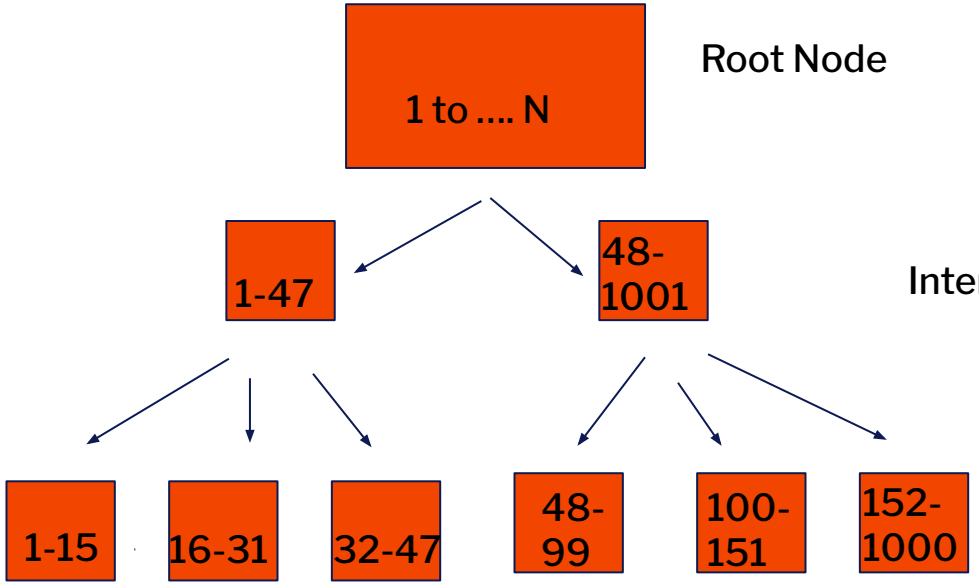
- Gain the ability to go to desired information
- Faster



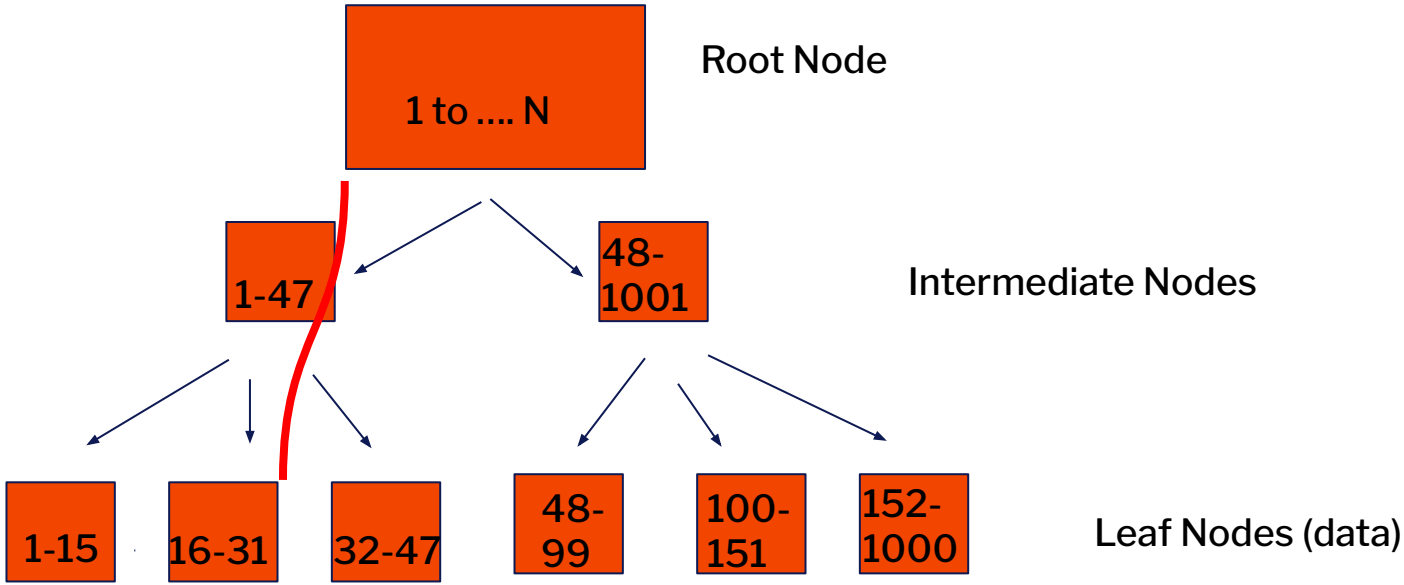
INDEX.

