B+Tree Indexes and InnoDB

Ovais Tariq
Percona Live London 2011

Agenda

- Why Index
- What is a B+Tree
- B+Tree Characteristics
- Cost Estimation Formulae
- A Few Advantages
- B+Tree Index in InnoDB
- Primary and Secondary B+Tree Indexes
- Characteristics of an Ideal Primary Index

Agenda (Cont ..)

- In-order INSERTs vs Random-order INSERTs
- Composite Indexes
- B+Tree and Index Prefix
- Index Selectivity
- Speeding up Secondary Indexes
- Tips and Take-away
- Percona Live London Sponsors
- Annual MySQL Users Conference

Why Index?

- Linear search is very slow, complexity is O(n)
- Indexes are used in variety of DBMS
- Many different type of indexes
  - Hash Indexes (only MEMORY SE and NDB)
  - Bitmap Indexes (not available in MySQL)
  - BTree Indexes and derivates (MyISAM, InnoDB)
- Indexes improve search performance
- But add extra cost to INSERT/UPDATE/DELETE

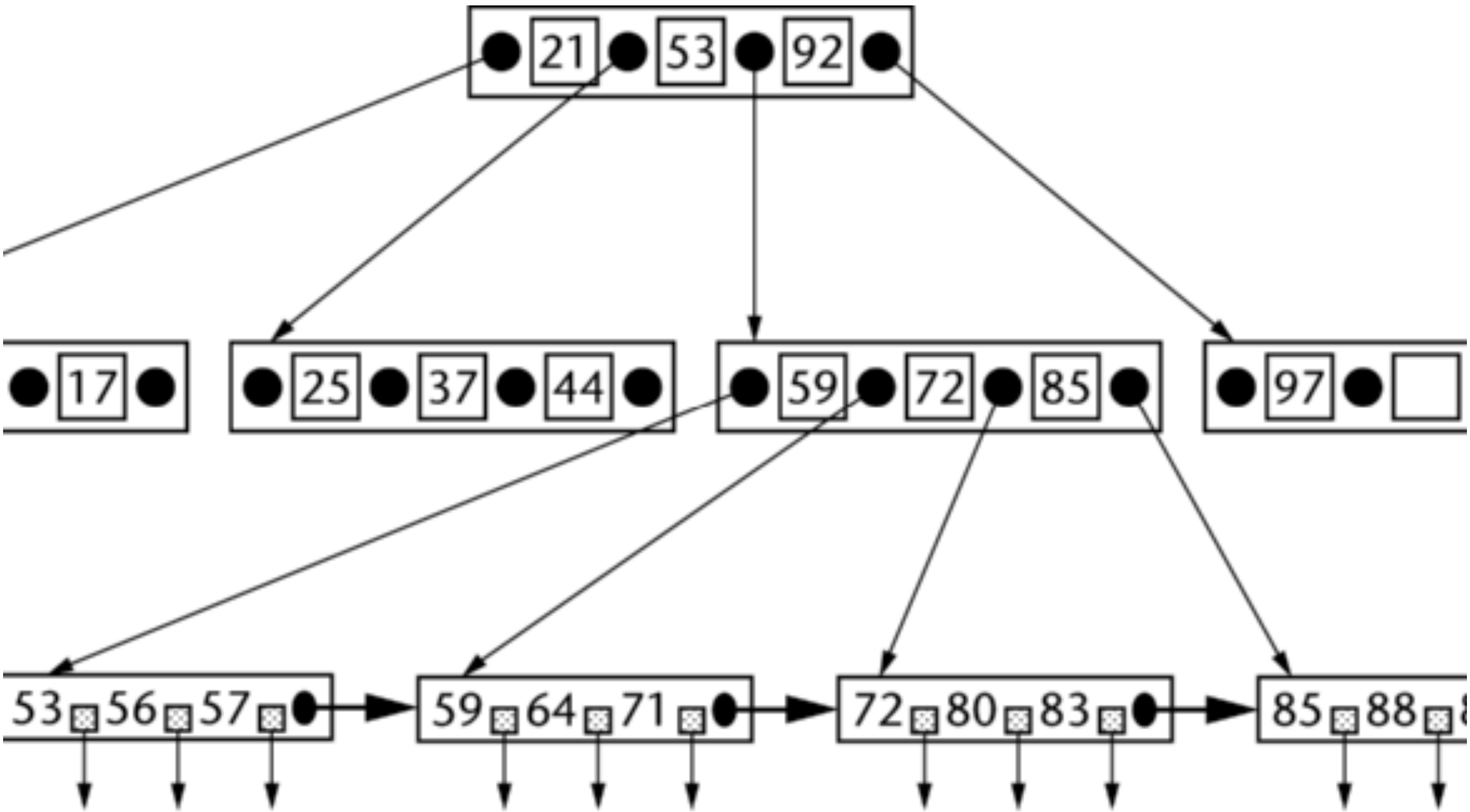
What is a B+Tree

- A generalized version of Binary Search Tree http://en.wikipedia.org/wiki/Binary_search_tree
- Classic disk based structure for indexing records based on an ordered key set
- Reading a single record from a very large table, results in only a few pages of data being read
- Any index structure other then B+Tree is subject to overflow

B+Tree Characteristics

- Every node can have $p - 1$ key values and p node pointers (p is called the order of the tree)
- The leaf node contains data, internal nodes are only used as guides
- The leaf nodes are connected together as doubly linked list
- Keys are stored in the nodes in sorted order
- All leaf nodes are at the same height, that's why it's called a balanced tree

Typical B+Tree structure



Cost Estimation Formulae

- Some Assumptions
- Cost Calculations
- A Few Extra Considerations

Some Assumptions

- h is the height of the tree
- p is the branching factor of the tree
- n is the number of rows in a table
- p = (page size in bytes/key length in bytes) + 1
