MySQL Weirdness

+ some tips (part 1)

Richard Baker / @r_bake_r

Indexes

Indices? ¯_(ツ)_/¯

Example Table

developers

id	name	email	dob
1	Steve	steve@3sidedcube.com	20/01/1986
2	Ben	ben@3sidedcube.com	30/04/1989
3	Chris	chris@3sidedcube.com	15/07/1989
4	Kev	kev@3sidedcube.com	30/04/1989
5	Rich	richard@3sidedcube.com	22/07/1986
6	Charlton	charlton@3sidedcube.com	01/01/2001

*ages changed to protect the aged

```
CREATE TABLE `developers` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `dob` date DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

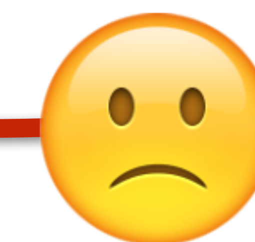
```
> SELECT * FROM developers WHERE name = 'Rich'

5 |  Rich | richard@3sidedcube.com | 1986-07-22
```

EXPLAIN

```
> EXPLAIN SELECT *
  FROM developers
  WHERE name = 'Rich'
```

id	select_type	table	partitions	type	possible_keys	key	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	ALL	NULL	NULL	NULL	6	16.67	Using where



No index to use

Filter applied to all rows (table scan)

Single Index

Add a **single** index on **each** column

```
> CREATE INDEX idx_name ON developers (name)
> CREATE INDEX idx_email ON developers (email)
> CREATE INDEX idx_dob ON developers (dob)

> SHOW INDEX FROM developers
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
developers	0	PRIMARY	1	id	6
developers	1	idx_name	1	name	6
developers	1	idx_email	1	email	6
developers	1	idx_dob	1	dob	3

Cardinality = Uniqueness of values contained by the column

Single Index

```
> EXPLAIN SELECT *
  FROM developers
  WHERE name = 'Rich'
```

Single WHERE condition

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	ref	idx_name	idx_name	768	const	1	100.00	

idx_name is used

Points to the row we need (100%)

Single Index

```
> EXPLAIN SELECT *
  FROM developers
  WHERE dob < '2001-01-01'
  AND name = 'Rich'
```

Multiple WHERE conditions

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	ref	idx_name,idx_dob	idx_name	768	const	1	83.33	Using where

Optimiser chooses idx_name due to **greater cardinality** (6 > 3)

Filter is applied to remaining rows (1) idx_dob is **not** used

Compound Index

Add a **single** index including **multiple** columns

```
> CREATE INDEX idx_dob_name ON developers (dob, name)

> SHOW INDEX FROM developers
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
developers	0	PRIMARY	1	id	6
developers	1	idx_dob_name	1	dob	5
developers	1	idx_dob_name	2	name	6

Compound Index

```
> EXPLAIN SELECT *
  FROM developers
  WHERE dob < '2001-01-01'
  AND name = 'Rich'
```

Much better….

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	ALL	idx_dob_name	NULL	NULL	NULL	6	16.67	Using where

Nope.

WHERE conditions are poorly ordered.
"WHERE dob <" range as first condition prevents optimiser using index
Optimiser chooses a full table scan over our index

Compound Index

```
> EXPLAIN SELECT *
  FROM developers
  FORCE INDEX (idx_dob_email)
  WHERE dob < '2001-01-01'
  AND name = 'Rich'
```

Dirty (?) hack if you need a quick fix

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	range	idx_dob_name	idx_dob_name	4	NULL	5	16.67	Using index condition

Better, but: poor index order requires scan of remaining rows (5) returned by range condition

Compound Index

Improve the index

```
> CREATE INDEX idx_name_dob
  ON developers (name, dob)

> SHOW INDEX FROM developers
```

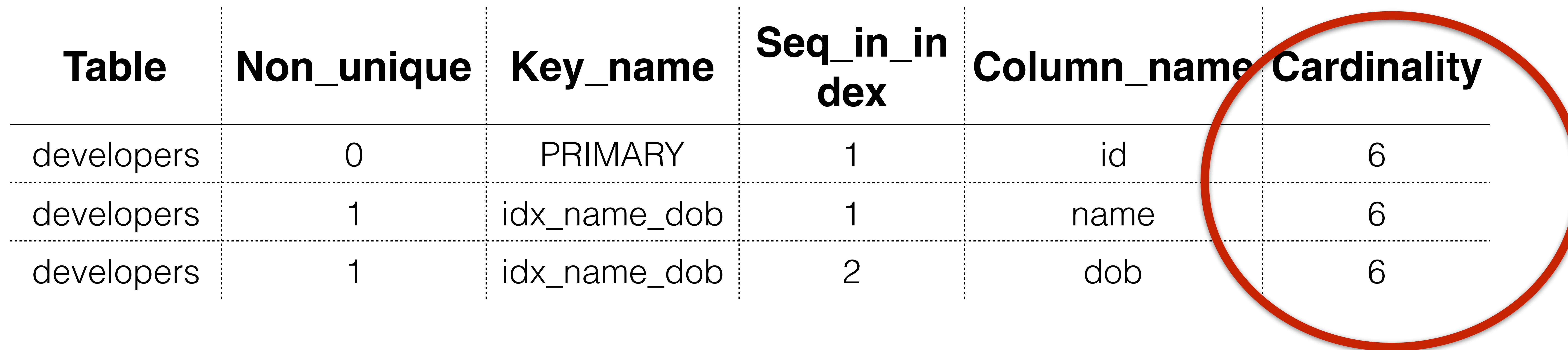


Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
developers	0	PRIMARY	1	id	6
developers	1	idx_name_dob	1	name	6
developers	1	idx_name_dob	2	dob	6

High cardinality!

Compound Index

Refactor the query

```
> EXPLAIN SELECT *
FROM developers
WHERE name = 'Rich'
AND dob < '2001-01-01'
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	range	idx_name_dob	idx_name_dob	772	NULL	1	100.00	Using index condition

idx_name_dob contains all information needed to filter exact result

Compound Index

- Useful for speeding up slow queries with multiple conditions
- Compound index column order is important
- Match indexed column order with WHERE condition order
- Keep equality (**=**) conditions to the left of the query and index
- Keep columns used in range (**>**) conditions to the right
- Consider cardinality of individual columns when choosing compound order (generally greatest first)
- Consider existing compound indexes when writing new queries

Covering Index

```
> CREATE INDEX idx_name_dob ON developers (name, dob)

> EXPLAIN SELECT dob
  FROM developers
  WHERE name = 'Rich'
```

id	select _type	table	partitio ns	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	developers	NULL	ref	idx_name_dob	idx_name _dob	768	const	1	100.00	Using index

dob is the 2nd column in idx_name_dob compound index

MySQL can return dob value directly from the index without reading table data

WHERE conditions must match index column order i.e. **name** first

Covering Index

Tip:

The InnoDB table engine implicitly includes the primary key in secondary indexes

Wat?

```
> CREATE INDEX idx_email ON developers (email)

> EXPLAIN SELECT email
  FROM developers
  WHERE id = 1
```

email is read from idx_email (rather than table) because idx_email implicitly includes id.

Secondary indexes are basically compound indexes using pragma: idx_id_email

Summary

- All apps are different. Consider secondary indexes case by case.
- Use cases evolve and so do number and complexity of queries
- Consider new and existing indexes when writing and refactoring SQL.
- Read / Write query ratio is important. Many indexes can affect write performance.
- Consider how MySQL query optimizer will interpret your queries.
- Benchmark & inspect using EXPLAIN