Clustered Index


1 to .... N
Root Node
1-47
48-1001
Intermediate Nodes
1-15
16-31
32-47
48-99
100-151
152-1000
Leaf Nodes (data)

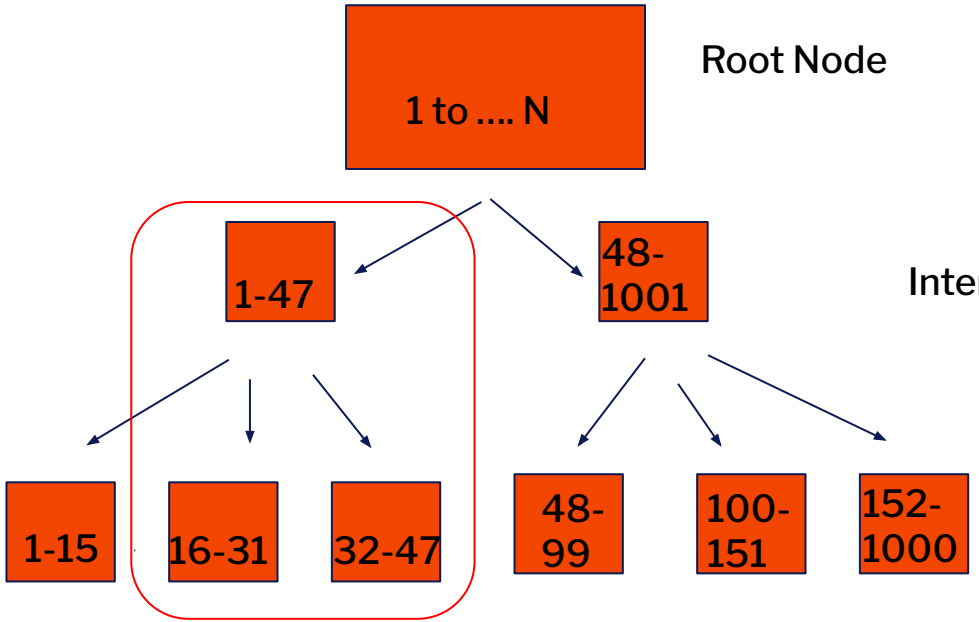


It is very easy with a B+ Tree to search for one record or a range



It is very easy with a B+ Tree to search for one record or a range

Find record 22



It is very easy with a B+ Tree to search for one record or a range

Records between 17 and 42

Records are stored by primary key.

InnoDB stores records by the primary key and will pick one for you if you do not designate one.

And the one it picks will not be optimal.

Please pick your OWN primary key!!

Primary key 001	Data for 001
Primary key 002	Data for 002
Primary key 003	Data for 003
Primary Key ….	