- $h > \log n / \log p$

Cost Calculations

- Search cost for a single row
  - S = h I/O ops
- Update cost for a single row
  - U = search cost + rewrite data page = $h + 1$ I/O ops
- Insert cost for a single row
  - I = search cost + rewrite index page + rewrite data page
  - I = $h + 1 + 1 = h + 2$ I/O ops

Cost Calculations (Cont ..)

- Delete cost for a single row
  - D = search cost + rewrite index page + rewrite data page
  - D = h + 1 + 1 = h + 2 I/O ops

A Few Extra Considerations

- Updates are in place only if the new data is of the same size, otherwise its delete plus insert
- Inserts may require splits if the leaf node is full
- Occasionally the split of a leaf node necessitates split of the next higher node
- In worst case scenarios the split may cascade all the way up to the root node
- Deletions may result in emptying a node that necessitates the consolidation of two nodes

A Few Advantages

- Reduced I/O
- Reduced Rebalancing
- Extremely efficient range scans
- Implicit sorting

Reduced I/O

- Height of a B+Tree is very small (and has a very large branching factor)
- Generally every node in a tree corresponds to a page of data (page size ranges from 2^{11} to 2^{14} bytes)
- A node read = read a page = 1 random I/O
- So to reach leaf node, we need to read *h* pages
- No matter if requested row is at the start or end of table, same number of I/O is needed

Reduced Rebalancing

- A tree needs rebalancing after an insertion or deletion
- B+Tree is wide, more keys can fit in node, so rebalancing needed few times on insertions and deletions
- Note that rebalancing means extra I/O, so rebalancing saved is I/O saved

Extremely Efficient Range Scans

- Leaf nodes are linked together as doubly linked list
- So need to traverse from root -> leaf just once
- Move from leaf -> leaf until you reach the end of range
- Entire tree may be scanned without visiting the higher nodes at all

