效能校調

叢集與二級（Secondary）索引

- 資料表中的資料順序會與叢集索引順序一致
- 一個資料表只能有一個叢集索引，主索引就是叢集索引
- 非叢集索引有獨立的索引資料儲存區，透過資料列定位器指向實際資料
- 沒有索引的資料搜尋稱為「資料表掃描」

Explain 欄位

此表為 MariaDB，
MySQL會多幾個欄位

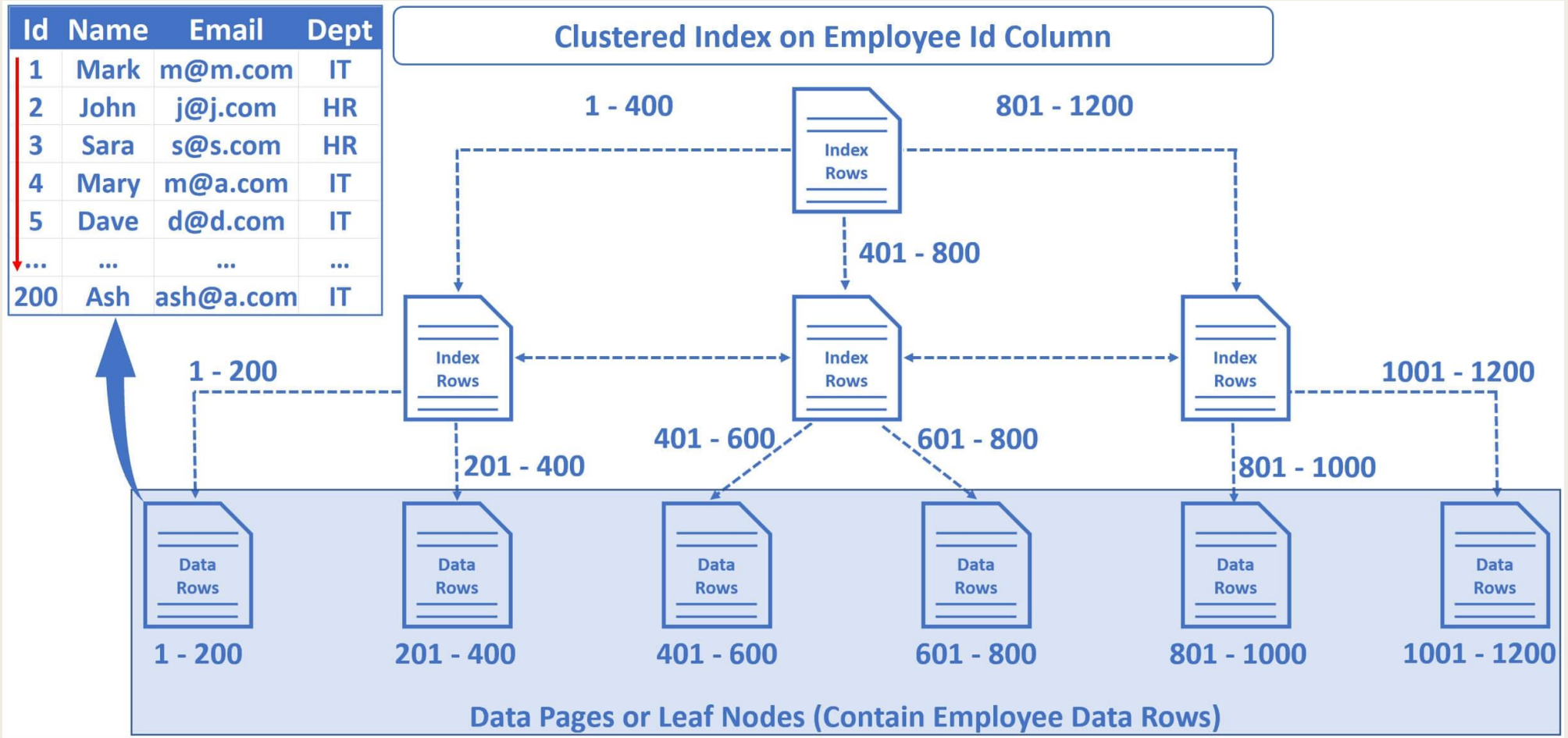
Column	Meaning
id	查詢編號
select_type	查詢型態，SIMPLE 為沒有子查詢的基本查詢
table	目前在查詢的表
type	JOIN 型態
possible_keys	可能使用到的索引
key	實際使用到的索引名稱，NULL表示未使用索引
key_len	使用到的索引長度
ref	使用索引中的哪個欄位查找。const **表示只有一筆資料符合。**
rows	預估列數
Extra	其他額外資訊