Primary Key vs Secondary

1. Try to have a **unique** primary key **not nullable** for each row
2. Secondary key are non-primary keys that are used to access records in other tables, **not nullable**
3. Keys are what make up an index
4. Avoid nulls in indexes
5. No GUID PRIMARY Keys,
   maybe secondary

Creating Indexes

Indexes are easy to create but remember they

1. Take up space
2. Add overhead for maintenance – INSERT/UPDATE/DELETE

A Quick Demo

```
CREATE SCHEMA pldemo;
USE pldemo;

CREATE TABLE ex01 (my_id SERIAL PRIMARY KEY, c1 INT, c2 INT);


SHOW CREATE TABLE ex01;

ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

```
ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

```
ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

Bigint unsigned provides a range of 0 to 2^^63

```
ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

NOT NULL no null values in column!

```
ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

```
ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

But you end up with TWO indexes and their overhead — and auto increments assures uniqueness so skip the UNIQUE KEY!

```
ex01, CREATE TABLE `ex01` (
  `my_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_id`),
  UNIQUE KEY `my_id` (`my_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

But you end up with TWO indexes and their overhead — and auto increments assures uniqueness so skip the UNIQUE KEY!

But this is valid in the PostgreSQL world where you can re-index one of the indexes and the system will switch over to the better index automatically!

BIG RECOMMENDATION

Whenever possible have your primary key be:

- INT/BIGIT
- UNSIGNED
- NOT NULL
- AUTO_INCREMENT

Insert 100 records

```
INSERT INTO ex01 (my_id, c1, c2) values (NULL, rand(), RAND());
```

```
EXPLAIN select * from ex01 where my_id = 7;
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | ex01  | NULL       | const | PRIMARY,my_id | PRIMARY | 8       | const |    1 |      100 | NULL  |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.0010 sec)
Note (code 1003): /* select#1 */ select '7' AS `my_id`,'1' AS `c1`,'0' AS `c2` from `pldemo`.`ex01` where true
```

We'll use the 'bad' KEYs from the previous example to show multiple possible keys

For comparison

```
create table ex02 (bad_id int unsigned not null, c1 int, c2 int);

SHOW CREATE TABLE ex02;
'ex02', 'CREATE TABLE `ex02` (
  `bad_id` int unsigned NOT NULL,
  `c1` int DEFAULT NULL,
  `c2` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci'
```

(insert records)

```
SQL > explain format=tree select * from ex02 where bad_id = 7;
+-----------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                       |
+-----------------------------------------------------------------------------------------------+
| -> Filter: (ex02.bad_id = 7)  (cost=10.25 rows=1)
    -> Table scan on ex02  (cost=10.25 rows=10)
 |
+-----------------------------------------------------------------------------------------------+
1 row in set (0.0007 sec)
```

NO INDEX

```
SQL > explain format=tree select * from ex01 where my_id = 7;
+-------------------------------------------------------------+
| EXPLAIN                                                     |
+-------------------------------------------------------------+
| -> Rows fetched before execution  (cost=0.00..0.00 rows=1)
 |
+-------------------------------------------------------------+
1 row in set (0.0009 sec)
```

WITH INDEX

```
{
 "query_block": {
   "select_id": 1,
   "cost_info": {
     "query_cost": " 10.25"
   },
   "table": {
     "table_name": "ex02",
     "access_type": "ALL",
     "rows_examined_per_scan": 100,
     "rows_produced_per_join": 10,
     "filtered": "10.00",
     "cost_info": {
       "read_cost": "9.15",
       "eval_cost": "1.00",
       "prefix_cost": "10.25",
       "data_read_per_join": "16"
     },
     "used_columns": [
       "bad_id",
       "c1",
       "c2"
     ],
     "attached_condition":
"(`pldemo`.`ex02`.`bad_id` = 7)"
   }
  }
}
```