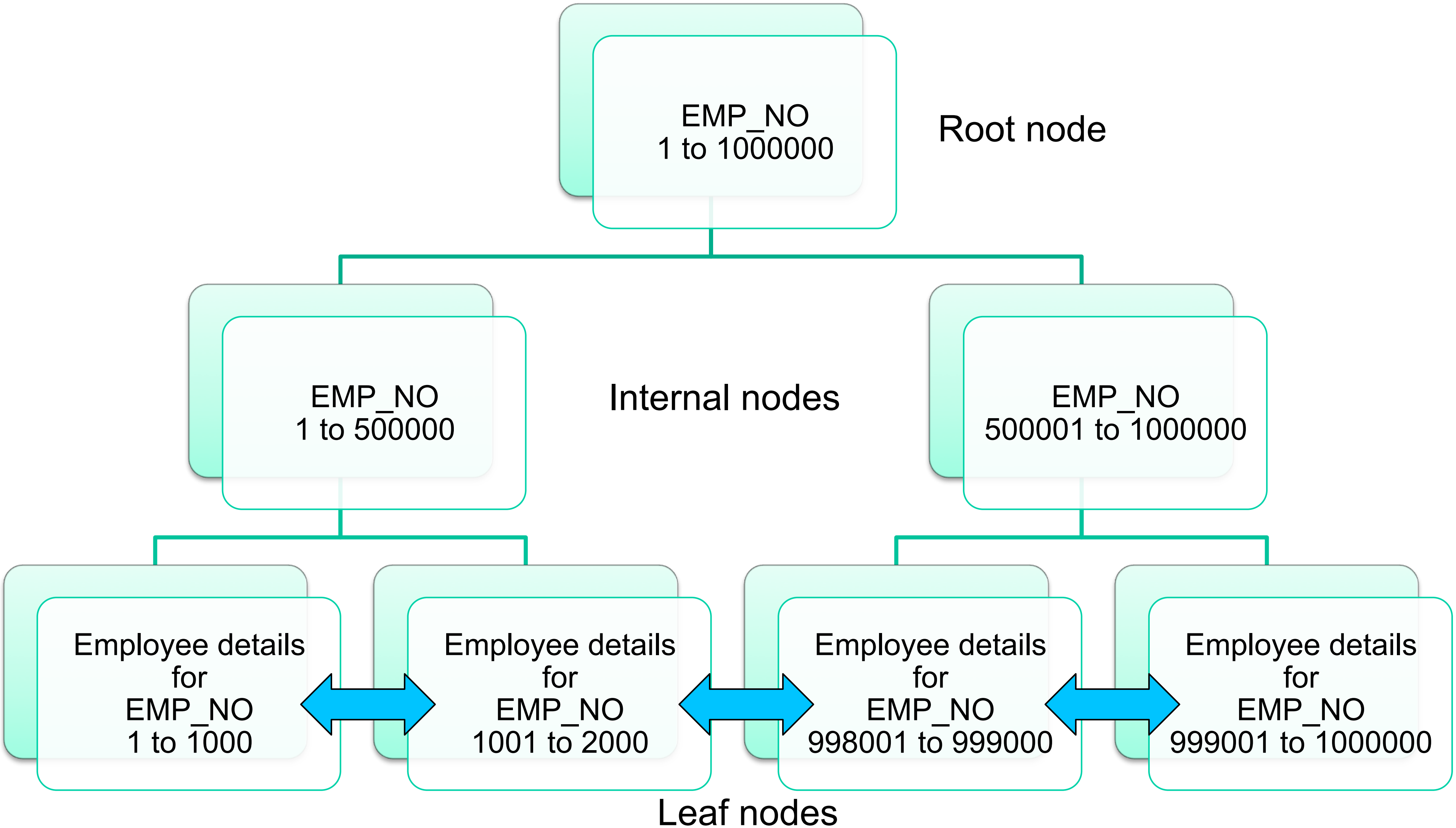
Implicit Sorting

- Nodes contain keys sorted in key-order
- Therefore records can be implicitly returned in sorted order
- No external sorting needed hence memory and CPU cycles saved
- Sometimes sorted data cannot fit into buffer, and data needs to be sorted in passes, needing I/O, which can be avoided if you need data in key order