Type 欄位

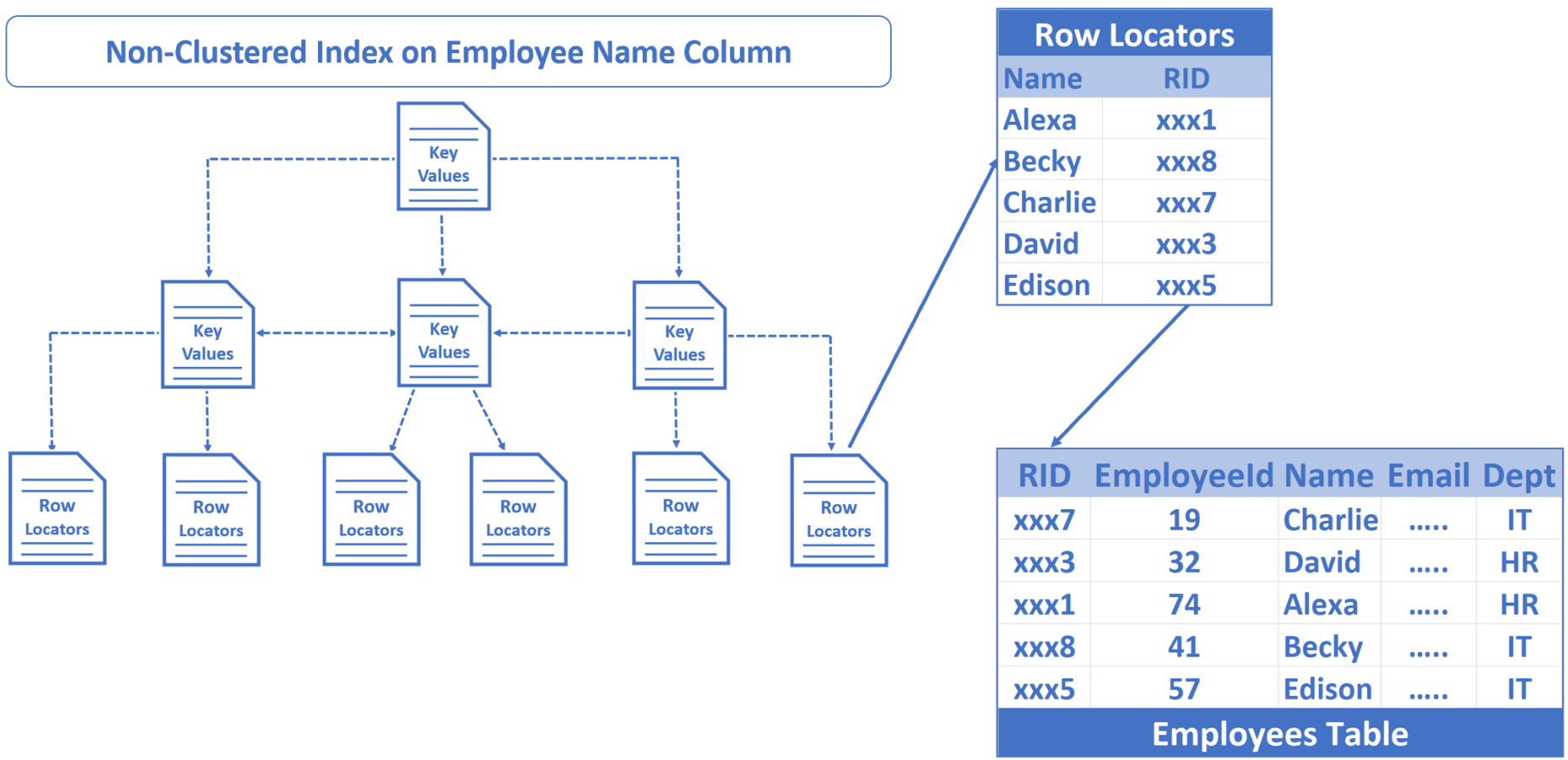
- system：資料表中只有 0 或 1 筆資料
- const：只一筆可能資料符合查詢條件
- eq_ref：使用了 unique index 查詢資料
- ref：使用非 unique index 查詢資料，可能有多筆資料被找到
- ref_or_null：類似 ref，但可以搜尋 NULL
- index_merge：查詢結果使用了多個索引
- range：查詢條件使用了範圍或比較運算
- index：索引掃描，效率很差的查找資料方式 ❌
- ALL：資料表掃描，效率最差的查找資料方式 ❌