```
{
 "query_block": {
  "select_id": 1,
  "cost_info": {
   "query_cost": "1.00"
  },
  "table": {
   "table_name": "ex01",
   "access_type": "const",
   "possible_keys": [
    "PRIMARY",
    "my_id"
   ],
   "key": "PRIMARY",
   "used_key_parts": [
    "my_id"
   ],
   "key_length": "8",
   "ref": [
    "const"
   ],
   "rows_examined_per_scan": 1,
   "rows_produced_per_join": 1,
   "filtered": "100.00",
   "cost_info": {
    "read_cost": "0.00",
    "eval_cost": "0.10",
    "prefix_cost": "0.00",
    "data_read_per_join": "24"
   },
   "used_columns": [
    "my_id",
    "c1",
    "c2"
   ]
  }
 }
}
```

So, Indexes do work!

Can we just index everything?

PERCONA

NO!

Details will come later

PERCONA

What can I Index?

A whole lot!

ALTER TABLE too

ALTER TABLE can used to create or alter an index

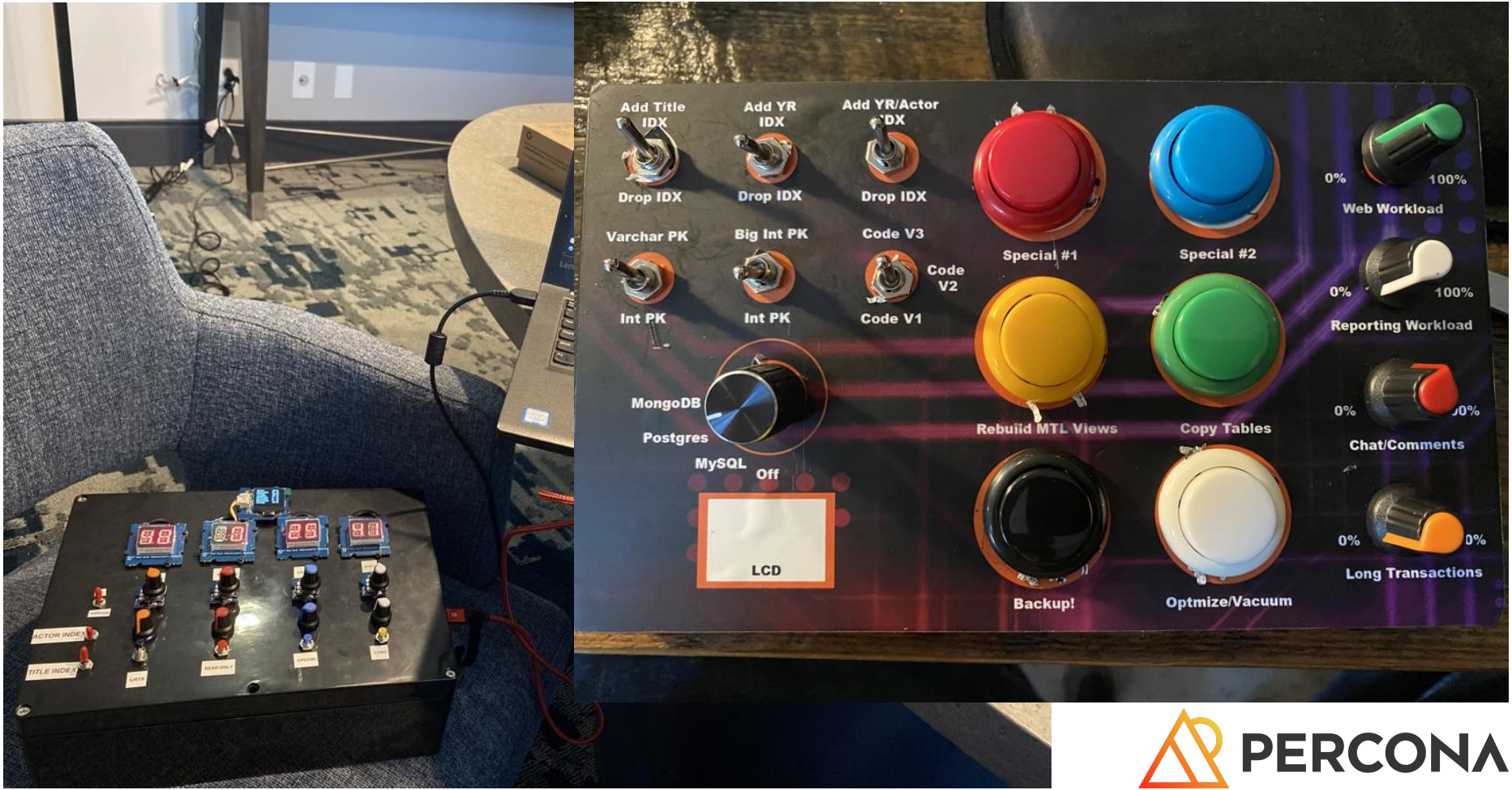
Yonk Box LLC

Congratulations! You are now an employee of Yonk Box LLC and your first task is to create a table for orders of Yonk Boxes.

PERCONA

Covering Index

Get answers from just the index!


Add Title IDX
Add YR IDX
Add YR/Actor IDX
Drop IDX
Drop IDX
Drop IDX
Special #1
Special #2
0%
100%
Web Workload
Varchar PK
Big Int PK
Code V3
Code V2
Int PK
Int PK
Code V1
0%
100%
Reporting Workload
MongoDB
Postgres
MySQL
Off
Rebuild MTL Views
Copy Tables
0%
0%
Chat/Comments
LCD
Backup!
Optmize/Vacuum
0%
0%
Long Transactions

PERCONA