B+Tree Index in InnoDB

- B+Tree Index in InnoDB is a typical B+Tree structure, no strings attached!
- Leaf nodes contain the data (what the data is depends whether it's a Primary Index or a Secondary Index)
- Root nodes and internal nodes contain only key values

A Typical Index



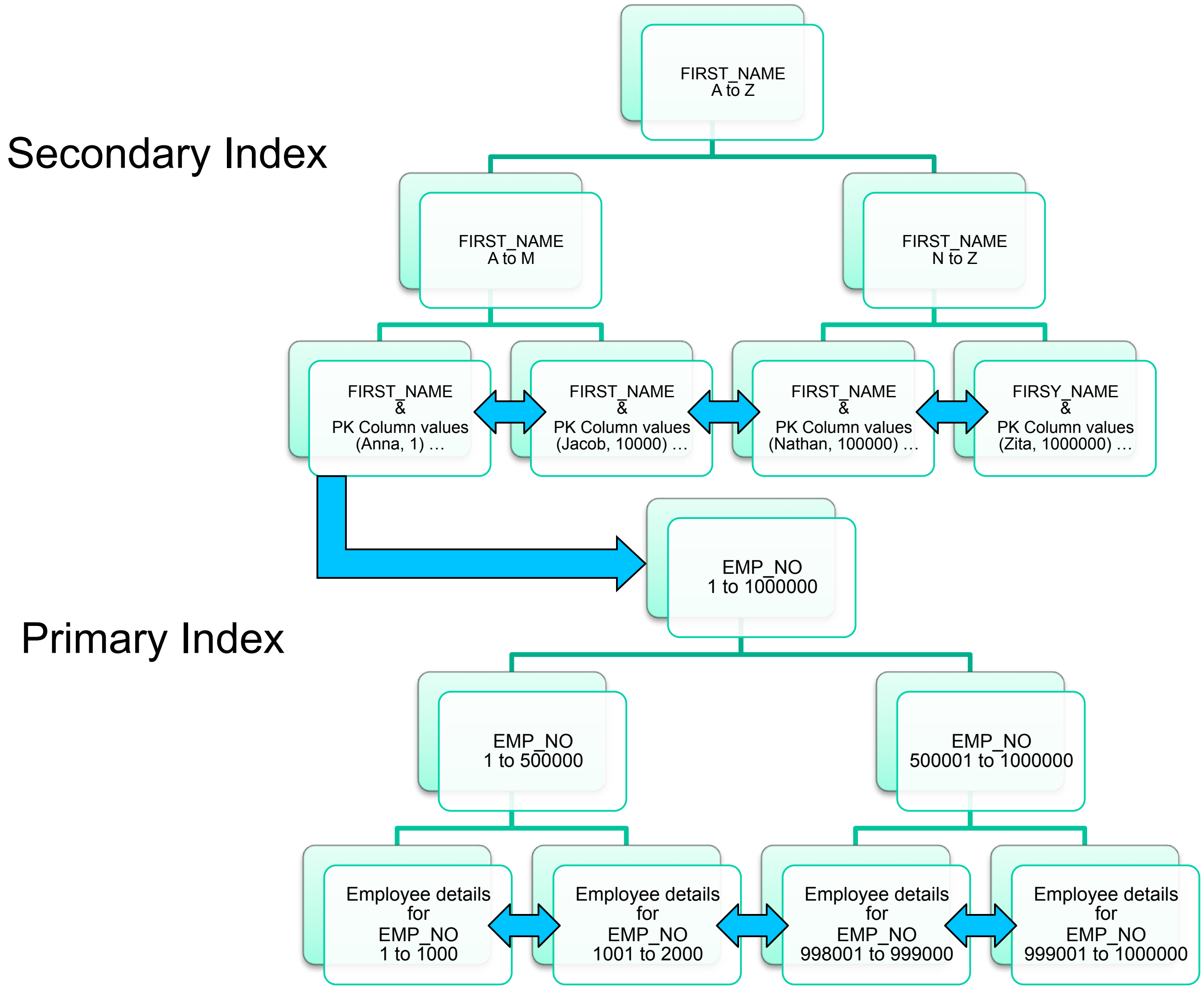
Primary and Secondary B+Tree Indexes

- Primary index holds the entire row data in its leaf nodes
- Primary index can also be called a clustered index, because data is clustered around PK values
- A single PK per table means, a single clustered index per table
- Secondary Indexes have the key values and PK values in the index and no row data