All 與 index 都超慢

叢集索引結構


Id Name Email Dept
1 Mark m@m.com IT
2 John j@j.com HR
3 Sara s@s.com HR
4 Mary m@a.com IT
5 Dave d@d.com IT
... ...
200 Ash ash@a.com IT
Clustered Index on Employee Id Column
1 - 400
801 - 1200
Index Rows
401 - 800
Index Rows
Index Rows
Index Rows
1 - 200
1001 - 1200
401 - 600
601 - 800
201 - 400
801 - 1000
Data Rows
Data Rows
Data Rows
Data Rows
Data Rows
Data Rows
1 - 200
201 - 400
401 - 600
601 - 800
801 - 1000
1001 - 1200
Data Pages or Leaf Nodes (Contain Employee Data Rows)

二級索引結構


Non-Clustered Index on Employee Name Column
Key Values
Key Values
Key Values
Key Values
Row Locators
Row Locators
Row Locators
Row Locators
Row Locators
Row Locators
Row Locators
Name RID
Alexa xxx1
Becky xxx8
Charlie xxx7
David xxx3
Edison xxx5
RID EmployeeId Name Email Dept
xxx7 19 Charlie ..... IT
xxx3 32 David ..... HR
xxx1 74 Alexa ..... HR
xxx8 41 Becky ..... IT
xxx5 57 Edison ..... IT
Employees Table

何時使用叢集索引

- JOIN 的欄位（被參考的欄位）
- 當下列的查詢經常是 select * 的時候，建議使用叢集索引
- 用作 >、<、>=、<= 與 between ... and ...
- Order By 或 Group By 欄位，因為他們都跟排序有關

何時使用二級索引

- 使用 JOIN 或 GROUP BY 的欄位
- 用來做搜尋條件的欄位
- 要注意過多的非叢集索引會影響資料異動效能
- 非叢集索引只適合資料筆數少的查詢，若資料筆數多，查詢計畫有可能會改為掃描造成查詢效率變低

測試一

- 將 UserInfo 資料表中的所有索引與主鍵都刪除
- 執行下列指令看查詢計畫

```
explain select * from UserInfo
```

- 為最慢的資料表掃描

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	ALL	*NULL*	*NULL*	*NULL*	*NULL*	6	

測試二

- 將 UserInfo 的 uid 設定為主索引
- 執行下列指令

```
explain select * from UserInfo where uid = 'A06'
```

- 使用了索引

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	const	PRIMARY	PRIMARY	62	const	1	

測試三

- 將 UserInfo 的 uid 設為 PK，cname 設為 index
- 比較下列指令

```
explain
select * from UserInfo where uid = 'A03' or cname = '王大明'
```

- 與

```
explain
select * from UserInfo where uid = 'A03'
union all
select * from userInfo where cname = '王大明'
```

- 哪個好？

查詢條件使用了函數

- 對欄位使用函數或運算時，就會採用索引掃描，即便該欄位設了索引也一樣
- 將 UserInfo 的 PK 移除，僅對 cname 設索引

```
select cname from UserInfo where left(cname, 1) = '王'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	index	*NULL*	idx_cname	138	*NULL*	6	Using where; Using index

- 應改成

```
select cname from UserInfo where cname like '王%'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	range	idx_cname	idx_cname	138	*NULL*	2	Using index condition

小心使用 LIKE

- 使用 LIKE 時，% 放在前面索引就發揮不了作用

```
select cname from UserInfo where cname like '%王%'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	ALL	*NULL*	*NULL*	*NULL*	*NULL*	6	Using where

索引不一定都會使用

- 有時建立了索引，但查詢時會依據實際狀況，並不一定會使用到索引
- 將 UserInfo 的 uid 設定 PK，cname 設定二級索引
- 執行下列指令，感覺上會使用到索引，事實上還是在索引掃描

```
select cname from UserInfo where cname = '王小毛'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	index	idx_cname	idx_cname	138	*NULL*	6	Using where; Using index

- 原因：資料太少，此時不用索引比用索引快
- 當資料多的時候（例如五百筆資料），就會真正使用到索引搜尋

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	range	idx_cname	idx_cname	138	*NULL*	2	Using where; Using index

還有那些字串處理函數

- 子字串處理：LEFT()、RIGHT()、SUBSTRING()
- 頭尾去空白：LTRIM()、RTRIM()、TRIM()
- 大小寫：UPPER()、LOWER()
- 字串相加：CONCAT()
- 小心使用這些函數，使用在查詢條件中的欄位時，就不會使用索引
- 例如 UserInfo 無 PK，且 cname 設定二級索引

```
explain
select cname from UserInfo where trim(cname) = '王大明'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	index	*NULL*	idx_cname	138	*NULL*	506	Using where; Using index

指定索引

- 如果發現資料庫挑選的索引不正確，可以下指令要求使用特定索引