```
CREATE TABLE customer_order (
        order_id SERIAL PRIMARY KEY,
        customer_id INT UNSIGNED NOT NULL,
        order_date DATE,
        est_delivery DATE,
        item_nbr INT UNSIGNED NOT NULL,
        worker SMALLINT UNSIGNED);
```

We know each Yonk Box has a serial number, a customer, a date when ordered, a estimated delivery date, an item number (delux Yonk, platinum edition, etc.), and maybe a worker assigned to the project.

What if we want to quickly search by customer_id?

(this is why you need to know how you will *use* your data)

```
CREATE INDEX
    customer_order_customer_idx
ON
    customer_order (customer_id);
```

Yes, customer_order_customer_idx is a long name but not confusing.

What if we want to quickly search by the first five numbers of the customer_id?

```
CREATE INDEX
    customer_order_customer_idx
ON
    customer_order (customer_id:5);
```

What about location?

```
CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);
ALTER TABLE geom ADD SPATIAL INDEX(g);
```

MySQL can index spatial data

Key words?

```
ALTER TABLE customer_order
    ADD COLUMN description VARCHAR(500),
    ADD FULLTEXT desc_idx (description);
```

Here we have made two modification to the table. First we add a *description* column and then create a FULLTEXT index on that new column.

```
ALTER TABLE customer_order
    ADD COLUMN description VARCHAR(500),
    ADD FULLTEXT desc_idx (description);
```

```
SELECT   order_id,
         customer_id
FROM customer_order
WHERE MATCH(description) AGAINST ('Gold');
+----------+-------------+
| order_id | customer_id |
+----------+-------------+
|     1357 |          42 |
+----------+-------------+
1 row in set (0.0009 sec)
```

Functional Indexes

A **functional index** is **defined** on the result of a **function** applied to one or more columns of a single table

ALTER TABLE customer_order ADD INDEX est_month_idx((month(est_delivery)));

explain format=tree select month(est_delivery) from customer_order where month(est_delivery) = 5;

```
+-----------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                             |
+-----------------------------------------------------------------------------------------------------+
| -> Index lookup on customer_order using est_month_idx (month(est_delivery)=5)  (cost=0.35 rows=1)
 |
+-----------------------------------------------------------------------------------------------------+
1 row in set (0.0010 sec)
```

ALTER TABLE product ADD INDEX total_production_cost_idx ((cost_of_good_sold *1.5));

explain format=tree select id as 'item', cost_of_good_sold as 'cost to product' from product where cost_of_good_sold * 1.5 > 10.0\G

```
*************************** 1. row ***************************
EXPLAIN: -> Filter: ((cost_of_good_sold * 1.5) > 10.0)  (cost=1.16 rows=2)
    -> Index range scan on product using total_production_cost_idx over (10.000 < (`cost_of_good_sold` *
1.5))  (cost=1.16 rows=2)
```

Multi-Value Indexes

A Multi-Valued Index (MVI) is a secondary index defined on a JSON column made up of an array of values.

Traditionally indexes where you have one value per index entry, a 1:1 ratio.

A MVI can have multiple records for *each* index record.