Primary and Secondary B+Tree Indexes (Cont ..)

- PK values stored in the leaf nodes of a secondary index act as pointer to the data
- This means secondary index lookups are two lookups
- Cost of secondary index lookup
  - C = Height of Secondary Index B+Tree + Height of Primary Index B+Tree

A Typical Secondary Index



Characteristics of an Ideal Primary Index

- Create primary index on column(s) that are not updated too often
- Keep the size of the primary index as small as possible
- Select the column(s) to create primary index on, that have sequentially increasing value
- Random value columns, such as those that store UUID, are very bad candidates for primary index

In-order INSERTs vs Random-order INSERTs

- In-order INSERTs result in good page fill percentage, meaning InnoDB can keep on inserting in the same page till its full
- Good insert speed, good page fill percentage
- Reduced page and extent fragmentation

Leaf Page 001	Leaf Page 002	Leaf Page 100
(1, 'Alister') (2, 'Anna') ….. (100, 'Cathy')	(101, 'Celvin') (102, 'Donald') ….. (200, 'Frank')	(10001, 'Stan') (10002, 'Steve') ….. (10100, 'Suzzan')

In-order INSERTs vs Random-order INSERTs (Cont ..)

- Random-order inserts introduce overhead
- Result in page and extent fragmentation
- Bad insert speed and bad page fill percentage (resulting in wasted space)
- Data is not actually physically clustered together
- Scanning ranges do not result in pages read in sequential order
- That is why UUID is not a good PK candidate

In-order INSERTs vs Random-order INSERTs (Cont ..)

Extent Fragmentation

Leaf Page 101

(10101, ‘Alister’)
(10102, ‘Anna’)
…..
(10200, ‘Cathy’)

Leaf Page 96

(10201, ‘Celvin’)
(10202, ‘Donald’)
…..
(10300, ‘Frank’)

Leaf Page 102

(10301, ‘Stan’)
(10302, ‘Steve’)
…..
(10400, ‘Suzzan’)

Page Fragmentation

Leaf Page 101

(10101, ‘Alister’)

Leaf Page 102

(10201, ‘Celvin’)
(10202, ‘Donald’)

Leaf Page 103

(10301, ‘Stan’)
(10302, ‘Steve’)
…..
(10400, ‘Suzzan’)