```
select * from UserInfo use index (primary)
```

強制要求使用主索引

- 不使用索引

```
select * from UserInfo use index ()
```

複合欄位索引

稱為主要欄位

- 當索引包含兩個以上欄位時，查詢條件須含索引中「第一個欄位」才會使用到索引
- 例如一個資料表有三個欄位：a、b、c，設定這三個欄位的複合欄位索引。
- 當查詢條件有 a 欄位時，就會用到索引，例如

```
select * from NewTable where a = 20                ✅
select * from NewTable where a = 20 and c = 10     ✅
select * from NewTable where b = 20 or c = 10      ❌
select * from NewTable where b = 10                ❌
```

複合欄位索引與排序

- 索引中的欄位排序會跟查詢結果排序有關，例如：每個欄位的排序分別是：a(+)、b(+)、c(+) => 目前 MySQL 與 MariaDB 都只有正向排序
- 根據排序列出下表

  a(+)、b(+)、c(+)
  a(+)、b(+)
  a(+)
  a(-)
  a(-)、 b(-)
  a(-)、 b(-)、c(-)

- 只要 Order By 的欄位符合這些順序中的一項，就會使用索引中的排序，例如

```
select * from NewTable order by a desc, b desc
```
✅

```
select * from NewTable order by a desc, b
```
❌

JOIN – 1/3

- 移除 UserInfo、Live、House 主鍵，並且不設定任何索引

```
explain select UserInfo.uid, cname, address
from UserInfo, Live, House
where
   UserInfo.uid = Live.uid and Live.hid = House.hid
   and cname = '王大明'
```

- 三個資料表均使用最慢的資料表掃描

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Live	ALL	*NULL*	*NULL*	*NULL*	*NULL*	4	
1	SIMPLE	House	ALL	*NULL*	*NULL*	*NULL*	*NULL*	4	Using where; Using join buffer (flat, BNL join)
1	SIMPLE	UserInfo	ALL	*NULL*	*NULL*	*NULL*	*NULL*	6	Using where; Using join buffer (incremental, BNL join)

JOIN – 2/3

- 設定 PK：UserInfo(uid)、Live(uid + hid)、House(hid)
- 設定 Index：UserInfo(cname)
- 三個資料表的資料筆數多加一些，太少可能不會觸發索引搜尋
- 現在三個資料表都變成快速的搜尋

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	UserInfo	ref	PRIMARY,idx_cname	idx_cname	138	const	1	Using where; Using index
1	SIMPLE	Live	ref	PRIMARY	PRIMARY	62	addressbook.UserInfo.uid	1	Using index
1	SIMPLE	House	eq_ref	PRIMARY	PRIMARY	4	addressbook.Live.hid	1	

JOIN – 3/3

- House 資料表中的 address 欄位設定 unique index
- 查詢條件改為地址

```
select UserInfo.uid, cname, address
from UserInfo, Live, House
where
   UserInfo.uid = Live.uid and Live.hid = House.hid
   and address = '台北市南京東路1號'
```

- 為什麼 Live 資料表變成索引掃描？如何校調？

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	House	const	PRIMARY,address	address	603	const	1	Using index
1	SIMPLE	Live	index	PRIMARY	PRIMARY	66	*NULL*	504	Using where; Using index
1	SIMPLE	UserInfo	eq_ref	PRIMARY	PRIMARY	62	addressbook.Live.uid	1	

新增與索引

- 沒有索引與只有叢集索引的兩資料表，在新增資料時兩者效能完全一樣
- 當資料表的二級索引越多，新增資料時效能就越低

更新與索引

- 更新指令具有 where 條件，這裡會受到索引影響
- 更新指令所更新的欄位會影響與之有關的索引，並不會影響所有索引

刪除與索引

- 刪除指令會影響所有索引，並且 where 部分會受到索引而影響效能
- 刪除所有資料時，使用 truncate 指令，效能遠高於 delete
- truncate 刪除資料後，不會維護相關索引，但 delete 會

最佳化索引

- 當資料庫用久了，索引就會變的破碎，此時就需要重建索引來提升查詢效率
- 重建指令為

```
OPTIMIZE TABLE my_table
```

- 或

```
ALTER TABLE my_table ENGINE=InnoDB
```

- 何時執行
  - *建議大量新增與刪除資料後執行*