```
select * from customers;
+----+-------+-----------------------------------+
| id | name  | info                              |
+----+-------+-----------------------------------+
| 12 | Fred  | {"zipcode": [12345, 78901]}       |
| 12 | Matt  | {"zipcode": [22221, 64263, 11111]} |
| 15 | Kenny | {"zipcode": [12345]}              |
| 15 | Peter | {"zipcode": [54321, 65432]}       |
+----+-------+-----------------------------------+
```

```
SELECT id, name
FROM customers
WHERE 12345
MEMBER OF (info->"$.zipcode");
+----+--------+
| id | name   |
+----+--------+
| 15 | Kenny  |
| 12 | Fred   |
+----+--------+
2 rows in set (0.0009 sec)
```

```
create table customers (
    id int,
    name varchar(20),
    info JSON,
    INDEX zidx
      ((cast(info->'$.zipcode' AS UNSIGNED ARRAY)))
    );
```

Multi Column

You can index more than one column in a row!

Put highest cardinality/rarest in the left most column (and repeat until done)

```
create table x (c1 int, c2 int, c3 int, c4 int);
insert into x values
(1,2,3,100),(1,2,4,101),(2,2,2,102),(4,1,1,104),(5,6,2,109);
create index x_index on x (c1,c2,c3);
```

```
explain format=tree select * from x where  c1=1 and c2=2 and c3=3;
+-------------------------------------------------------------------------------+
| EXPLAIN                                                                       |
+-------------------------------------------------------------------------------+
| -> Index lookup on x using x_index (c1=1, c2=2, c3=3)  (cost=0.35 rows=1)  |
+-------------------------------------------------------------------------------+
```

create table x (c1 int, c2 int, c3 int, c4 int);
insert into x values (1,2,3,100),(1,2,4,101),(2,2,2,102),(4,1,1,104),(5,6,2,109);
create index x_index on x (c1,c2,c3);

```
explain format=tree select * from x where  c1=1 and c2=2;
+-----------------------------------------------------------------------------+
| EXPLAIN                                                                     |
+-----------------------------------------------------------------------------+
| -> Index lookup on x using x_index (c1=1, c2=2)  (cost=0.70 rows=2)
 |
+-----------------------------------------------------------------------------+
```

create table x (c1 int, c2 int, c3 int, c4 int);
insert into x values (1,2,3,100),(1,2,4,101),(2,2,2,102),(4,1,1,104),(5,6,2,109);
create index x_index on x (c1,c2,c3);

```
explain format=tree select * from x where  c1=1 and c3=3;
+-------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                         |
+-------------------------------------------------------------------------------------------------+
| -> Index lookup on x using x_index (c1=1), with index condition: (x.c3 = 3)  (cost=0.54 rows=2)  |
+-------------------------------------------------------------------------------------------------+
```

```
create table x (c1 int, c2 int, c3 int, c4 int);
insert into x values (1,2,3,100),(1,2,4,101),(2,2,2,102),(4,1,1,104),(5,6,2,109);
create index x_index on x (c1,c2,c3);
```

```
explain format=tree select * from x where c2=2 and c3=3;
+-------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                           |
+-------------------------------------------------------------------------------------------------------------------+
| -> Filter: ((x.c3 = 3) and (x.c2 = 2))  (cost=0.75 rows=1)
    -> Table scan on x  (cost=0.75 rows=5)|
+-------------------------------------------------------------------------------------------------------------------+
```

create table x (c1 int, c2 int, c3 int, c4 int);
insert into x values **(1,2,3,100)**,(1,2,4,101),(2,2,2,102),(4,1,1,104),(5,6,2,109);
create index x_index on x **(c1,c2,c3)**;