Example Schema

```
CREATE TABLE `employees` (
  `emp_no` int(11) NOT NULL,
  `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,
  PRIMARY KEY (`emp_no`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Composite Indexes

- A single index can be defined on more than one column
- The index key is composed of more than one key value
- Let’s consider a query
  - SELECT emp_no, first_name, last_name
    FROM employees
    WHERE hire_date = '1985-03-22'
    AND last_name = 'Peek';

Composite Indexes (Cont ..)

- One way of indexing is to create two separate indexes on hire_date and last_name columns
- Search cost would be
  - S = h(hire_date) + h(last_name) + Merge and Intersect cost
  - $S = h_1 + h_2$ I/O ops + Merge and Intersect cost
- Consider if you create a composite index (hire_date, last_name)
- The cost of composite index is one lookup

Composite Indexes (Cont ..)

- Composite index will not require extra merge-intersect step, hence saving memory and CPU cycles
- To generalize, if a composite index has *k* columns, then equivalent cost in single value indexes is *k* index lookups
- Similarly, *k* single value indexes will need more pages to be read into memory

Composite Indexes (Cont ..)

- Suppose you have
  - 1000 rows match hire_date = '1985-03-22'
  - 1000 rows that match last_name = 'Peek'
  - 4 rows that match both conditions
- See less pages to load into memory when using composite index

B+Tree and Index Prefix

- By design B+Tree can only work with filters that filter at least on the prefix of the index
- So an index idx(first_name, last_name) can only be used for following searches
  - SELECT … WHERE first_name = x AND last_name = y
  - SELECT … WHERE first_name = x
- And cannot be used for the following search
  - SELECT … WHERE last_name = y

B+Tree and Index Prefix (Cont ..)

- Same rule also hold for single column indexes
- So an index idx(first_name) can only be used for following searches
  - SELECT … WHERE first_name LIKE ‘ova%’
- But cannot be used for following
  - SELECT … WHERE first_name LIKE ‘%ais’
- If an index cannot be used that means a table scan

Index Selectivity

- What is selectivity?
- Selectivity = unique values / total no. of records
- Primary Index is the most selective index
- Suppose you index a column that stores gender, meaning only two distinct values
- Remember secondary index only store a pointer to the data in the primary index
- Indexing a gender column means each key value with thousands of PK pointers

Index Selectivity (Cont ..)

- Each pointer lookup will be a random PK lookup
- Its much better to scan the PK in order and filter by gender
- But you can improve the selectivity of a column by combining it with other columns and creating a composite index

Speeding up Secondary Indexes

- Remember secondary indexes only store PK pointers meaning two index lookups
- Performance can be dramatically improved if we avoid extra PK lookups
- The trick is to include all the columns queried, in the definition of the secondary index
- Example query
    - SELECT emp_no, first_name, last_name
      FROM employees WHERE hire_date = '1985-03-22'
      AND last_name = 'Peek';

Speeding up Secondary Indexes (Cont ..)

- Originally the index is idx(hire_date, last_name)
- Let's try to modify the index
  - No need to add the emp_no column as its PK
  - Add first_name column to right of index definition
  - idx(hire_date, last_name, first_name)
- Note we add the column to the right of index definition, remember B+Tree can only filter on prefix of index
- This is known as covering index optimization

Tips and Take-away

- Indexing should always be used to speed up access
- Index trade-off analysis can be done easily using the cost estimation formulae discussed
- Select optimal data types for columns, especially ones that are to be indexed – int vs bigint
- When selecting columns for PK, select those that would make the PK short, sequential and with few updates

Tips and Take-away (Cont ..)

- Avoid using UUID style PK definitions
- Insert speed is best when you insert in PK order
- When creating index on string columns, you don't need to index the entire column, you can index a prefix of the column – idx(str_col(4))
- B+Tree indexes are only suitable for columns with good selectivity
- Don't shy away from creating composite indexes

Percona Live London Sponsors

Platinum Sponsor

Gold Sponsor

Silver Sponsors

Percona Live London Sponsors

Exhibitor Sponsors

BOOKING.COM online hotel reservations

SCALEBASE Your Database. Transparently scaled.

Friends of Percona Sponsors

Couchbase

devart All Flavors of MySQL Management and Connectivity

From Dual

ivinco

Monty Program

Sphinx

Tokutek

Webyog

Media Sponsors

apress

Drizzle

PalominoDB

O'REILLY

Annual MySQL Users Conference

Presented by Percona Live

The Hyatt Regency Hotel, Santa Clara, CA

April 10th-12th, 2012

Featured Speakers

Mark Callaghan, Facebook

Jeremy Zawodny, Craigslist

Marten Mickos, Eucalyptus Systems

Sarah Novotny, Blue Gecko

Peter Zaitsev, Percona

Baron Schwartz, Percona

The Call for Papers is Now Open!

Visit www.percona.com/live/mysql-conference-2012/

ovais.tariq@percona.com
@ovaistariq

We're Hiring! www.percona.com/about-us/careers/

PERCONA LIVE

www.percona.com/live