```
explain format=tree select c1,c2 from x where c1=1 and c2=2 and c3=3;
+-------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                   |
+-------------------------------------------------------------------------------------------+
| -> Covering index lookup on x using x_index (c1=1, c2=2, c3=3)  (cost=0.35 rows=1)  |
+-------------------------------------------------------------------------------------------+
```

Foreign Keys

MySQL supports foreign keys to cross-referencing related data across tables and foreign key constraints, which help keep the related data consistent.

A foreign key relationship involves a parent table that holds the initial column values, and a child table with column values that reference the parent column values. A foreign key constraint is defined on the child table.

```
CREATE TABLE parent (
    id INT NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE child (
    id INT,
    parent_id INT,
    INDEX par_ind (parent_id),
    FOREIGN KEY (parent id)
        REFERENCES parent(id)
        ON DELETE CASCADE
);
```

Hash Joins

Hash join is a way of executing a join where a hash table is used to find matching rows between the two tables.

EQUALITIES only X = Y

It is typically more efficient than nested loop joins, especially if one of the inputs can fit in memory.

PERCONA

explain format=tree select a.c1, b.c2, c.c1 from a join b on (a.c2=b.c2) join c on (a.c2=c.c1)\G

*************************** 1. row ***************************

EXPLAIN: -> Inner hash join (c.c1 = a.c2) (cost=4.35 rows=4)

    -> Table scan on c (cost=0.09 rows=4)

    -> Hash

        -> Inner hash join (b.c2 = a.c2) (cost=2.50 rows=4)

            -> Table scan on b (cost=0.09 rows=4)

            -> Hash

                -> Table scan on a (cost=0.65 rows=4)

MySQL employs a hash join for any query for which each join has an equi-join condition, and in which there are no indexes that can be applied to any join conditions

Invisible Indexes

MySQL supports invisible indexes.

An invisible index is not seen by the query optimizer.

The feature applies to indexes other than primary keys.

Much easier than removing an index for testing and then having to rebuild.

PERCONA

```
$EXPLAIN format=tree select count(CountryCode) from City where District='Texas'\G
*************************** 1. row ***************************
EXPLAIN: -> Aggregate: count(city.CountryCode)  (cost=484.12 rows=419)
    -> Filter: (city.District = 'Texas')  (cost=442.24 rows=419)
        -> Table scan on City  (cost=442.24 rows=4188)


$ALTER TABLE City ADD INDEX district_idx (District);
Query OK, 0 rows affected (0.1120 sec)


$EXPLAIN format=tree select count(CountryCode) from City where District='Texas'\G
*************************** 1. row ***************************
EXPLAIN: -> Aggregate: count(city.CountryCode)  (cost=11.70 rows=26)
    -> Index lookup on City using district_idx (District='Texas')  (cost=9.10 rows=26)

1 row in set (0.0006 sec)
$ ALTER TABLE City ALTER INDEX district_idx INVISIBLE;


$ EXPLAIN format=tree select count(CountryCode) from City where District='Texas'\G
*************************** 1. row ***************************
EXPLAIN: -> Aggregate: count(city.CountryCode)  (cost=425.36 rows=3)
    -> Filter: (city.District = 'Texas')  (cost=425.05 rows=3)
        -> Table scan on City  (cost=425.05 rows=4188)
```

Histograms?

Instead of indexing, you may want a histogram!

Great for data with low ‘churn rate’

Optimizer ‘assumes’ even distribution of data within a column – a very rare occurrence

```
$create table h1 (id int unsigned auto_increment,
            x int unsigned,
            primary key(id));

$ insert into h1 (x) values
(1),(2),(2),(3),(3),(3),(4),(4),(4),(4),(17) ;

$explain SELECT x FROm h1 WHERE x> 0\G
*************************** 1. row
***************************
           id: 1
  select_type: SIMPLE
        table: h1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 11
     filtered: 33.32999801635742
        Extra: Using where
```

Estimate from optimizer -
Will need to read 33.33 of the 11 rows
To find all rows with x > 0

All rows have values of x > 0

Not a great estimate

```
$ ANALYZE TABLE h1 UPDATE HISTOGRAM ON x WITH 10 BUCKETS;
+----------+-----------+----------+-----------------------------------------+
| Table    | Op        | Msg_type | Msg_text                                |
+----------+-----------+----------+-----------------------------------------+
| world.h1 | histogram | status   | Histogram statistics created for column 'x'. |
+----------+-----------+----------+-----------------------------------------+

 $explain SELECT x FROm h1 WHERE x> 0\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: h1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 11
     filtered: 100
        Extra: Using where
```

Better Estimate

Index or Histogram?

Index

- Fast
- Requires maintenance
- Index Dive by Server
- Take up space on disk and in memory

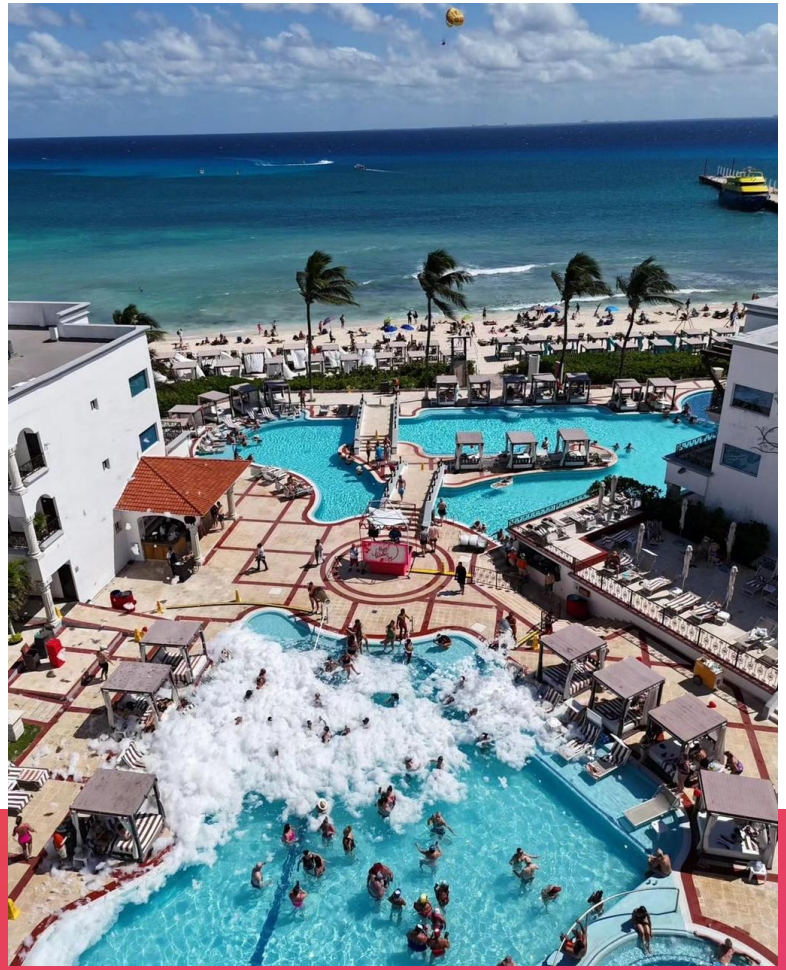
Histogram

- Fast-ish
- Requires maintenance as data ages
- Not for rapidly changing data

PERCONA

Whew!

That Was A lot To Cover



Thank you!

https://speakerdeck.com/stoker @Stoker

PERCONA